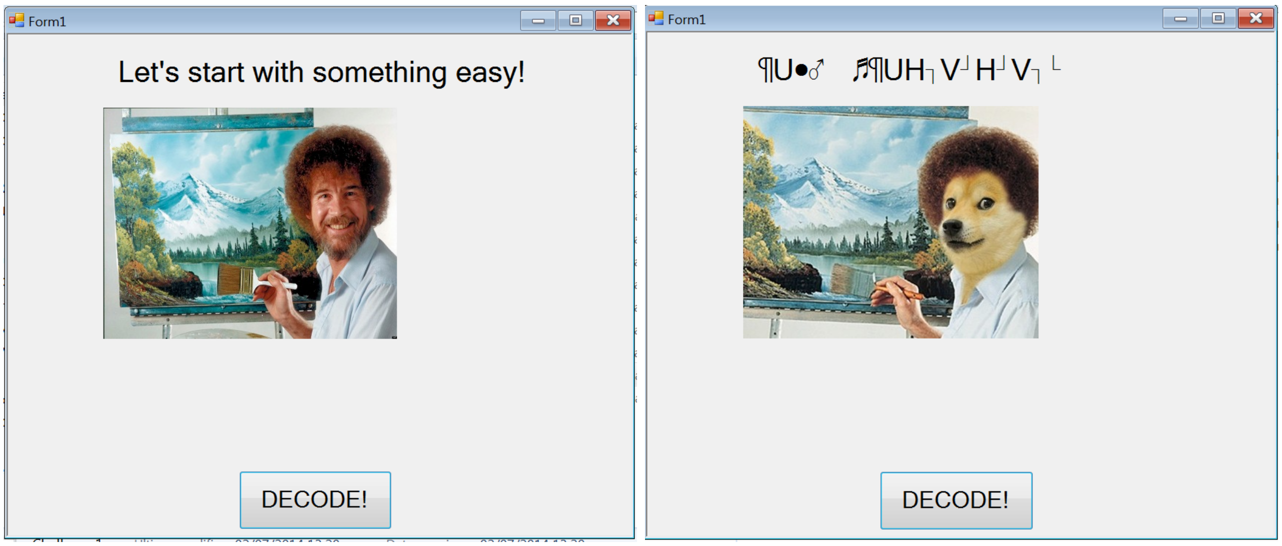
Flare-On 2014

Challenge 1

flag: 3rmahg3rd.b0b.d0ge@flare-on.com
tools: dnSpy

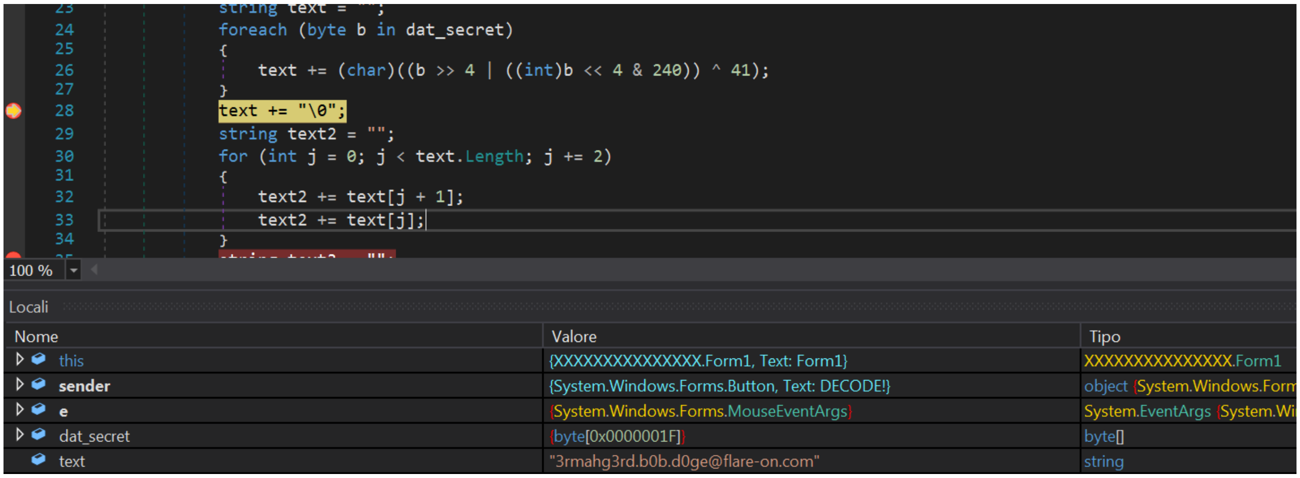
The first executable shows a Bob Ross picture with the text *"Let's start with something easy!"* and a button labeled "DECODE!". Once the button is clicked, a string made up (apparently) of "garbage" characters will be printed.



The sample is written in `.NET`, so it can be disassembled by using **dnSpy**. We can see that the function associated with the button object is called `btnDecode_Click`. This function will decrypt data from a buffer called `dat_secret` in a three-stage decoding process, as shown below.

```
{
    this.pbRoge.Image = Resources.bob_roge;
    byte[] dat_secret = Resources.dat_secret;
    string text = "";
    foreach (byte b in dat_secret)
    {
        text += (char)((b >> 4 | ((int)b << 4 & 240)) ^ 41);
    }
    text += "\0";
    string text2 = "";
    for (int j = 0; j < text.Length; j += 2)
    {
        text2 += text[j + 1];
        text2 += text[j];
    }
    string text3 = "";
    for (int k = 0; k < text2.Length; k++)
    {
        char c = text2[k];
        text3 += (char)((byte)text2[k] ^ 102);
    }
    this.lbl_title.Text = text3;
}
```

We already know that the final output is not what we are looking for, which means that the flag must be hidden somewhere in between. Place a breakpoint after each string is created and run the debbugger: the flag will be in the `text` variable, as shown below.



Challenge 2

flag: a11.that.java5crap@flare-on.com
tools: HxD, Python

The second challenge is a zip archive containing a html file and a png image. Looking at the source code of the html file, one particulare line stood out (shown in the picture below). A png image being loaded as php code? Not suspicious at all...

```
t><?php include "img/flare-on.png" ?>
```

Loading the image in a hex editor (I used **HxD**) will reveal the embedded php code, as shown in the excerpt below.

```
flare-on.png
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Testo decodificato
000019C0  AE 42 60 82 3C 3F 70 68 70 20 24 74 65 72 6D 73  ®B`‚<?php $terms
000019D0  3D 61 72 72 61 79 28 22 4D 22 2C 20 22 5A 22 2C  =array("M", "Z",
000019E0  20 22 5D 22 2C 20 22 70 22 2C 20 22 5C 5C 22 2C   "]", "p", "\\",
000019F0  20 22 77 22 2C 20 22 66 22 2C 20 22 31 22 2C 20   "w", "f", "1", 
00001A00  22 76 22 2C 20 22 3C 22 2C 20 22 61 22 2C 20 22  "v", "<", "a", "
00001A10  51 22 2C 20 22 7A 22 2C 20 22 20 22 2C 20 22 73  Q", "z", " ", "s
00001A20  22 2C 20 22 6D 22 2C 20 22 2B 22 2C 20 22 45 22  ", "m", "+", "E"
00001A30  2C 20 22 44 22 2C 20 22 67 22 2C 20 22 57 22 2C  , "D", "g", "W",
00001A40  20 22 5C 22 22 2C 20 22 71 22 2C 20 22 79 22 2C   "\"", "q", "y",
00001A50  20 22 54 22 2C 20 22 56 22 2C 20 22 6E 22 2C 20   "T", "V", "n", 
00001A60  22 53 22 2C 20 22 58 22 2C 20 22 29 22 2C 20 22  "S", "X", ")", "
00001A70  39 22 2C 20 22 43 22 2C 20 22 50 22 2C 20 22 72  9", "C", "P", "r
```

The extracted code was heavily obfuscated: there were actually several stages of obfuscation that combined different techniques, like base64 encoding and strings manipulation.

```php
<?php $terms=array("M", "Z", "]", "p", "\\", "w", "f", "1", "v", "<", "a", "Q", "z", " ", "s", "m", "+", "E", "D", "g", "W", "\"", "q", "y", "T", "V", "n",
 ")", "9", "C", "P", "r", "&", "\'", "!", "x", "G", ":", "2", "~", "O", "h", "u", "U", "@", ";", "H", "3", "F", "6", "b", "L", ">", "^", ",", ".", "l", "$",
, "%", "N", "*", "[", "0", "}", "J", "-", "5", "_", "A", "=", "{", "k", "o", "7", "#", "i", "I", "Y", "(", "j", "/", "?", "K", "c", "B", "t", "R", "4", "8",
);
$order=array(59, 71, 73, 13, 35, 10, 20, 81, 76, 10, 28, 63, 12, 1, 28, 11, 76, 68, 50, 30, 11, 24, 7, 63, 45, 20, 23, 68, 87, 42, 24, 60, 87, 63, 18, 58, 87
, 58, 87, 63, 83, 43, 87, 93, 18, 90, 38, 28, 18, 19, 66, 28, 18, 17, 37, 63, 58, 37, 91, 63, 83, 43, 87, 42, 24, 60, 87, 93, 18, 87, 66, 28, 48, 19, 66, 63
 91, 63, 17, 1, 87, 93, 18, 45, 66, 28, 48, 19, 40, 11, 25, 5, 70, 63, 7, 37, 91, 63, 12, 1, 87, 93, 18, 81, 37, 28, 48, 19, 12, 63, 25, 37, 91, 63, 83, 63,
18, 87, 23, 28, 18, 75, 49, 28, 48, 19, 49, 0, 50, 37, 91, 63, 18, 50, 87, 42, 18, 90, 87, 93, 18, 81, 40, 28, 48, 19, 40, 11, 7, 5, 70, 63, 7, 37, 91, 63,
87, 93, 18, 81, 7, 28, 48, 19, 66, 63, 50, 5, 40, 63, 25, 37, 91, 63, 24, 63, 87, 63, 12, 68, 87, 0, 24, 17, 37, 28, 18, 17, 37, 0, 50, 5, 40, 42, 50, 5, 49
 5, 91, 63, 50, 5, 70, 42, 25, 37, 91, 63, 75, 1, 87, 93, 18, 1, 17, 80, 58, 66, 3, 86, 27, 88, 77, 80, 38, 25, 40, 81, 20, 5, 76, 81, 15, 50, 12, 1, 24, 81
 40, 90, 58, 81, 40, 30, 75, 1, 27, 19, 75, 28, 7, 88, 32, 45, 7, 90, 52, 80, 58, 5, 70, 63, 7, 5, 66, 42, 25, 37, 91, 0, 12, 50, 87, 63, 83, 43, 87, 93, 18
 28, 48, 19, 7, 63, 50, 5, 37, 0, 24, 1, 87, 0, 24, 72, 66, 28, 48, 19, 40, 0, 25, 5, 37, 0, 24, 1, 87, 93, 18, 11, 66, 28, 18, 87, 70, 28, 48, 19, 7, 63, 5
 0, 18, 1, 87, 42, 24, 60, 87, 0, 24, 17, 91, 28, 18, 75, 49, 28, 18, 45, 12, 28, 48, 19, 40, 0, 7, 5, 37, 0, 24, 90, 87, 93, 18, 81, 37, 28, 48, 19, 49, 0,
40, 63, 25, 5, 91, 63, 50, 5, 37, 0, 18, 68, 87, 93, 18, 1, 18, 28, 48, 19, 40, 0, 25, 5, 37, 0, 24, 90, 87, 0, 24, 72, 37, 28, 48, 19, 66, 63, 50, 5, 40, 63
, 91, 63, 24, 63, 87, 63, 12, 68, 87, 0, 24, 17, 37, 28, 48, 19, 40, 90, 25, 37, 91, 63, 18, 90, 87, 93, 18, 90, 38, 28, 18, 19, 66, 28, 18, 75, 70, 28, 48,
90, 58, 37, 91, 63, 75, 11, 79, 28, 27, 75, 3, 42, 23, 88, 30, 35, 47, 59, 71, 71, 73, 35, 68, 38, 63, 8, 1, 38, 45, 30, 81, 15, 50, 12, 1, 24, 81, 66, 28, 4
58, 81, 40, 30, 75, 1, 27, 19, 75, 28, 23, 75, 77, 1, 28, 1, 43, 52, 31, 19, 75, 81, 40, 30, 75, 1, 27, 75, 77, 35, 47, 59, 71, 71, 71, 73, 21, 4, 37, 51, 40
91, 7, 4, 37, 77, 49, 4, 7, 91, 70, 4, 37, 49, 51, 4, 51, 91, 4, 37, 70, 6, 4, 7, 91, 91, 4, 37, 51, 70, 4, 7, 91, 49, 4, 37, 51, 6, 4, 7, 91, 91, 4, 37, 51,
 47, 93, 8, 10, 58, 82, 59, 71, 71, 71, 82, 59, 71, 71, 29, 29, 47);
$do_me="";
for ($i=0;$i<count($order);$i++)
{$do_me=$do_me.$terms[$order[$i]];}
eval($do_me); ?>
```

I created a custom python script to help deobfuscate the payloads and get to the final stage: once you manage to print the flag, you will need to add the proper punctuation (replace "DOT" with ".", "DASH" with "-" and so on).

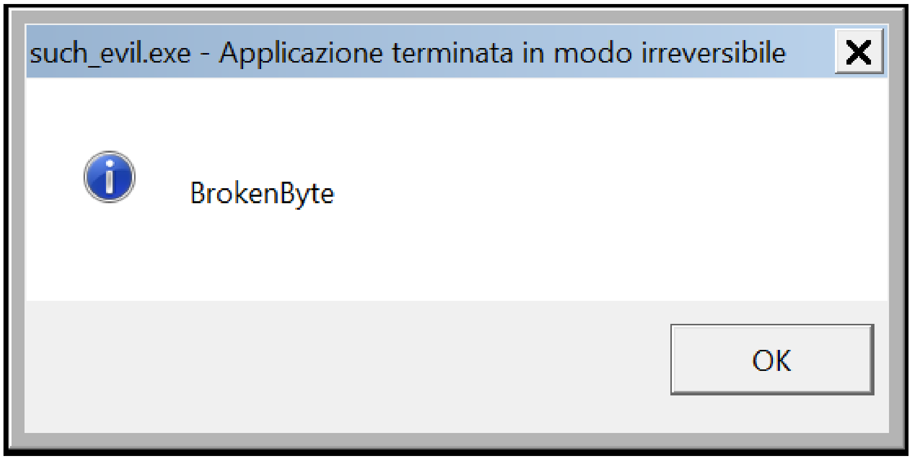
```
C:\Users\anfl\Desktop\flare>python final_stage.py
a11DOTthatDOTjava5crapATflareDASHonDOTcom

C:\Users\anfl\Desktop\flare>python final_stage.py
a11.that.java5crap@flare-on.com
```

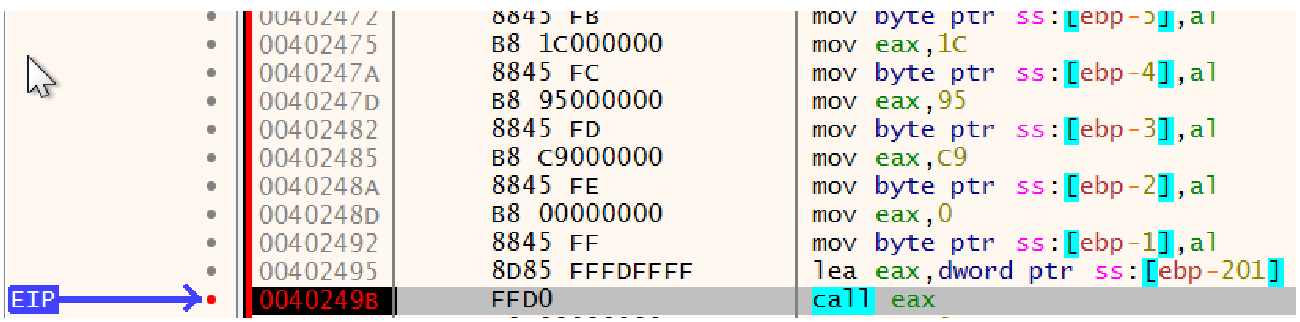
Challenge 3

flag: such.5h311010101@flare-on.com
tools: x32dbg

The sample for this challenge is called `suchevil.exe`. Once executed, the application will show the following error message and then close.

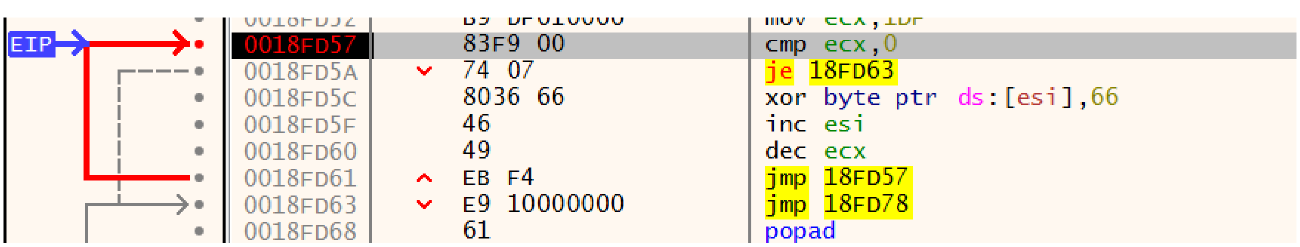


I used **x32dbg** to debug the sample. The executable code starts at `00401000`, where you can see several `MOV` instructions to push data onto the stack, followed by a dynamic function call (`call eax`).

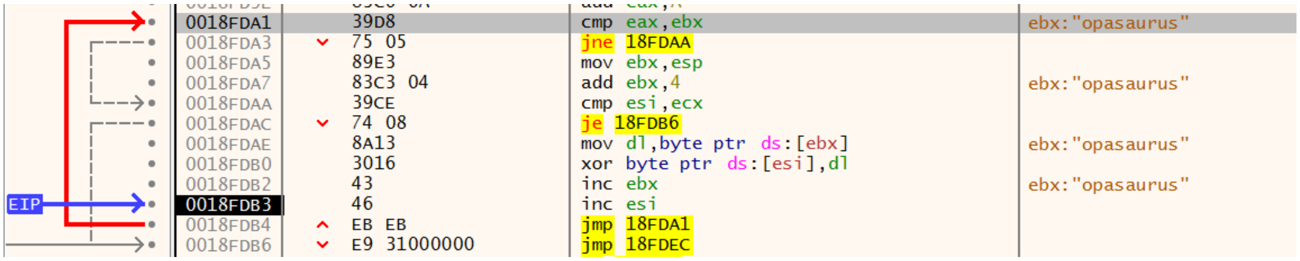


Stepping into the call will reveal several `XOR` decryption routines: the program is basically modifying itself by decrypting new instructions at runtime.

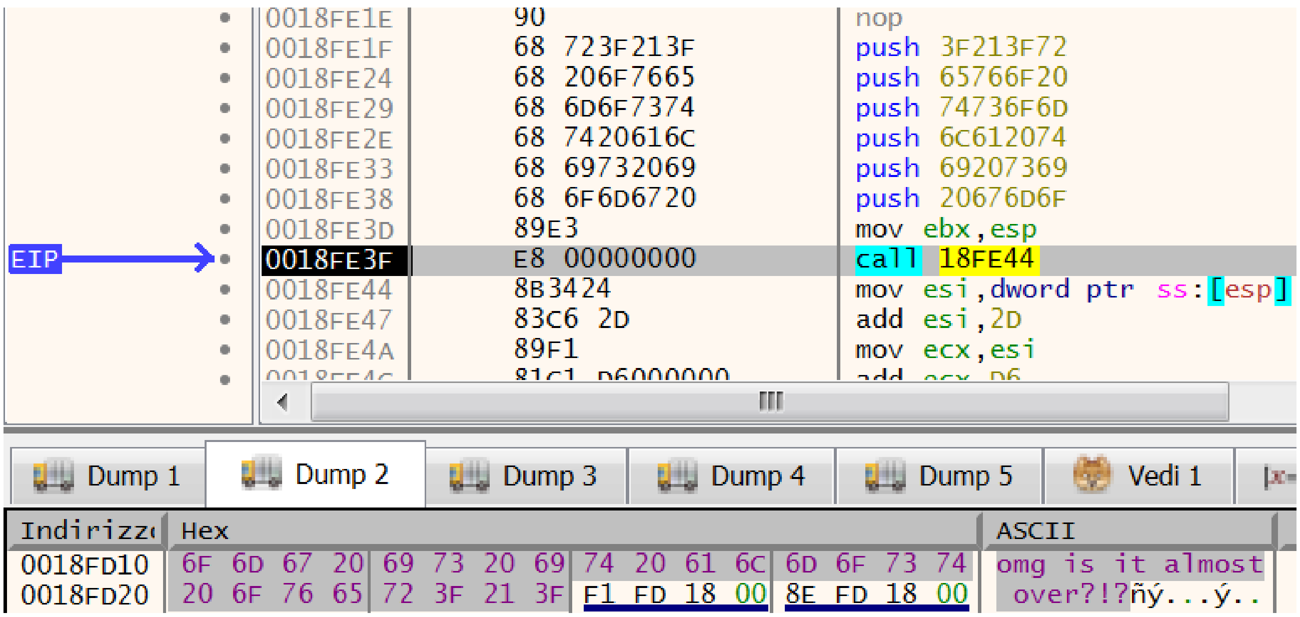
The first block, shown below, uses `0x66` as the decryption key. Instead of just stepping over the instructions, I placed a breakpoint on the first instruction to be executed after the loop (in this case, `jmp 18FD78` at `0018FD63`).



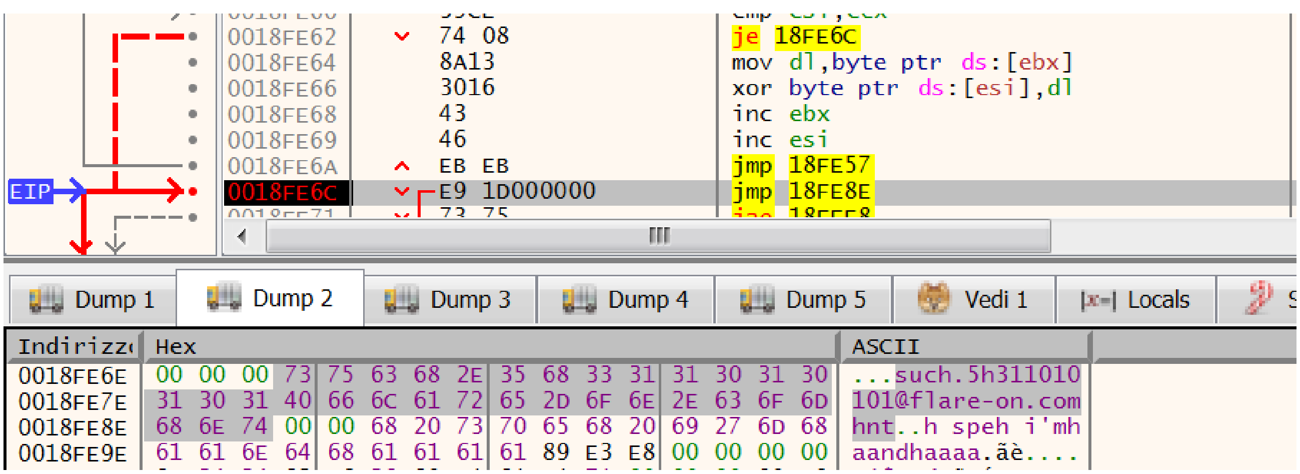
The second XOR block used the string `opasaurus` as the decryption key. Once again, I placed a breakpoint right after the loop and moved onto the next one.



The third XOR block looks similar to the others, just with a different key. Nothing interesting here, so let's just move on! We finally get to the fourth decryption routine. We can see the string *"omg is it almost over?!?"* being pushed onto the stack, which will be used as the decryption key.



We can retrieve the flag by placing, once again, a breakpoint right after the end of the XOR block.



Challenge 4

flag: wa1ch.d3m.spl01ts@flare-on.com
tools: pdfid, pdf-parser, unicode2hex-escaped, shellcode2exe, x32dbg

The sample of this challenge is a pdf document called `APT9001.pdf`, which I'm assuming will exploit old & vulnerable versions of Adobe Acrobat. Unfortunately, I do not have any PDF reader on the VM I'm using for the challenge, so I guess we'll never know!

For a first, quick, analysis of the document, I used `pdfid` e `pdf-parser`, which allowed me to find and extract the JavaScript embedded inside the document.

```
C:\Users\anfl\Desktop\2014_FLAREOn_Challenges\C4>pdfid APT9001.pdf
PDFiD 0.2.5 APT9001.pdf
 PDF Header: %PDF-1.5
 obj                   10
 endobj                 9
 stream                 3
 endstream              3
 xref                   2
 trailer                2
 startxref              2
 /Page                  3(2)
 /Encrypt               0
 /ObjStm                0
 /JS                    1(1)
 /JavaScript            1(1)
 /AA                    0
 /OpenAction            1(1)
 /AcroForm              0
 /JBIG2Decode           1(1)
 /RichMedia             0
 /Launch                0
 /EmbeddedFile          0
 /XFA                   0
 /URI                   0
 /Colors > 2^24         0
```

The extracted JavaScript was heavily obfuscated. I tried my best to deobfuscate it, but eventually realised that most of the code was useless: the exploit was stored as escaped unicode in a variable I renamed `_payload`, as shown below.

```
var _payload = unescape(
"%u72f9%u4649%u1525%u7f0d%u3d3c%ue084%ud62a%ue139%ua84a%u76b9%u9824%u7378%u7d71%u757f%u2076%u96d4%uba91%u1970%ub8f9%ue232%u467b%u9ba8%ufe01%uc7c6%ue3c1%u7e24%u
37c%ue180%ub115%ub3b2%u4f66%u27b6%u9f3c%u7a4e%u412d%ubbbf%u7705%uf528%u9293%u9990%ua998%u0a47%u14eb%u3d49%u484b%u372f%ub98d%u3478%u0bb4%ud5d2%ue031%u3572%ud610
u6740%u2bbe%u4afd%u041c%u3f97%ufc3a%u7479%u421d%ub7b5%u0c2c%u130d%u25f8%u76b0%u4e79%u7bb1%u0c66%u2dbb%u911c%ua92f%ub82c%u8db0%u0d7e%u3b96%u49d4%ud56b%u03b7%ue1
7%u467d%u77b9%u3d42%u111d%u67e0%u4b92%ueb85%u2471%u9b48%uf902%u4f15%u04ba%ue300%u8727%u9fd6%u4770%u187a%u73e2%ufd1b%u2574%u437c%u4190%u97b6%u1499%u783c%u8337%u
3f8%u7235%u693f%u98f5%u7fbe%u4a75%ub493%ub5a8%u21bf%ufcd0%u3440%u057b%ub2b2%u7c71%u814e%u22e1%u04eb%u884a%u2ce2%u492d%u8d42%u75b3%uf523%u727f%ufc0b%u0197%u3d7f
u90f9%u41be%ua81c%u7d25%ub135%u7978%uf80a%ufd32%u769b%u921d%ubbb4%u77b8%u707e%u4073%u0c7a%ud689%u2491%u1446%u9fba%uc087%u0dd4%u4bb0%ub62f%ue381%u0574%u3fb9%u1b
7%u93d5%u8396%u66e0%u47b5%u98b7%u153c%ua934%u3748%u3d27%u4f75%u8cbf%u43e2%ub899%u3873%u7deb%u257a%uf985%ubb8d%u7f91%u9667%ub292%u4879%u4a3c%ud433%u97a9%u377e%u
347%u933d%u0524%u9f3f%ue139%u3571%u23b4%ua8d6%u8814%uf8d1%u4272%u76ba%ufd08%ube41%ub54b%u150d%u4377%u1174%u78e3%ue020%u041c%u40bf%ud510%ub727%u70b1%uf52b%u222f
u4efc%u989b%u901d%ub62c%u4f7c%u342d%u0c66%ub099%u7b49%u787a%u7f7e%u7d73%ub946%ub091%u928d%u90bf%u21b7%ue0f6%u134b%u29f5%u67eb%u2577%ue186%u2a05%u66d6%ua8b9%u15
5%u4296%u3498%ub199%ub4ba%ub52c%uf812%u4f93%u7b76%u3079%ubefd%u3f71%u4e40%u7cb3%u2775%ue209%u4324%u0c70%u182d%u02e3%u4af9%ubb47%u41b6%u729f%u9748%ud480%ud528%u
49b%u1c3c%ufc84%u497d%u7eb8%ud26b%u1de0%u0d76%u3174%u14eb%u3770%u71a9%u723d%ub246%u2f78%u047f%ub6a9%u1c7b%u3a73%u3ce1%u19be%u34f9%ud500%u037a%ue2f8%ub024%ufd4e
u3d79%u7596%u9b15%u7c49%ub42f%u9f4f%u4799%uc13b%ue3d0%u4014%u903f%u41bf%u4397%ub88d%ub548%u0d77%u4ab2%u2d93%u9267%ub198%ufc1a%ud4b9%ub32c%ubaf5%u690c%u91d6%u04
8%u1dbb%u4666%u2505%u35b7%u3742%u4b27%ufc90%ud233%u30b2%uff64%u5a32%u528b%u8b0c%u1452%u728b%u3328%ub1c9%u3318%u33ff%uacc0%u613c%u027c%u202c%ucfc1%u030d%ue2f8%u
1f0%u5bff%u4abc%u8b6a%u105a%u128b%uda75%u538b%u033c%uffd3%u3472%u528b%u0378%u8bd3%u2072%uf303%uc933%uad41%uc303%u3881%u6547%u5074%uf475%u7881%u7204%u636f%u7541
u81eb%u0878%u6464%u6572%ue275%u8b49%u2472%uf303%u8b66%u4e0c%u728b%u031c%u8bf3%u8e14%ud303%u3352%u57ff%u6168%u7972%u6841%u694c%u7262%u4c68%u616f%u5464%uff53%u68
2%u3233%u0101%u8966%u247c%u6802%u7375%u7265%uff54%u68d0%u786f%u0141%udf8b%u5c88%u0324%u6168%u6567%u6842%u654d%u7373%u5054%u54ff%u2c24%u6857%u2144%u2121%u4f68%u
e57%u8b45%ue8dc%u0000%u0000%u148b%u8124%u0b72%ua316%u32fb%u7968%ubece%u8132%u1772%u45ae%u48cf%uc168%ue12b%u812b%u2372%u3610%ud29f%u7168%ufa44%u81ff%u2f72%ua9f7
u0ca9%u8468%ucfe9%u8160%u3b72%u93be%u43a9%ud268%u98a3%u8137%u4772%u8a82%u3b62%uef68%u11a4%u814b%u5372%u47d6%uccc0%ube68%ua469%u81ff%u5f72%ucaa3%u3154%ud468%u65
b%u8b52%u57cc%u5153%u8b57%u89f1%u83f7%u1ec7%ufe39%u0b7d%u3681%u4542%u4645%uc683%ueb04%ufff1%u68d0%u7365%u0173%udf8b%u5c88%u0324%u5068%u6f72%u6863%u7845%u7469%u
f54%u2474%uff40%u2454%u5740%ud0ff");
```

I tried following the methodology explained here to extract the JavaScript shellcode and embed it into an executable. It was a complete failure and I was left with a corrupted executable!

I then found out about REMnux, an Ubuntu-based distro for reverse engineering, which had two perfect tools for this purpose: `unicode2hex-escaped` and `shellcode2exe`.

```
remnux@remnux:~/Desktop$ cat exploit.js | unicode2hex-escaped > shellcode
remnux@remnux:~/Desktop$ shellcode2exe -s shellcode
shellcode2exe: command not found
remnux@remnux:~/Desktop$ shellcode2exe.py -s shellcode
Shellcode to executable converter
by Mario Vilas (mvilas at gmail dot com)

Reading string shellcode from file shellcode
Generating executable file
Writing file shellcode.exe
Done.
remnux@remnux:~/Desktop$ _
```

When executed, the resulting executable will open a message box with the title "OWNED!!!" and an unreadable string, which I assumed was the encrypted flag. This was confirmed by debugging the executable in x32dbg: the cleartext flag was pushed onto the stack, encrypted, and later used inside the message box object.

```
00401361 | 8172 0B 16A3FB32 | xor dword ptr ds:[edx+B],32FBA316
00401368 | 68 6F6D4500      | push 456D6F
0040136D | 8172 17 AE45CF48 | xor dword ptr ds:[edx+17],48CF45AE
00401374 | 68 6F6E2E63      | push 632E6E6F
00401379 | 8172 23 10369FD2 | xor dword ptr ds:[edx+23],D29F3610
00401380 | 68 6172652D      | push 2D657261
00401385 | 8172 2F F7A9A90C | xor dword ptr ds:[edx+2F],CA9A9F7
0040138C | 68 7340666C      | push 6C664073
00401391 | 8172 3B BE93A943 | xor dword ptr ds:[edx+3B],43A993BE
00401398 | 68 6C303174      | push 7431306C
0040139D | 8172 47 828A623B | xor dword ptr ds:[edx+47],3B628A82
004013A4 | 68 6D2E7370      | push 70732E6D
004013A9 | 8172 53 D647C0CC | xor dword ptr ds:[edx+53],CCC047D6
004013B0 | 68 682E6433      | push 33642E68
004013B5 | 8172 5F A3CA5431 | xor dword ptr ds:[edx+5F],3154CAA3
004013BC | 68 77613163      | push 63316177
004013C1 | 8BCC             | mov ecx,esp
004013C3 | 57               | push edi
004013C4 | 53               | push ebx

ecx=0018FF30 "wa1ch.d3m.spl01ts@flare-on.comE"
esp=0018FF30 "wa1ch.d3m.spl01ts@flare-on.comE"
```

Challenge 5

flag:l0gging.ur.5troke5@flare-on.com
tools: IDA

The sample for this challenge was a DLL called `5get_it`. After opening it with IDA, I realized that the sample was basically a *keylogger*: the core of the execution was a loop using **GetAsyncKeyState** (followed by lots of if/then statements) to determine which keys were pressed and which actions to perform as a result.

```
74D29EDD 0F BF 45 FC movsx   eax, [ebp+var_4]
74D29EE1 50          push    eax             ; vKey
74D29EE2 FF 15 3C 41+call    ds:GetAsyncKeyState
74D29EE8 0F BF C8    movsx   ecx, ax
74D29EEB 81 F9 01 80+cmp     ecx, 0FFFF8001h
74D29EF1 0F 85 A8 04+jnz     loc_74D2A39F    ; jumptable 1000A2D4 default case
```

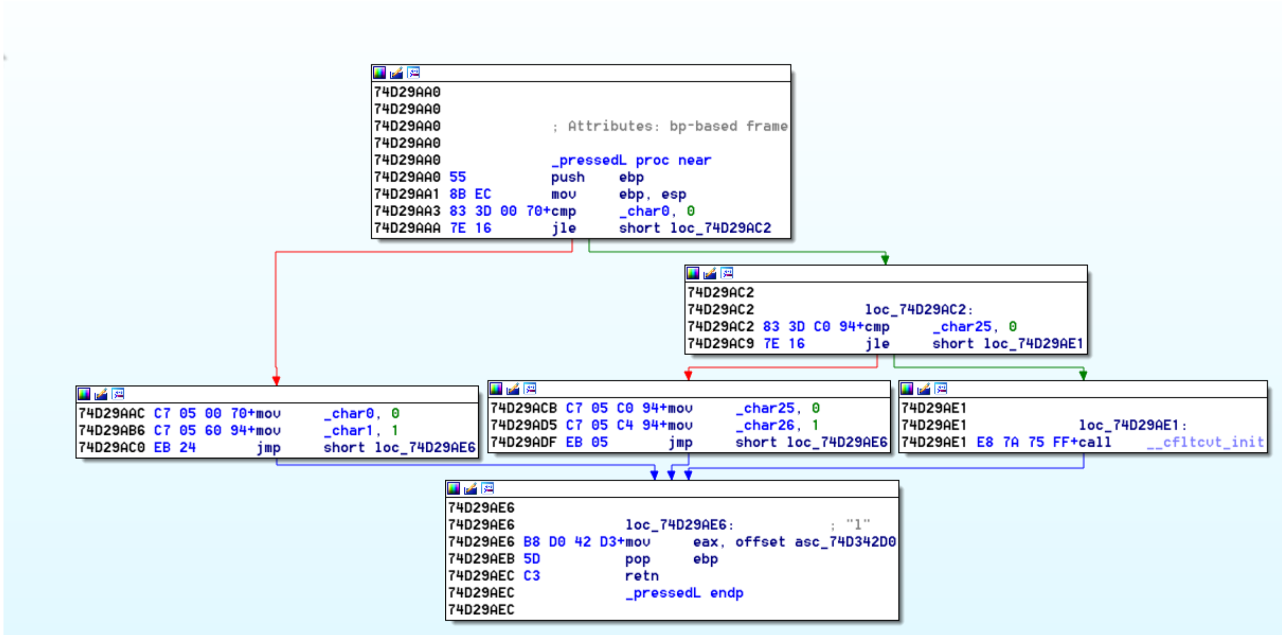
From my understanding, non alpha-numeric characters were simply ignored, while letters and numbers were appended at the end of a file called `svchost.log`, created in the same directory as the executable (this is something quite common with malware, as svchost is the name of a legitimate windows process and, as such, it is less suspicious).

```
74E51011 6A 0A       push    0Ah             ; dwMilliseconds
74E51013 FF 15 18 40+call    ds:Sleep
74E51019 68 90 41 E6+push    offset aA_1     ; "a+"
74E5101E 68 94 41 E6+push    offset aSvchost_log ; "svchost.log"
74E51023 E8 0E 9A 00+call    _fopen
74E51028 83 C4 08    add     esp, 8
74E5102B 89 45 FC    mov     [ebp+var_4], eax
74E5102E EB DB       jmp     short loc_74E5100B
```

I also found a function (called `__cfltcvt_init`) that was called several times throughout the code, which initializes a bunch of variables, with either 1 or 0.

```
71B71060
71B71060
71B71060                 ; Attributes: library function bp-based frame
71B71060
71B71060                 __cfltcvt_init proc near
71B71060 55              push    ebp
71B71061 8B EC           mov     ebp, esp
71B71063 C7 05 00 70+mov     dword_71B87000, 1
71B7106D C7 05 60 94+mov     dword_71B89460, 0
71B71077 C7 05 64 94+mov     dword_71B89464, 0
71B71081 C7 05 68 94+mov     dword_71B89468, 0
71B7108B C7 05 6C 94+mov     dword_71B8946C, 0
71B71095 C7 05 70 94+mov     dword_71B89470, 0
71B7109F C7 05 74 94+mov     dword_71B89474, 0
71B710A9 C7 05 78 94+mov     dword_71B89478, 0
```

These variables are used to check whether the user is typing the right keys for the flag following the expected order. The picture below shows an example of this logic: this block of code will be executed whenever the "L" key is pressed, which corresponds to the first character of the flag.



The function will check the value of the first variable (which I renamed `_char0` for clarity). If it equals 1, then it means we are at the beginning of the flag sequence, therefore `_char0` will be set at 0 and the next variable (`_char1`) will be set at 1.

If, however, `_char0` equals 0, one of two things may have happened: either the key was pressed for the second time (therefore `_char25` will be checked to determine if the previous characters were correctly typed) or the sequence is not being respected (therefore the variables will be initialized again by calling `__cfltcvt_init`).

To solve this I went back and forth in IDA, cross-referencing the variables (using `CTRL+X`) to find which functions where using them and then cross-referencing the function to find which key press would trigger it.

The resulting flag was *l0gging.ur.5troke5@flare-on.com* (see below).

```
74D21060                    __cfltcvt_init proc near
74D21060 55                 push    ebp
74D21061 8B EC              mov     ebp, esp
74D21063 C7 05 00 70+mov    _char0, 1       ; l
74D2106D C7 05 60 94+mov    _char1, 0       ; 0
74D21077 C7 05 64 94+mov    _char2, 0       ; g
74D21081 C7 05 68 94+mov    _char3, 0       ; g
74D2108B C7 05 6C 94+mov    _char4, 0       ; i
74D21095 C7 05 70 94+mov    _char5, 0       ; n
74D2109F C7 05 74 94+mov    _char6, 0       ; g
74D210A9 C7 05 78 94+mov    _char7, 0       ; d
74D210B3 C7 05 7C 94+mov    _char8, 0       ; o
74D210BD C7 05 80 94+mov    _char9, 0       ; t
74D210C7 C7 05 84 94+mov    _char10, 0      ; u
74D210D1 C7 05 88 94+mov    _char11, 0      ; r
74D210DB C7 05 8C 94+mov    _char12, 0      ; d
74D210E5 C7 05 90 94+mov    _char13, 0      ; o
74D210EF C7 05 94 94+mov    _char14, 0      ; t
74D210F9 C7 05 98 94+mov    _char15, 0      ; 5
74D21103 C7 05 9C 94+mov    _char16, 0      ; t
74D2110D C7 05 A0 94+mov    _char17, 0      ; r
74D21117 C7 05 A4 94+mov    _char18, 0      ; o
74D21121 C7 05 A8 94+mov    _char19, 0      ; k
74D2112B C7 05 AC 94+mov    _char20, 0      ; e
74D21135 C7 05 B0 94+mov    _char21, 0      ; 5
74D2113F C7 05 B4 94+mov    _char22, 0      ; a
74D21149 C7 05 B8 94+mov    _char23, 0      ; t
74D21153 C7 05 BC 94+mov    _char24, 0      ; f
74D2115D C7 05 C0 94+mov    _char25, 0      ; l
74D21167 C7 05 C4 94+mov    _char26, 0      ; a
74D21171 C7 05 C8 94+mov    _char27, 0      ; r
74D2117B C7 05 CC 94+mov    _char28, 0      ; e
74D21185 C7 05 D0 94+mov    _char29, 0      ; d
74D2118F C7 05 D4 94+mov    _char30, 0      ; a
74D21199 C7 05 D8 94+mov    _char31, 0      ; s
74D211A3 C7 05 DC 94+mov    _char32, 0      ; h
74D211AD C7 05 E0 94+mov    _char33, 0      ; o
74D211B7 C7 05 E4 94+mov    _char34, 0      ; n
74D211C1 C7 05 E8 94+mov    _char35, 0      ; d
74D211CB C7 05 EC 94+mov    _char36, 0      ; o
74D211D5 C7 05 F0 94+mov    _char37, 0      ; t
74D211DF C7 05 F4 94+mov    _char38, 0      ; c
74D211E9 C7 05 F8 94+mov    _char39, 0      ; o
74D211F3 C7 05 FC 94+mov    _char40, 0      ; m
```