Flare-On 5: Challenge Solution – leet_editr.exe

Challenge Author: Michael Bailey (@mykill)

More than one player noticed some similarities between this and the challenge I wrote last year. Aside from some reused ASCII art, `leet_editr.exe` bears substantive similarities to `zsud.exe`: it is a native executable that furtively loads a scripting runtime (read on to see which) and decrypts a script that calls functions to bind itself to its loader (read on to see how) such that it can't be run in a normal scripting environment; it also brings up a user interface that appears to be unrelated to the program's runtime. But the devil is in the details.

Executing `leet_editr.exe` produces a message box warning that you are about to run the coolest ASCII Art editor on earth, as shown in Figure 1.

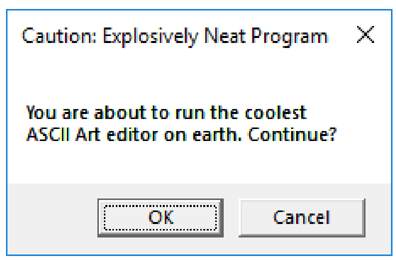

Figure 1: Corny warning message

Clicking OK led to disappointment on systems that were running things like IDA Pro or x64dbg (or under, ahem, "other" circumstances[1]). True to samples found in the wild, running reverse engineering (RE) tools can tip off malware or induce it to alter its behavior. You can get around this by executing `leet_editr.exe` without any RE tools running, or you can be stubborn about it and find one that I missed[2].

Once you satisfy `leet_editr.exe`, it spawns a copy of Internet Explorer displaying a sinister-looking ASCII art of Bob Dobbs' face, as visible in Figure 2.

[1] See https://twitter.com/alex_k_polyakov/status/1042844336902299648 and https://twitter.com/stuxxn/status/1042498255026716672
[2] See https://twitter.com/justadrawer2/status/1041040829366726656

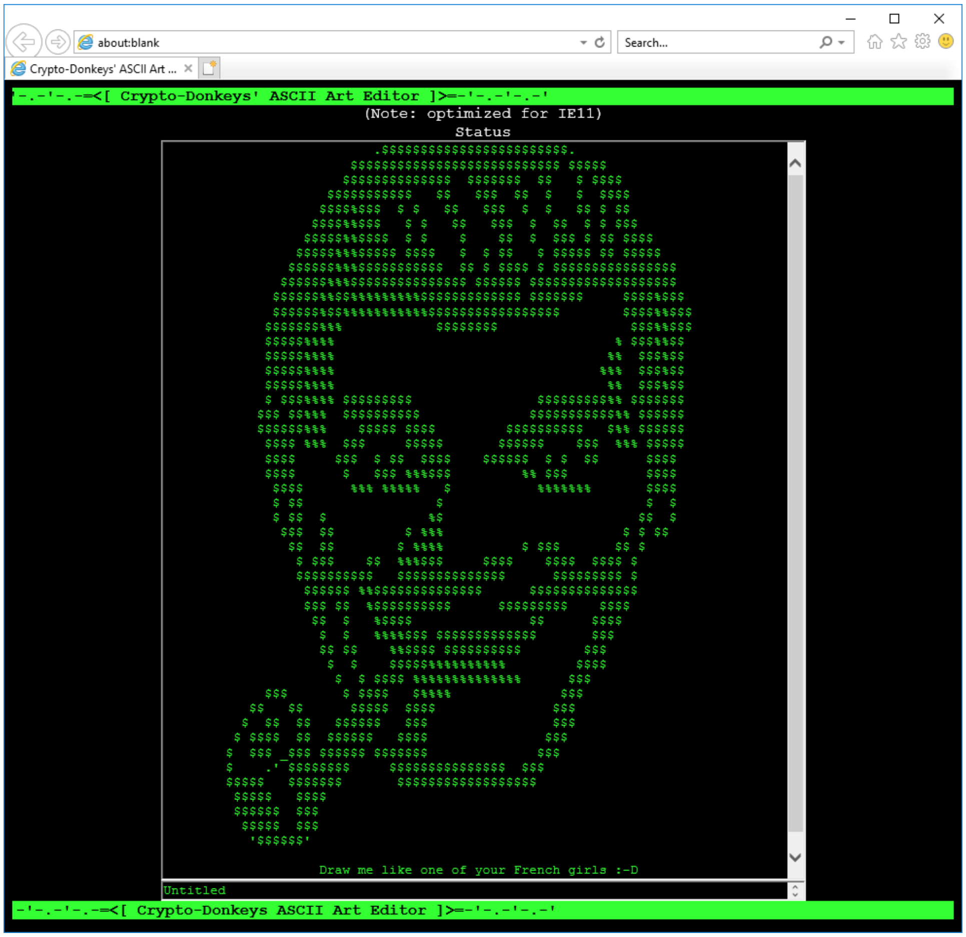

Figure 2: `Leet_editr.exe` *after dismissing the message box*

Viewing the source code in IE reveals JavaScript that implements string hash algorithms and sets the `status div` element to some hint text. But the script code seems incomplete because nothing calls these. Figure 3 display some of the script code.

```
<script language='JavaScript'>
function hint(s) {
    document.getElementById('status').innerText = 'You\'re on to something!' + s
}

function strhash2(s) {
    // Adapted from:
    // https://stackoverflow.com/questions/7616461/generate-a-hash-from-string-in-javascript-jquery
    var hash = 0, i, c;
    for (i=0; i<s.length; i++) {
        c = s.charCodeAt(i);
        hash = ((hash << 5) - hash) + c;
        hash |= 0;
    }
    return hash;
}

function strhash(s) {
    // Adapted from:
```

Figure 3: JavaScript in source view of Internet Explorer

Basic Static Analysis

There are only a few plain strings of interest in this binary, which are shown in Listing 1.

```
%d 0x%x
You are about to run the coolest
ASCII Art editor on earth. Continue?
Caution: Explosively Neat Program
Running shellcode crouching_vbs_hidden_title.asm...
HRESULT 0x%08x
%02x
createtextfile
gimmethatsweetsweetcrazylove
run
getspecialfolder
wimmymebrah
```

Listing 1: Obligatory strings listing for `Leet_editr.exe`

The string "`Running shellcode crouching_vbs_hidden_title.asm...`" suggests a few things:

- VBScript
- A hidden title (what might this mean?)
- Shellcode compiled from assembly language code

The strings `CreateTextFile` and `GetSpecialFolder` do correspond to a COM object that is indeed commonly used in VBScript, namely `Scripting.FileSystemObject` which is commonly referred to by script authors as `FSO`. The remaining strings are either marginally interesting or simply nonsensical.

The next stop for basic static analysis is imports. Here are some of the interesting ones along with a running commentary of some reasonable inferences they should probably trigger:

- `CoInitialize`: COM usage? But where's `CoCreateInstance`?
- `CryptAcquireContextA`, `CryptCreateHash`, etc.: Cryptographic hashing?
- `OutputDebugStringA`: You're going to be one of those sassy challenge authors then, eh?
- `AddVectoredExceptionHandler`: Giving the game away a little bit, aren't we?
- `VirtualAlloc`, `VirtualProtect`: Probably for that shellcode.
- `FlushInstructionCache`: Self-modifying? Or decoding shellcode? We'll see.
- `SetErrorMode`: Worried about suppressing Windows Error Reporting dialogs?

The PE headers in this file don't lend a lot of insight, so it's time to move on to the next analysis stage.

Basic Dynamic Analysis

Just because `leet_editr.exe` doesn't open when your favorite RE tool is open doesn't make it impossible to use basic dynamic analysis techniques. Assuming the mechanism used here is a blacklist that enumerates process names, you can exploit a weakness by simply renaming your RE tools. Doing so with Process Explorer yields the ability to conveniently review in-memory strings in search of any decoded VBScript or other strings, as shown in Figure 4.

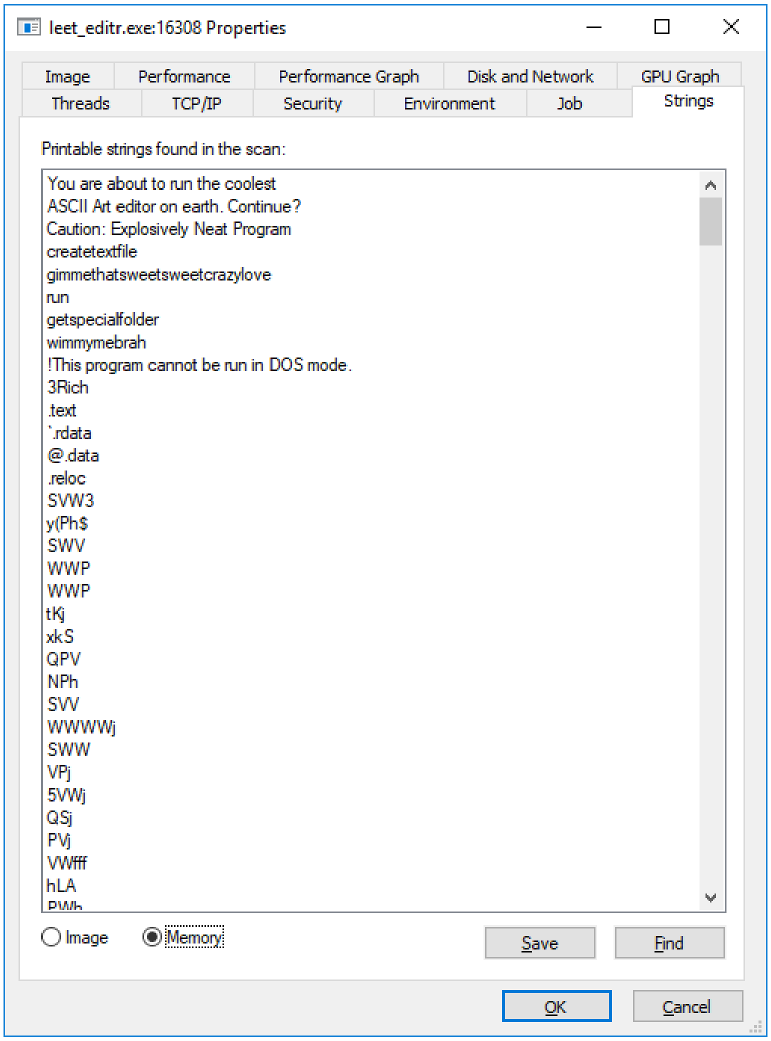

Figure 4: Process Explorer in-memory strings view

Figure 5 depicts a vimdiff comparison of a sorted listing of in-memory strings against those from the image. This yields no discoveries, suggesting that strings are re-encoded after they are used.

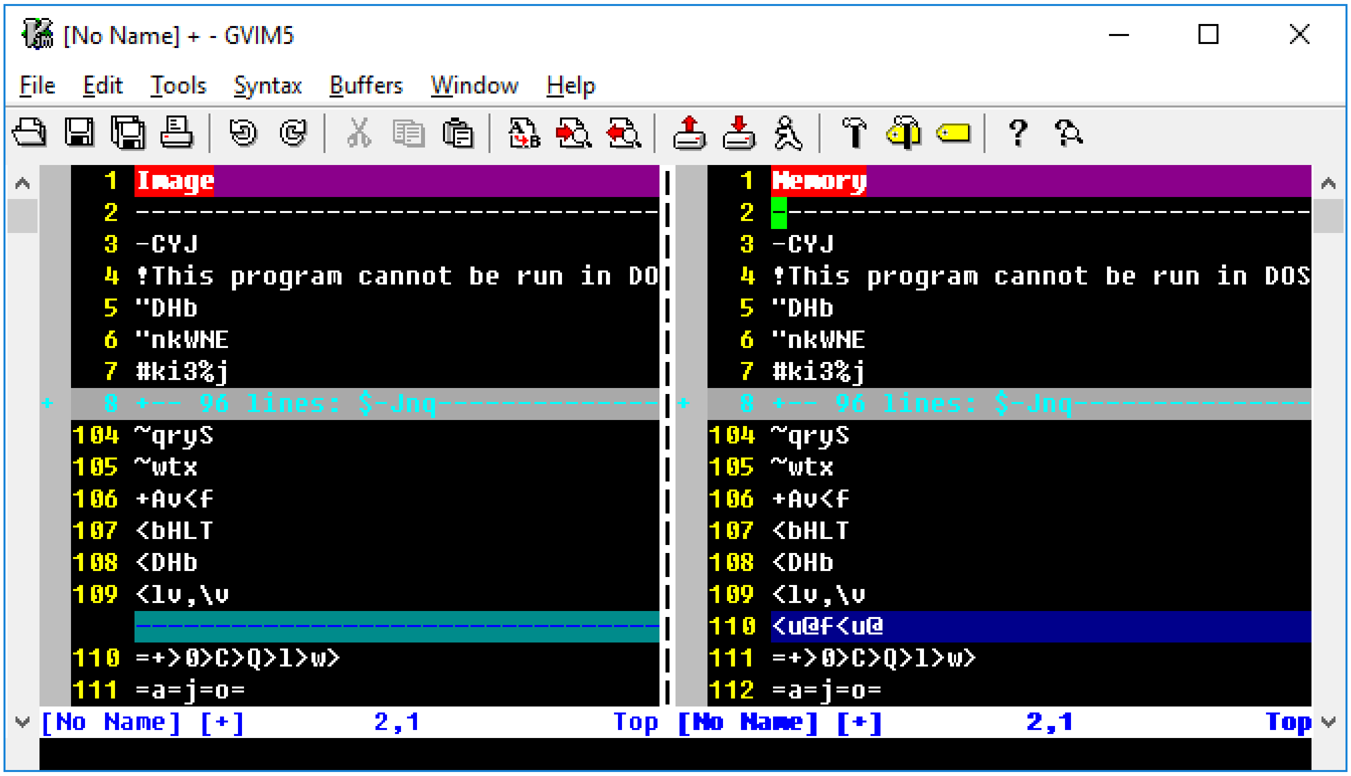

Figure 5: Comparison of in-memory strings against image

Advanced Static Analysis

The `WinMain` function for this is relatively short. It first copies byte strings into heap buffers, setting PAGE_NOACCESS. It then calls a function that copies further function addresses into a table. It finally installs a vectored exception handler and calls into a heap buffer.

The location accessed at `0x401045` is loaded into the `esi` register and then manipulated such that a structure can be discerned, as shown in Figure 6.

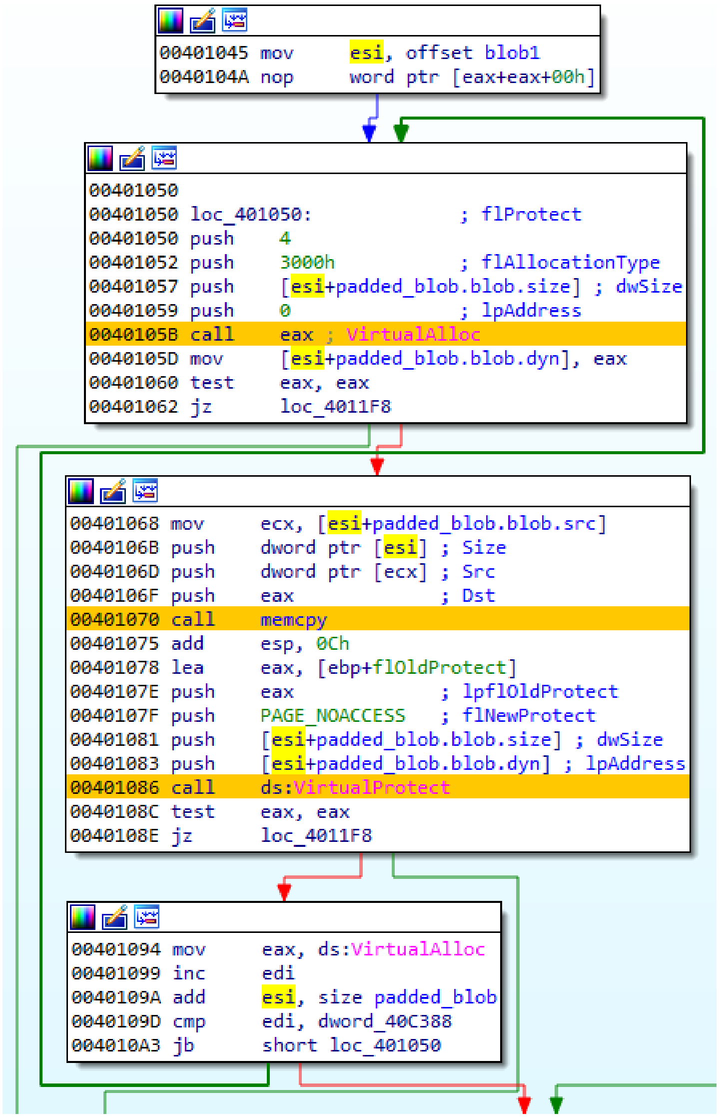

Figure 6: Memory-oriented structure accessed in loop

The compiler's optimizations obscure the real structure slightly, but analyzing the `VirtualAlloc`, memcpy, and `VirtualProtect` calls in `WinMain` yields a sufficient structure having the fields shown in the comment column of Figure 7.

```
0040C2B8 blob1           dd 6Eh                  ; blob.size
0040C2B8                                         ; DATA XREF: sub_401000+45↑o
0040C2B8                 dd offset off_40B000    ; blob.src
0040C2B8                 dd 30h                  ; blob.field_8
0040C2B8                 dd offset dword_40B010  ; blob.field_C
0040C2B8                 dd 1                    ; blob.field_10
0040C2B8                 dd offset dword_40B190  ; blob.ptr_to_len_and_buf
0040C2B8                 dd 0                    ; blob.dyn
0040C2B8                 dd 1                    ; field_1C
0040C2B8                 dd 0                    ; field_20
```

Figure 7: Unknown structure

Analyzing the structure as it is used in the loop reveals that the target of the indirect call toward the end of `WinMain` is the heap allocation saved in the sixth such structure, as shown in Figure 8.

```
004011B2 mov     esi, blob6_main.dyn ; size
004011B8 mov     edi, ds:OutputDebugStringA
004011BE push    offset OutputString ; "Running shellcode crouching_vbs_hidden_"...
004011C3 call    edi ; OutputDebugStringA
004011C5 lea     eax, [ebp+var_130]
004011CB push    eax
004011CC call    esi              ; blob6 heap buffer
004011CE add     esp, 4
004011D1 test    eax, eax
004011D3 jns     short loc_4011FD
```

Figure 8: Call into sixth structure's heap buffer

Although the data in that location has permissions set to PAGE_NOACCESS after the `VirtualProtect` call, it is possible to find the source of the `memcpy` that populates the heap buffer. The data at `0x404718` must be the shellcode that is later executed at the indirect call, however attempting to decode it as x86 instruction code leads to the dubious results shown in Figure 9.

```
00404718 loc_404718:                             ; DATA XREF: .data:src_entry↓o
00404718                 stosd
00404719                 jnz     short near ptr loc_404729+4
0040471B                 jge     short near ptr loc_40472E+1
0040471D                 fisubr  word ptr [eax-6909748Bh]
0040471D                                         ; CODE XREF: .rdata:00404705↑j
00404723                 db      26h
00404723                 out     dx, al
00404725                 db      2Eh
00404725                 push    esp
00404727                 test    al, 75h         ; CODE XREF: .rdata:004046E8↑j
```

Figure 9: Nonsense instructions at 0x404718

The shellcode mystery will have to be resolved later.

Meanwhile, `WinMain` also calls the function shown in Figure 10 which copies function addresses into a table.

```
00401990
00401990
00401990 ; Attributes: bp-based frame
00401990
00401990 sub_401990 proc near
00401990
00401990 arg_0= dword ptr  8
00401990
00401990 push    ebp
00401991 mov     ebp, esp
00401993 mov     eax, [ebp+arg_0]
00401996 mov     fptr_array, offset fptr_table
004019A0 mov     fptr_table, offset sub_401FA0
004019AA mov     fptr_table+4, offset sub_401970
004019B4 mov     fptr_table+8, offset sub_402030
004019BE mov     fptr_table+0Ch, offset sub_401B10
004019C8 mov     fptr_table+10h, offset sub_401AB0
004019D2 mov     fptr_table+14h, offset sub_4019F0
004019DC mov     fptr_table+18h, offset sub_401B30
004019E6 mov     dword ptr [eax], offset fptr_array
004019EC xor     eax, eax
004019EE pop     ebp
004019EF retn
004019EF sub_401990 endp
004019EF
```

Figure 10: Function pointer table

The functions referenced here all return values that correspond to well-known HRESULT values such as `E_INVALIDARG`, `E_NOINTERFACE`, `DISP_E_UNKNOWNNAME`, and `DISP_E_UNKNOWNINTERFACE`. The penultimate of these functions returns magic numbers in exchange for strings, and the last function executes specific logic corresponding to each number. If you haven't programmed or seen this before, it may be unrecognizable, but this is an implementation of the COM interface known as `IDispatch`[3], which allows for late binding. Setting the type of the structure at `0x40FD8C` to `IDispatchVtbl` from

[3] https://msdn.microsoft.com/en-us/library/ms526185.aspx

IDA's type libraries will cause IDA to name each function pointer with the corresponding name from `IDispatch`, as shown in Figure 11. I've manually renamed the functions themselves (on the right-hand side) to match their associated `IDispatchVtbl` struct field members.

```
00401990
00401990
00401990 ; Attributes: bp-based frame
00401990
00401990 setup_IDispatch proc near
00401990
00401990 ppv= dword ptr  8
00401990
00401990 push    ebp
00401991 mov     ebp, esp
00401993 mov     eax, [ebp+ppv]
00401996 mov     pv, offset vtbl
004019A0 mov     vtbl.QueryInterface, offset QueryInterface
004019AA mov     vtbl.AddRef, offset AddRef
004019B4 mov     vtbl.Release, offset Release
004019BE mov     vtbl.GetTypeInfoCount, offset GetTypeInfoCount
004019C8 mov     vtbl.GetTypeInfo, offset GetTypeInfo
004019D2 mov     vtbl.GetIDsOfNames, offset GetIDsOfNames
004019DC mov     vtbl.Invoke, offset Invoke
004019E6 mov     dword ptr [eax], offset pv
004019EC xor     eax, eax
004019EE pop     ebp
004019EF retn
004019EF setup_IDispatch endp
004019EF
```

Figure 11: IDispatch virtual function table setup

The `GetIDsOfNames` and `Invoke` methods define the method names and corresponding numeric IDs that can be used by a COM client to invoke methods of the type that is implemented by this `IDispatch` implementation. Table 1 lists the four method names, magic numbers, and their semantics based on what can be seen from reading the body of each function.

Name	Magic Number	Semantic
`createtextfile`	`0xCAFEBABE`	RC4 decrypt and return a BSTR (OLE automation string type) associated

		with the string wimmymebrah
gimmethatsweetsweetcrazylove	0x1337	Hash something and use its MD5 sum as the key to RC4 decrypt something else
run	0xDEADBEEF	Disable WER dialogs using SetErrorMode and crash by calling a NULL function pointer
getspecialfolder	0x1010101	Call the Sleep function

Table 1: Method names, IDs, and semantics based on GetIDsOfNames and Invoke

The vectored exception handler looms as the next major item pending analysis. It handles two main cases: access violations, and single-step exceptions. In the access violation case, the handler obtains a range structure that defines a base address and a length which it uses to decide where to change memory protections. It changes protections to read/write before calling a decoder at 0x4012A0 that consults a bitmask in the range structure to decide between the algorithms shown in Table 2.

Bitmask value	Encoding
1	XOR
2	Incrementing XOR
4	RC4
0x80	Combination of three encodings (In source code, I callled it Neapolitan)

Table 2: Encoding algorithm enumeration

After decoding is finished, the handler changes the permissions of the memory range to read/execute and calls FlushInstructionCache to ensure that the instruction cache is cleared of any invalid instructions. It then sets bit 8 (0x100) in EFLAGS within the context record for the faulting thread. The Intel 64 and IA-32 Architectures Software Development Manual shows this to be the Trap Flag (TF) bit of the EFLAGS register, as shown in Figure 12.

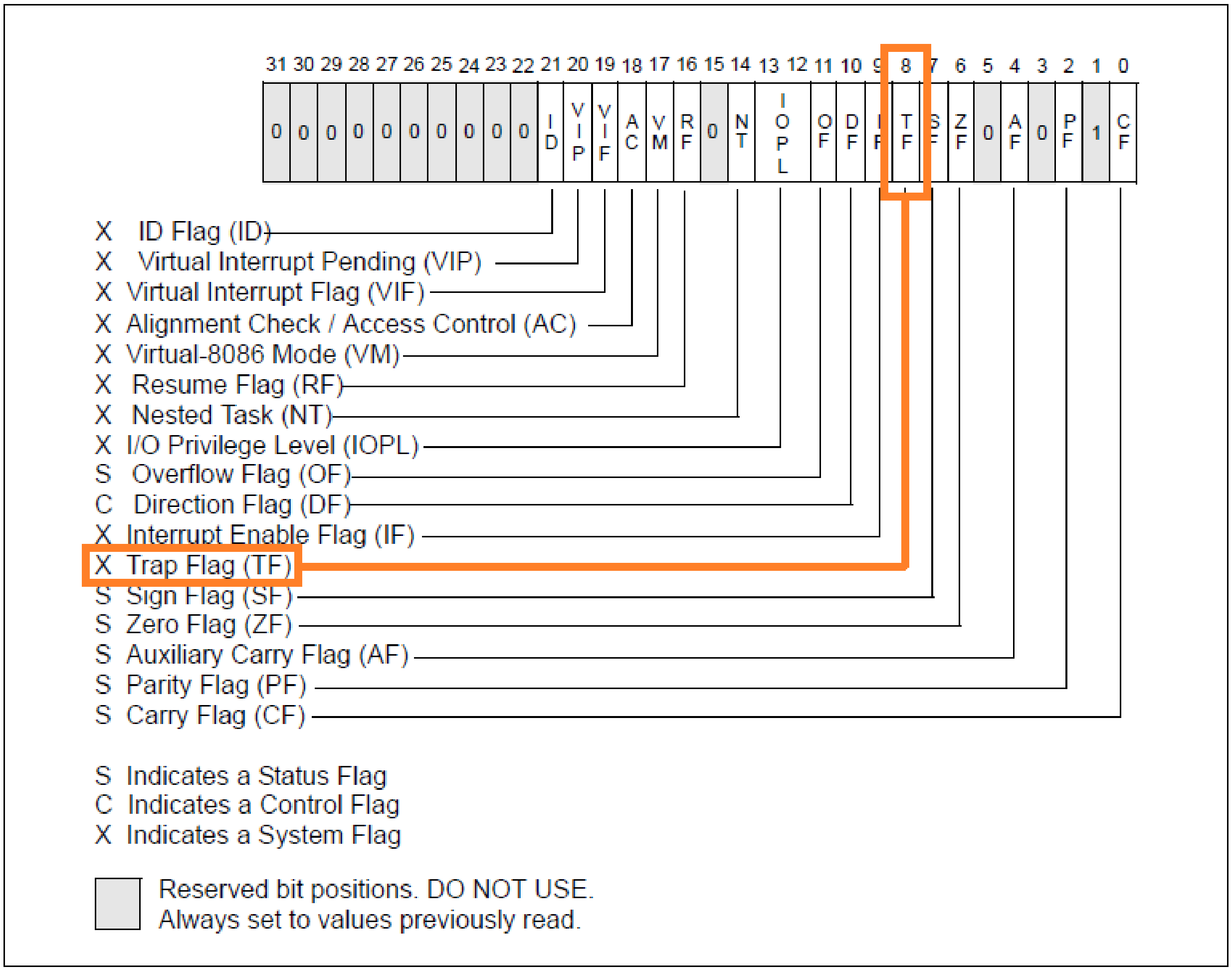

Figure 12: Bit 8 of EFLAGS is the Trap Flag (denoted TF)

The handler lastly returns `EXCEPTION_CONTINUE_EXECUTION` to permit the decoded shellcode to execute.

The single-step handling logic of the vectored exception handler likewise calls the encoder, thus re-encoding whatever data was decoded in the access violation case.

In summary, `leet_editr.exe` copies encoded shellcode from static buffers into heap locations with no page access to induce an access violation upon execution. It installs a vectored exception handler to catch these exceptions and then decode the data, set the trap flag, and re-encode the data after the instruction has executed. At this point, it is reasonable to consider dynamic analysis.

Advanced Hybrid Analysis: Dynamic, Static, and Bochs Debugging

In WinDbg, you can disable access violations, single-step exceptions, and other items that would produce unwanted console output. Special thanks to Tyler Dean of the FLARE team (Twitter: @spresec) for identifying the WinDbg syntax for preventing single-step exceptions (`sxi ssec`) and sharing one other solution rudiment that I have shamelessly borrowed (read on for more). Listing 2 shows a WinDbg session that ignores exceptions (1), executes up to the shellcode call instruction (2), sets a memory breakpoint on execution of the shellcode (3), and disassembles the first instruction after it has been decoded (4).

```
0:000> sxi av                $$ (1)
0:000> sxi sse
0:000> sxi ssec
0:000> sxi ld
0:000> bp leet_editr+0x11cc  $$ (2)
0:000> g
Running shellcode crouching_vbs_hidden_title.asm...
Breakpoint 0 hit
eax=0053fdb0 ebx=00000000 ecx=73e44063 edx=00000000 esi=00840000 edi=75df5c20
eip=011711cc esp=0053fd98 ebp=0053fee0 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
leet_editr+0x11cc:
011711cc ffd6            call    esi {00840000}
0:000> ba e1 @esi            $$ (3)
0:000> g
Breakpoint 1 hit
eax=0053fdb0 ebx=00000000 ecx=73e44063 edx=00000000 esi=00840000 edi=75df5c20
eip=00840000 esp=0053fd94 ebp=0053fee0 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
00840000 ??              ???
0:000> u @eip
00840000 ??              ???
            ^ Memory access error in 'u @eip'
0:000> p
eax=0053fdb0 ebx=00000000 ecx=73e44063 edx=00000000 esi=00840000 edi=75df5c20
eip=00840001 esp=0053fd90 ebp=0053fee0 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
00840001 7512            jne     00840015                                [br=0]
0:000> u @eip-1              $$ (4)
00840000 55              push    ebp
```

```
00840001 7512            jne     00840015
00840003 7d12            jge     00840017
00840005 dea8758bf696    fisubr  word ptr [eax-6909748Bh]
0084000b 26ee            out     dx,al
0084000d 2e54            push    esp
0084000f a875            test    al,75h
00840011 b8ee012e7d      mov     eax,7D2E01EEh
```
Listing 2: WinDbg output

Indeed `push ebp` is a coherent instruction to expect to see at the beginning of a function. One of the encoding algorithms listed in Table 2 was XOR. Since the original value of the first shellcode byte was `0xAB`, we can calculate a potential XOR key and see if using it produces coherent code throughout the shellcode. The XOR result of `0xAB ^ 0x55` is `0xFE`. The IDAPython one-liner in Listing 3 can be used to apply this to the shellcode region.

```
for n in range(0x70): PatchByte(here() + n, Byte(here() + n) ^ 0xfe)
```
Listing 3: XORing shellcode with 0xFE

The result is shellcode that calls several function pointers and references numeric constants that are reminiscent of string hashes (for details, see https://www.fireeye.com/blog/threat-research/2012/11/precalculated-string-hashes-reverse-engineering-shellcode.html).

Doing this with the other shellcode regions brings us to an intricate, 965-byte swath of shellcode at `0x4041D8` which IDA fails to turn into a procedure. Careful analysis (or use of the strings utility on the decoded shellcode) reveals the ASCII string `"If I were to title this piece, it would be 'A_FLARE_f0r_th3_Dr4m4t1(C)'\r\n"` as shown in Figure 13.

```
004043D8 83 C4 14                          add     esp, 14h
004043DB 81 FE 48 69 6E 74                 cmp     esi, 'tniH'
004043E1 75 51                             jnz     short loc_404434
004043E1                   ; ---------------------------------------------------------------------------
004043E3 49 66 20 49 20 77 65 72 65+aIfIWereToTitle db 'If I were to title this piece, it would be ',27h,'A_FLARE_f0r_th3'
004043E3 20 74 6F 20 74 69 74 6C 65+                db '_Dr4m4t1(C)',27h,0Dh,0Ah
0040442C                   ; ---------------------------------------------------------------------------
0040442C 8B 5D 08                          mov     ebx, [ebp+8]
0040442F 8B 7D F8                          mov     edi, [ebp-8]
00404432 EB 08                             jmp     short loc_40443C
```

Figure 13: ASCII text interspersed between instructions

This string interrupts IDAs disassembly and analysis of the shellcode function, so it is useful to take note of the string and then nop it out with a `PatchByte` one-liner similar to Listing 3.

There are a number of stack strings and other elements in the shellcode. A quick way to evaluate these is to use IDA's Bochs debugger integration in IDB mode and advance EIP over the function calls to get the stack strings written into debug memory. This yields the strings and structures in Figure 14.

```
IDA View-EIP
    STACK:00420F44 aVbscript:
    STACK:00420F44 text "UTF-16LE", 'VBScript',0
    STACK:00420F56 db    0
    STACK:00420F57 db    0
    STACK:00420F58 dd 0E59F1D5h                          ; Data1
    STACK:00420F58 dw 1FBEh                              ; Data2
    STACK:00420F58 dw 11D0h                              ; Data3
    STACK:00420F58 db 8Fh, 0F2h, 0, 0A0h, 0D1h, 0, 38h, 0BCh; Data4
    STACK:00420F68 dd 0E59F1D3h                          ; Data1
    STACK:00420F68 dw 1FBEh                              ; Data2
    STACK:00420F68 dw 11D0h                              ; Data3
    STACK:00420F68 db 8Fh, 0F2h, 0, 0A0h, 0D1h, 0, 38h, 0BCh; Data4
EAX STACK:00420F78 aCocreateinstan db 'CoCreateInstance',0
    STACK:00420F89 dd 0
    STACK:00420F8D dw 0
    STACK:00420F8F db    0
    STACK:00420F90 aOleaut32Dll_0 db 'oleaut32.dll',0
    STACK:00420F9D db    0
    STACK:00420F9E db    0
    STACK:00420F9F db    0
    STACK:00420FA0 aPoo:
    STACK:00420FA0 text "UTF-16LE", 'poo',0
    STACK:00420FA8 db  0Dh
    STACK:00420FA9 db  24h ; $
    STACK:00420FAA db  36h ; 6
UNKNOWN 00420F56: STACK:00420F56 (Synchronized with EIP)
```

Figure 14: Stack strings and GUIDs found using Bochs in IDB mode

Along with the shellcode, these stack artifacts tell a story. The stack string `CoCreateInstance` is created within `var_3C` to resolve the COM function that instantiates COM class instances; the shellcode resolves this function by name and stores the result in `var_C`. When the shellcode calls this function, it pushes the IID and a CLSID that are also constructed in stack memory. We also see the strings `VBScript` and poo (I'm regretting that choice now that I must write it up).

To understand the COM function pointers being used throughout the shellcode, it is ideal to identify the interface ID (IID) or class ID (CLSID) and associated function pointer table. I want to share some tactics I use in practice for resolving questions that arise from reversing COM client code. The payoff for this is that we can use it to understand something about how the VBScript code is going to interface with the native binary.

COM Rabbit Hole

The goal here is to get a structure that defines the virtual function table offsets being used in the COM client code. I usually get good results grepping for standard IIDs and CLSIDs in the Windows headers directory and by searching in the registry. The Windows headers are silent here but searching the registry for `E59F1D3` yields `IScriptControl` as shown in Figure 15.

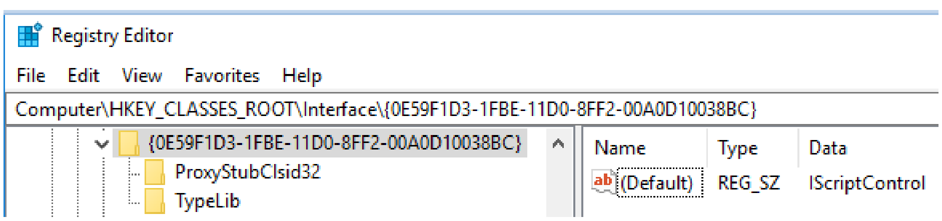

Figure 15: IScriptControl IID

This registry finding is the beginning of a trail of breadcrumbs that will allow us to make the shellcode more coherent. Underneath the IID is a `TypeLib` key that points to a Universally Unique Identifier (UUID) as shown in Figure 16.

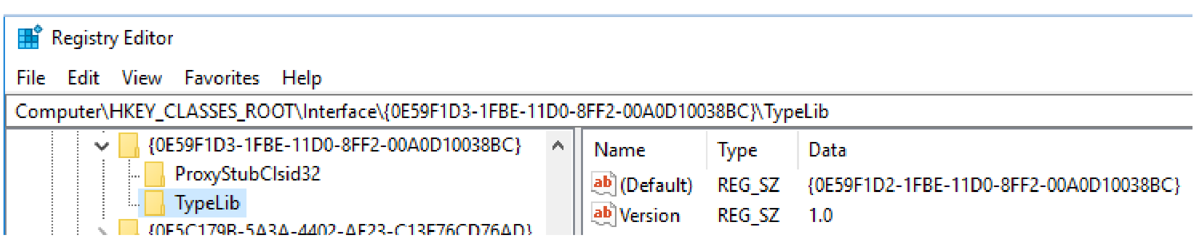

Figure 16: TypeLib GUID associated with IScriptControl

A type library (normally a `.tlb` file) can be used to derive an interface definition (a `.idl` file) for the COM interface and then derive header files that can be modified and imported into IDA Pro to become structures for use in enriching the disassembly. Searching the registry for the UUID associated with this type library yields the path to `msscript.ocx` shown in Figure 17.

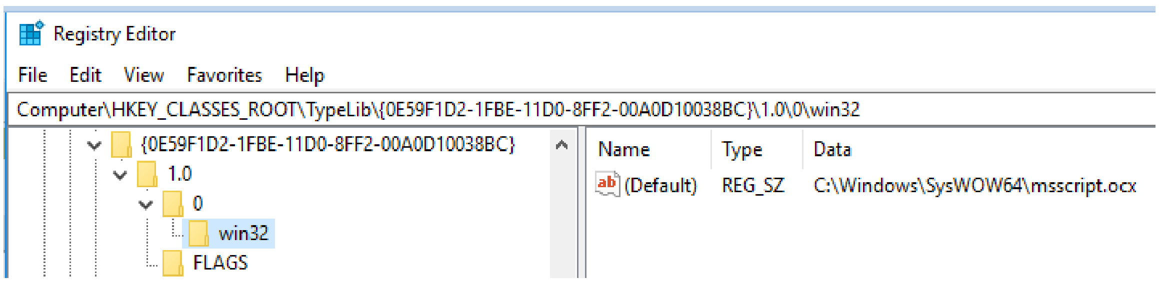

Figure 17: Path to Microsoft Script Control type library

Granted, the `.ocx` file path here isn't a `.tlb`, but as it turns out, it contains exactly what we need. Opening `msscript.ocx` in OleView displays the generated IDL file shown in Figure 18. This is a language-neutral definition of the interfaces supported by `msscript.ocx`.

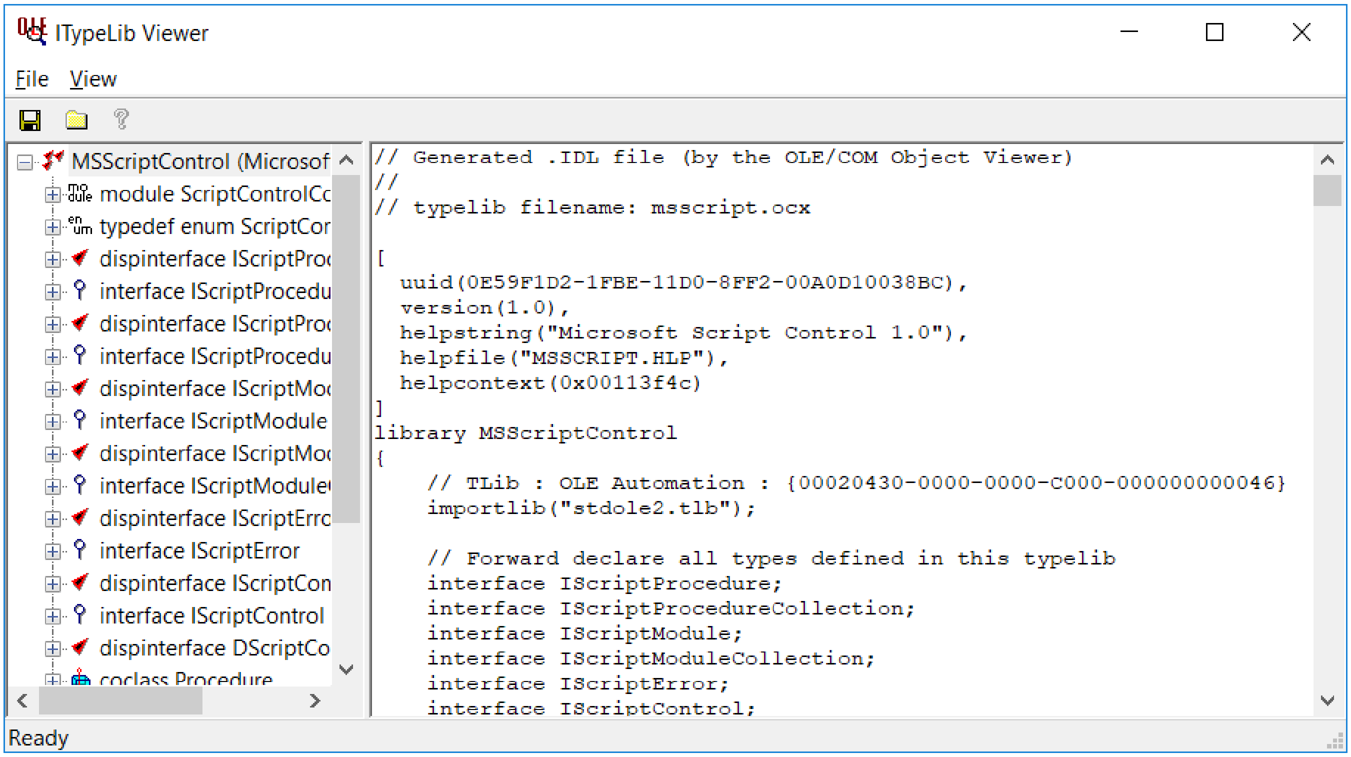

Figure 18: Using `OleView` on `msscript.ocx`

Saving this IDL file to disk with File -> Save As… and choosing a name such as `msscript.idl` allows us to use Microsoft's MIDL compiler (`midl.exe`) from a Visual Studio Tools prompt. There is one hitch, however: Microsoft's MIDL compiler complains about the syntax of the IDL generated from Microsoft's own type library! Figure 19 shows the MIDL compiler's complaints (`MIDL2400` and `MIDL2401`).

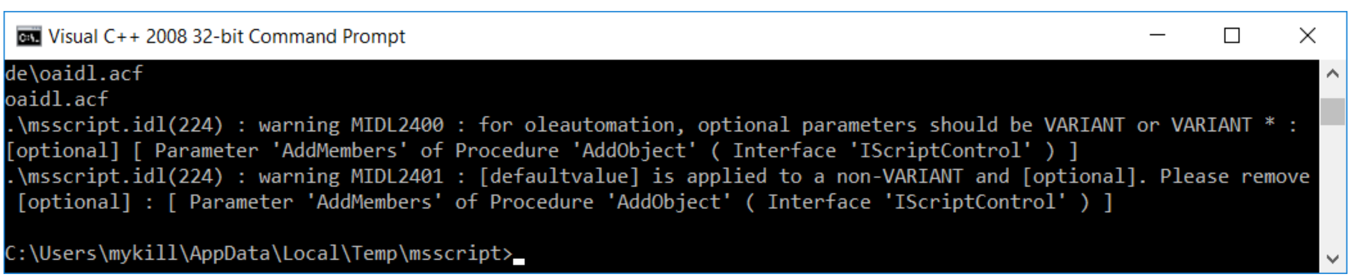

Figure 19: MIDL compiler errors 2400 and 2401

Figure 20 shows how to disable these warnings 2400 and 2401 with `midl_pragma` statements.

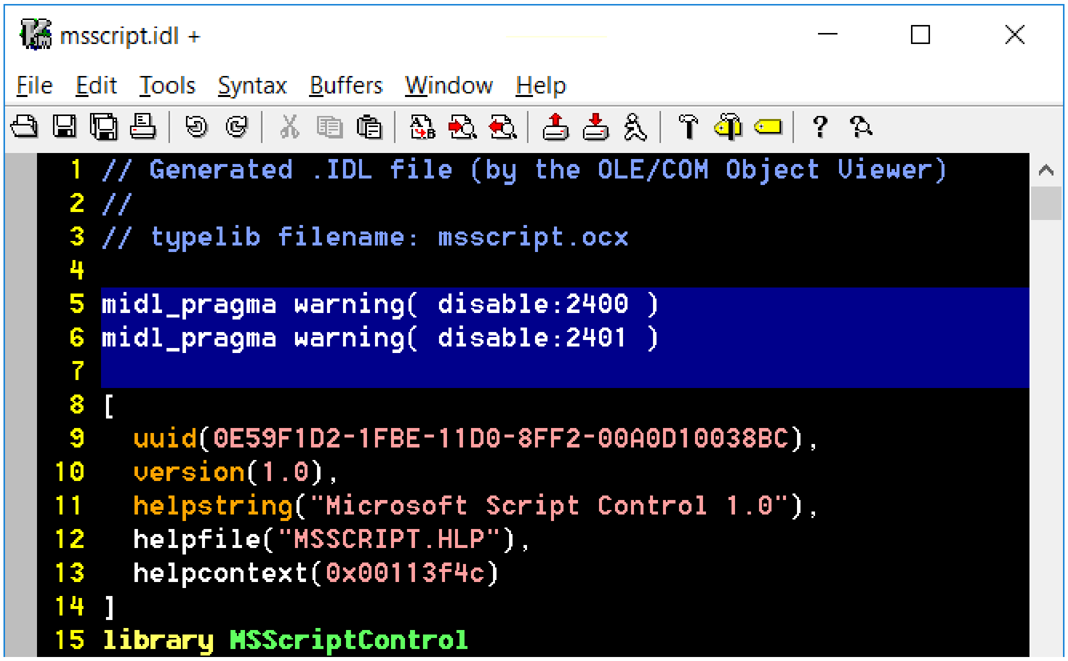

Figure 20: Disabling MIDL warnings 2400 and 2401

The generated header file `msscript.h` is hostile to IDA's type importation system due to the numerous COM-related definitions, so it is expedient to gut the definition of `IScriptControlVtbl` and import the simplified version shown in Figure 21.

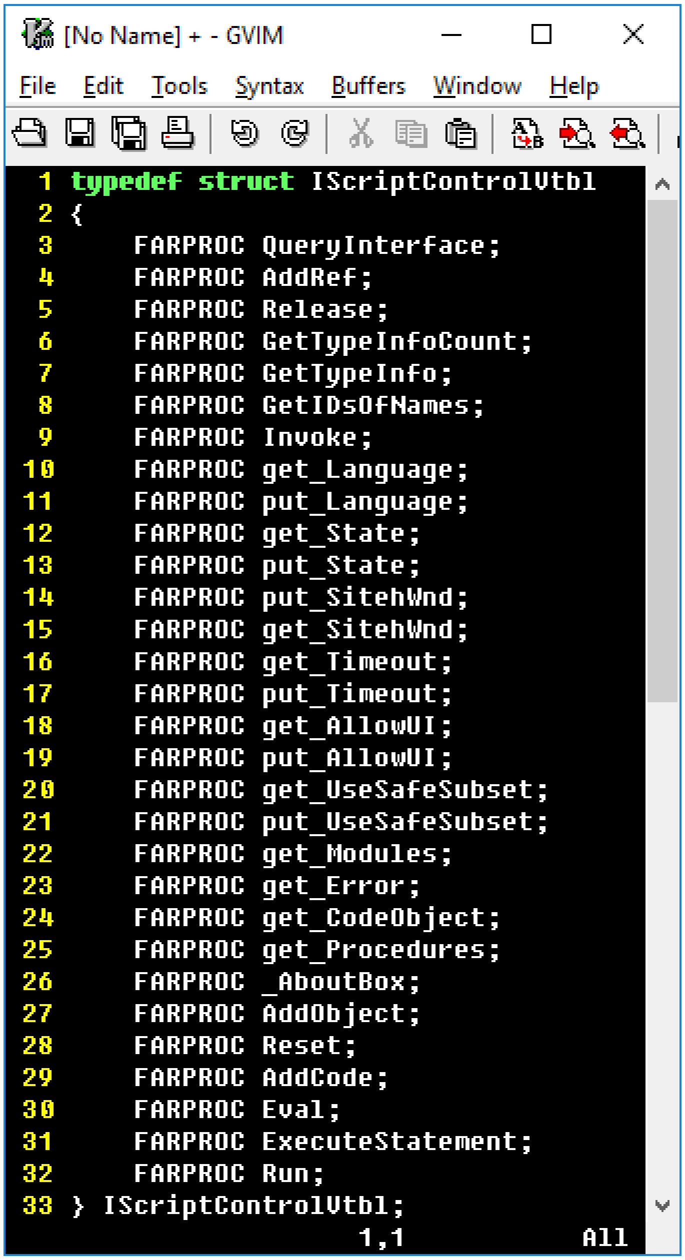

Figure 21: Simplified `IScriptControlVtbl` definition derived from `msscript.h`

With the `IScriptControlVtbl` type added to IDA's types, it is now possible to conveniently add it as a structure of the same name and then use its structure offsets to make sense of the COM function calls. For instance, Figure 22 shows the call to `IScriptControlVtbl.put_Language` which sets the

Language property of the script control instance.

```
004044D1
004044D1 loc_4044D1:
004044D1                 mov     eax, [ebp+ppv]
004044D4                 push    [ebp+str_VBScript]
004044D7                 push    eax
004044D8                 mov     ecx, [eax]
004044DA                 mov     eax, [ecx+IScriptControlVtbl.put_Language]
004044DD                 call    eax ; IScriptControl->put_Language(this, BSTR)
004044DF                 mov     esi, eax
004044E1                 test    esi, esi
004044E3                 js      short loc_404562
```

Figure 22: Using IScriptControlVtbl struct offsets to mark up the COM call at 0x4044DD

Aside from `put_Language`, the shellcode also calls `AddObject` three times and `ExecuteStatement` once before releasing the COM instance. Now we can debug the binary again using `sxe ld msscript` and use the WinDbg `x` (Examine Symbols) command to search `msscript` and find addresses to set breakpoints and observe details.

```
1:001> x msscript!*put_Language
6ab631e0          msscript!CScriptControl::put_Language (<no parameter info>)
1:001> x msscript!*AddObject
6ab63910          msscript!CScriptControl::AddObject (<no parameter info>)
1:001> x msscript!*ExecuteStatement
6ab68250          msscript!CModuleObject::ExecuteStatement (<no parameter info>)
6ab63110          msscript!CScriptControl::ExecuteStatement (<no parameter info>)
6ab64899          msscript!CScriptControl::ModuleExecuteStatement (<no parameter info>)
```

Figure 23: Examining `msscript` symbols in search of COM function addresses

In summary, the shellcode adds objects poo, oSh, and `fso` as aliases for a single interface pointer. This pointer corresponds to the `IDispatch` interface that was set up at `0x40FD8C`. What this does is allow the VBScript to use the aforementioned object names to invoke any of the methods in Table 1.

The shellcode next calls `ExecuteStatement` passing an encoded buffer that has PAGE_NOACCESS set. The encoded, protected buffer is burdensome to reverse statically, and VBScript builds an abstract syntax tree (AST) instead of maintaining the decoded string in memory, so a dynamic solution is preferable.

The decoder at `0x4012A0` ends at `0x4013C6`, so the solution employed by Tyler Dean of the FLARE team (Twitter: @spresec) was to break there and dump the bytes to be stitched together later by a Python function. Listing 4 shows a sequence of WinDbg commands that can be used to dump a list of addresses and decoded bytes

```
sxi av
sxi ld
sxi sse
sxi ssec
ba e1 leet_editr+0x13C6 "db poi(esp+8) L2; g"
g
```

Listing 4: WinDbg sequence to dump decoded Unicode bytes of VBScript

Although this takes a long time to run under the debugger, it does finally produce the decoded bytes, with encoded versions interspersed between. Eliminating the extraneous WinDbg commands and output, and deleting every other line, makes a text file we can parse with the Python in Figure 24.

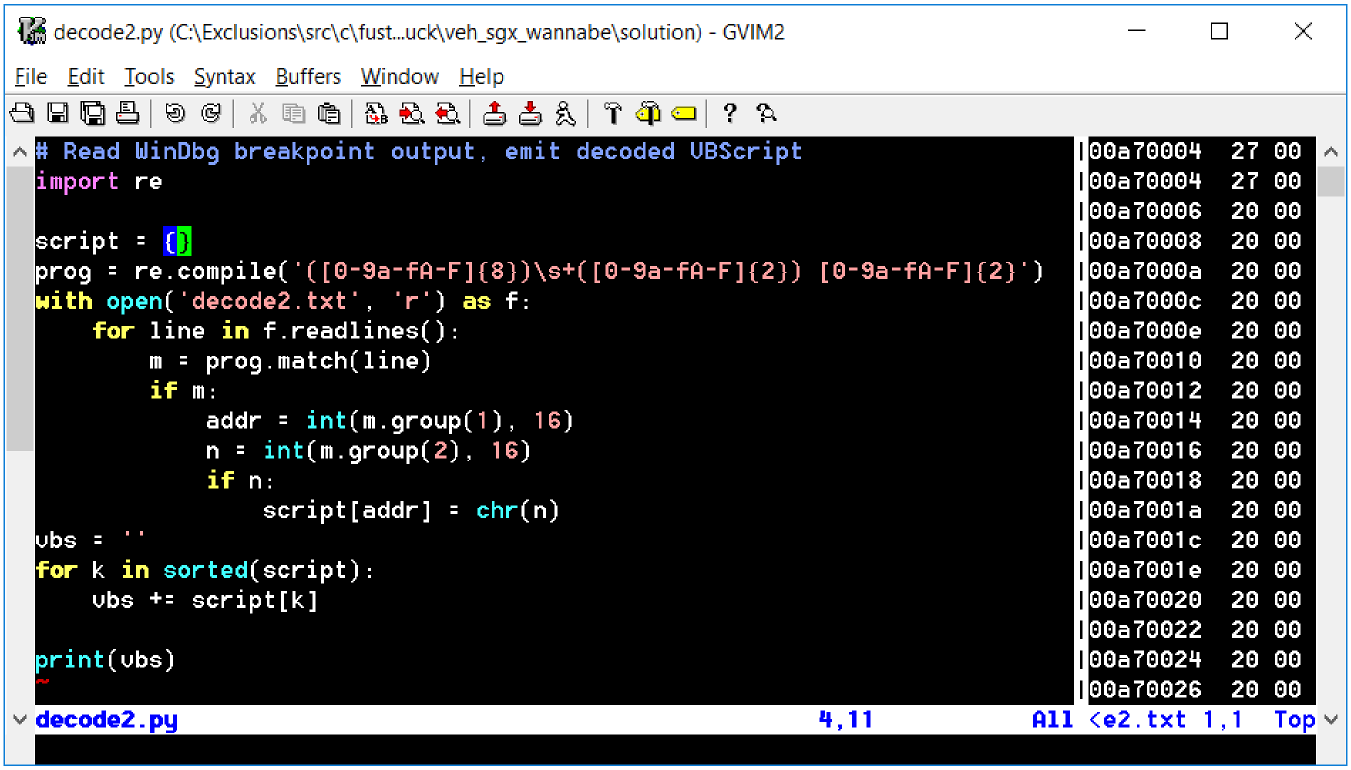

Figure 24: Python script to convert WinDbg breakpoint hex dump into VBScript

The result of this is the decoded script which bears the ASCII art preamble in Figure 25.

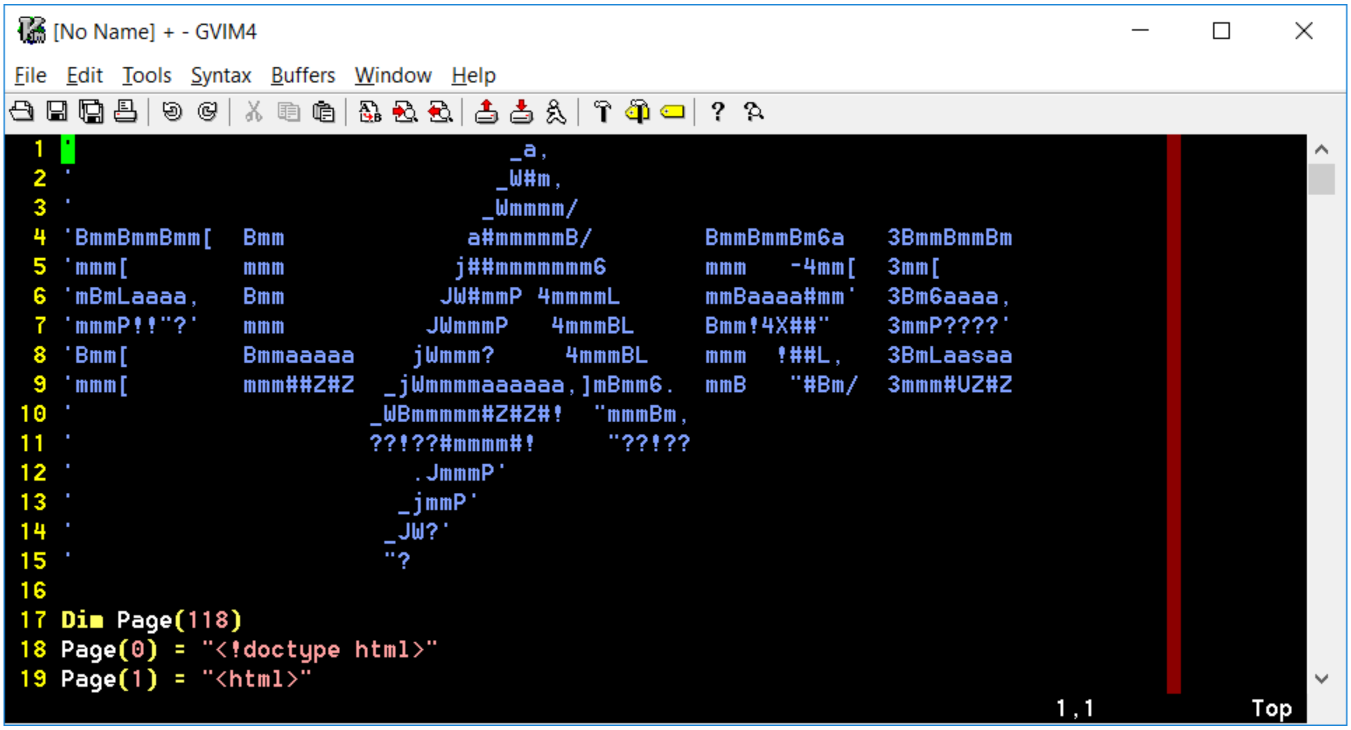

Figure 25: ASCII art preamble to VBScript

The VBScript uses the `InternetExplorer.Application` COM object to inject the hard-coded HTML in the Page variable into a blank browser page and then inject a script separately to poll the contents of the `textarea` element named `textin` until it contains a substring of the ASCII art at the top of the script and matches a particular string hash value. A second similar check validates that the user has entered a certain title for their ASCII art. The script calls the `createtextfile`, `run`, `getspecialfolder`, and `gimmethatsweetsweetcrazylove` COM methods through the objects provided via `IScriptControl->AddObject`. This binds the VBScript to the native program that loaded it and prevents it from being executed outside of that environment without modification. The function `gimmethatsweetsweetcrazylove` is what decrypts the flag and injects it into the browser.

All that is necessary to satisfy the first (`textarea`) check and proceed to the next is to paste the ASCII art (comment characters and all) into the `textarea` element. Figure 26 shows how the hint box is updated, prompting the user to enter a title for the ASCII art.

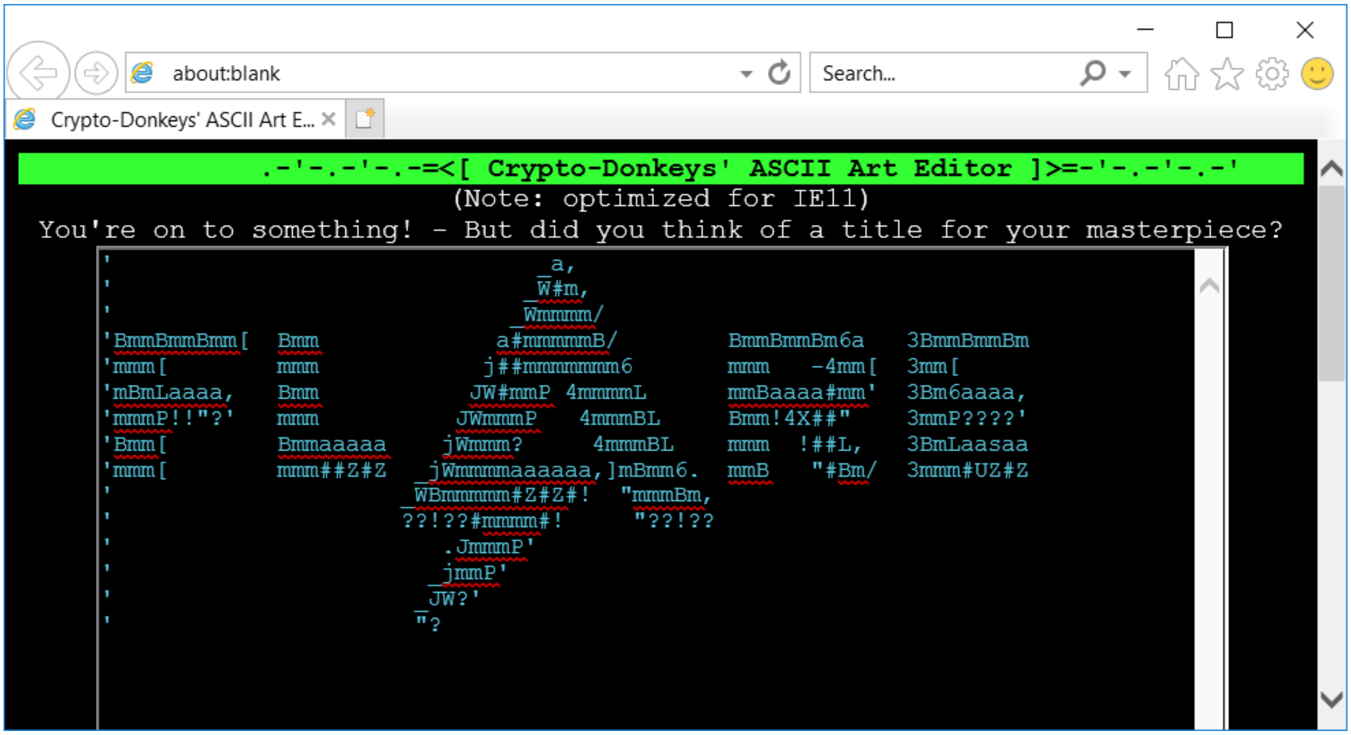

Figure 26: After pasting ASCII art into ASCII art editor

Recall that the shellcode contained the string `"If I were to title this piece, it would be 'A_FLARE_f0r_th3_Dr4m4t1(C)'"`. Figure 27 shows the title pasted into the title element.

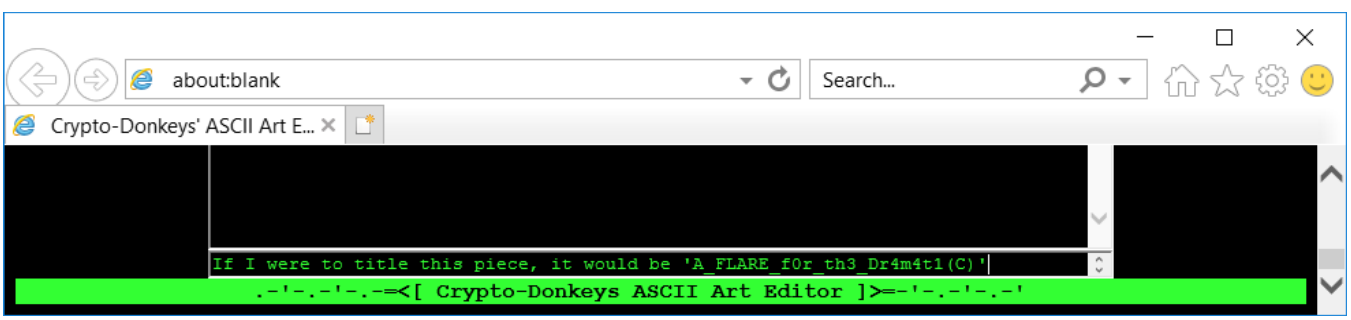

Figure 27: Pasting the title into the web page

When this is done, the VBScript injects the HTML shown in Figure 28, which displays a marquee version of the flag.

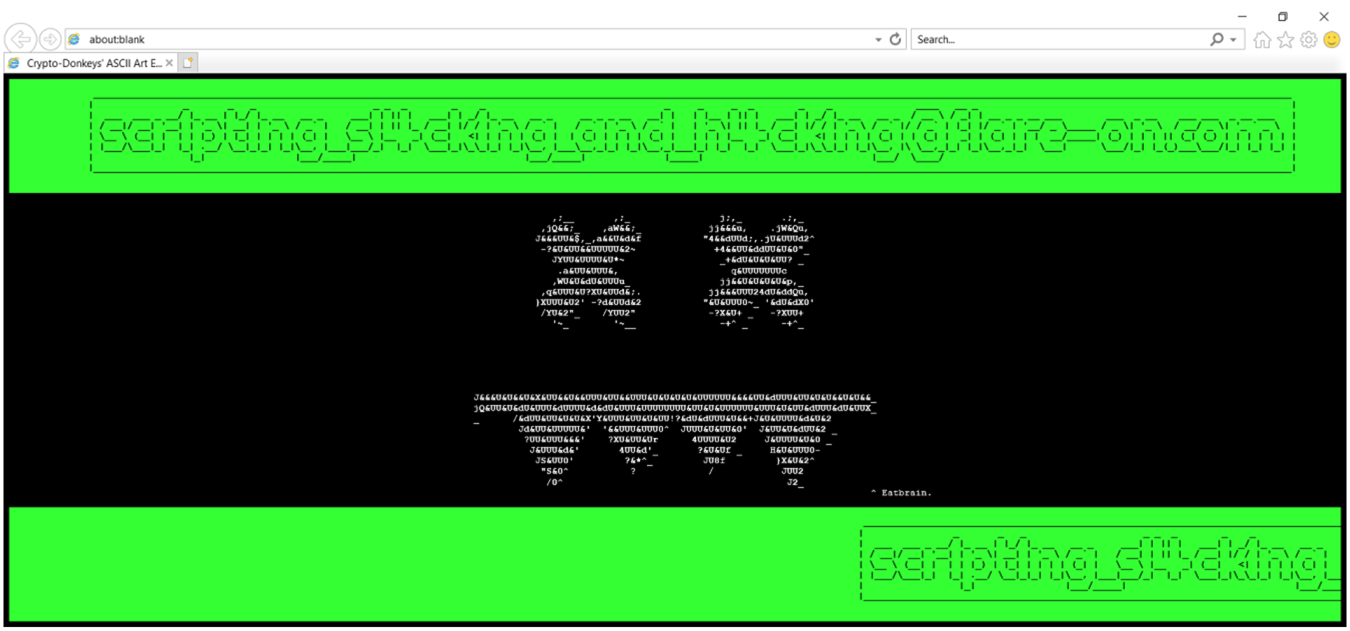

Figure 28: Sweet, sweet flag

The flag is `scr1pt1ng_sl4ck1ng_and_h4ck1ng@flare-on.com`.

Props to:

Tyler Dean for making sense of the documentation to identify the right exceptions to enable in WinDbg to ignore single-step exceptions. Tobias Krueger for finding a flaw in which RC4 keytext was excessively long and was discarded from the RC4 key scheduling algorithm allowing the player to solve the challenge without finding the second half of the key! Alexander Polyakov for bringing to my attention an issue that players were noticing with the `CryptAcquireContext` flags as well as internationalization issues (who would have thought that 2-ish weeks of development and testing on about five different systems would not be enough!). And to Eatbrain (https://eatbrain.net/) for allowing me to make use an ASCII of their logo to greet players who beat this challenge.