

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" circumstances¹). 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 missed².

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.





```
about:blank
                 X
    24 <script language='JavaScript'>
   25 function hint(s) {
   26
           document.getElementById('status').innerText = 'You\'re on to something!' + s
   27 }
   28
   29 function strhash2(s) {
   30
          // Adapted from:
          // https://stackoverflow.com/questions/7616461/generate-a-hash-from-string-in-javascript-jquery
   31
   32
         var hash = 0, i, c;
   33
         for (i=0; i<s.length; i++) {</pre>
   34
               c = s.charCodeAt(i);
   35
               hash = ((hash << 5) - hash) + c;
               hash |= 0;
   36
   37
           }
   38
          return hash;
   39 }
   40
   41 function strhash(s) {
           // Adapted from:
   42
    . .
           ....
                    ....
```

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.





Image	Performance	Performance	Graph	Disk and N	letwork	GPU Grap
Threads	TCP/IP	Security	Environ	ment	Job	Strings
Printable str	ings found in the s	can:				
You are ab ASCII Art e Caution: E	out to run the coo ditor on earth. Cor kplosively Neat Pro	lest ntinue? ogram				,
createtextfi gimmethats	ile sweetsweetcrazylo	ve				
run getspecialf	older					
wimmymeb	rah					
! This progra	am cannot be run i	in DOS mode.				
3Rich						
text 'edata						
.ruata @ data						
@.uata						
.reloc						
3VVV3						
y(Fria)						
244 A						
VV VV F						
ury ut-C						
NPL						
SVA/						
SWW						
VPi						
5\/\/i						
OSi						
PVi						
VW						
bl A						
PM/h						~
🔿 Image	Memory			Sa	ve	Find
				_		-

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.





🕼 [No Name] + - GVIM5	- 🗆 ×
<u>F</u> ile <u>E</u> dit <u>T</u> ools <u>S</u> yntax <u>B</u> uffers <u>W</u> indow <u>H</u> elp	
스 모 🛱 본 9 ଓ X 🗈 🏚 🌺 🏵	📥 🍰 🎗 🕆 🖓 💶 ? 🎗
1 Image	1 Henory A
2	- 2
4 This program cannot be run in D	4 ! This program cannot be run in DOS
5 "DHb	5 "DHb
ó "nkWNE	ó "nkWNE
7 #ki3%j	7 #ki3%j
+ 8 + 96 lines: \$-Jnq	- + 8 + 96 lines: \$-Jnq
104 ~qryS	104 ~qryS
105 ~wtx	105 ~wtx
196 +Au <f< td=""><td>106 +Au<f< td=""></f<></td></f<>	106 +Au <f< td=""></f<>
107 KOHLT	107 <bht< td=""></bht<>
198 <dhb< td=""><td>1<u>88</u> <dhb< td=""></dhb<></td></dhb<>	1 <u>88</u> <dhb< td=""></dhb<>
189 <1v,\v	109 <lu,\u< td=""></lu,\u<>
	119 Ju@fJu@
118 =+>0>C>0>1>w>	111 =+>0>C>Q>1>w>
111 =a=j=o=	112 =a=j=o=
✓ [No Name] [+] 2,1 To	p [No Name] [+] 2,1 Top v

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.





00401045 0040104A	<pre>mov esi, offset blob1 nop word ptr [eax+eax+00h]</pre>
🗾 🔏 📼	
00401050	
00401050 loc 40	1050: : flProtect
00401050 push	4
00401052 push	3000h ; flAllocationType
00401057 push	[<mark>esi</mark> +padded_blob.blob.size] ; dwSize
00401059 push	0 ; lpAddress
0040105B call	eax ; VirtualAlloc
0040105D mov	[<mark>esi</mark> +padded_blob.blob.dyn], eax
00401060 test	eax, eax
00401062 jz	100_4011F8
📕 🚄 🖼	
00401068 mov	<pre>ecx, [esi+padded blob.blob.src]</pre>
0040106B push	dword ptr [esi] ; Size
0040106D push	dword ptr [ecx] ; Src
0040106F push	eax ; Dst
00401070 call	memcpy
00401075 add	esp, 0Ch
00401078 lea	eax, [ebp+f101dProtect]
0040107E push	eax ; IpTIOIdProtect
0040107F push	PAGE_NUACCESS ; TINEWProtect
00401081 push	[esitpadded_blob.blob.size] , dwsize
00401085 pash	ds:WintualProtect
0040108C test	eax, eax
0040108E jz	loc 4011F8
	_
	· · · · · · · · · · · · · · · · · · ·
🗾 🚄 🖼	
00401094 mov	eax, ds:VirtualAlloc
00401099 inc	edi
0040109A add	esi, size padded_blob
0040109D cmp	edi, dword_40C388
004010A3 jb	short loc_401050
	₩

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	<mark>esi</mark> , 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.

îj
tj
1





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	; Attrib	utes: bp-based frame
00401990		
00401990	sub_4019	90 proc near
00401990		
00401990	arg_0= d	word ptr 8
00401990		
00401990	push	ebp
00401991	mov	ebp, esp
00401993	mov	eax, [ebp+arg_0]
00401996	mov	fptr_array, offset <mark>fptr_table</mark>
004019A0	mov	<pre>fptr_table, offset sub_401FA0</pre>
004019AA	mov	<pre>fptr_table+4, offset sub_401970</pre>
004019B4	mov	<pre>fptr_table+8, offset sub_402030</pre>
004019BE	mov	<pre>fptr_table+0Ch, offset sub_401B10</pre>
004019C8	mov	<pre>fptr_table+10h, offset sub_401AB0</pre>
004019D2	mov	<pre>fptr_table+14h, offset sub_4019F0</pre>
004019DC	mov	<pre>fptr_table+18h, offset sub_401B30</pre>
004019E6	mov	dword ptr [eax], offset fptr_array
004019EC	xor	eax, eax
004019EE	рор	ebp
004019EF	retn	
004019EF	sub_4019	90 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 IDispatch³, 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	; Attrib	outes: bp-based frame
00401990		
00401990	<pre>setup_ID</pre>	Dispatch proc near
00401990		
00401990	ppv= dwo	ord 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	<pre>vtbl.GetTypeInfoCount, offset GetTypeInfoCount</pre>
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	рор	ebp
004019EF	retn	
004019EF	setup_ID	Dispatch 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	ØxCAFEBABE	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	ØxDEADBEEF	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.





	31 3	30 29	9 28	27	26 :	25 24	123	3 22	21	20	19	18	17	16	15	14	13	12 1	11	10	g	8	7	6 !	5,4	4 3	3 2	2_1	0
	0	0 0	0	0	0	0 0	0	0	I D	V I P	V I F	A C	∨ M	R F	0	N T	I O P L		0 F	D F	F	T F	5	Z F	o í	A F	D F	1	CF
 X ID Flag (II X Virtual Intel X Virtual Intel X Alignment (I X Virtual-808 X Resume F X Nested Tas X I/O Privileg S Overflow F C Direction F X Interrupt En X Trap Flag (I S Sign Flag (I S Sign Flag (I S Parity Flag S Carry Flag)) errupt Tupt Chec 5 Mc 5 Mc 6 Mc 6 Mc 6 Mc 1 ag (lag 1 able 1 ag 1 able 1 ag 1 able 7 F) 2 F) 2 F) 2 F) 2 F) 2 F) 2 F) 2 F) 2	ot P t Fla ck / ode (OF (DF e Fl Fla) —	en ag (V ()	din (VI cce M) DPI (IF	g ((F)- ss	Cor) ntro) -																			
S Indicates a C Indicates a X Indicates a	Stat Cor Sys	tus ntro sterr	Fla I F 1 F	ag lag lag																									
Reserve Always	d bit set to	t po o va	sit alu	ion es	s. [pre)O vio	N(usl	OT ly r	US ead	E.																			

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 reencoding 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 \$\$ <mark>(2)</mark>
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
<pre>leet_editr+0x11cc:</pre>
011711cc ffd6 call esi {00840000}
0:000> ba el @esi \$\$ <mark>(3)</mark>
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 ef1=00000246
0:000> u @eip
00840000 ?? ???
^ Memory access error in 'u @eip'
0:000> p
eax=0053fdb0 ebx=00000000 ecx=73e44063 edx=00000000 es1=00840000 ed1=75df5c20
eip=00840001 esp=0053fd90 ebp=0053fee0 iop1=0 nv up ei pl zr na pe nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b ef1=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 $0 \times 4041D8$ 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 for th3 Dr4m4t1(C)'\r\n" as shown in Figure 13.

004043D8 004043DB 004043E1 004043E1 004043E3	83 81 75 49	C4 FE 51	14 48 20	69 49	6E 20	74 77	65	72	; 65+aIfIWereToTitle	add cmp jnz db 'If]	esp, 14h esi, 'tniH' short loc_404434 : were to title this piece, it would be ',27h,'A FLARE f6	0r th3'
004043E3 0040442C 0040442C 0040442F 00404432	20 8B 8B EB	74 5D 7D 08	6F 08 F8	20	74	69	74	6C	65+ ;	db '_Dr4 mov mov jmp	m4t1(C)',27h,0Dh,0Ah ebx, [ebp+8] edi, [ebp-8] 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
- 1	STACK:00420F90	aOleaut32D11_0 db 'oleaut32.d11',0
- 1	STACK:00420F9D	db 0
	STACK:00420F9E	db 0
Ĩ	STACK:00420F9F	
	STACK:00420FA0	
	STACK:00420FA0	text UIF-16LE, poo,0
	STACK:00420FA8	db opn
	STACK:00420FA9	ab = 24n; p
	STACK:00420FAA	uu ooli ; o
	UNKNOWN 00420F56: STA	ACK: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.

📑 Registry Editor								
File Edit View Favorites Help								
Computer\HKEY_CLASSES_ROOT\Interface\{0E59F1D3-1FBE-11D0-8FF2-00A0D10038BC}								
{0E59F1D3-1FBE-11D0-8FF2-00A0D10038BC}	^ Nam	ne	Туре	Data				
ProxyStubClsid32	ab (Default)	REG SZ	IScriptControl				
In TypeLib								

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.

👫 Registry Editor									
File Edit View Favorites Help									
Computer\HKEY_CLASSES_ROOT\Interface\{0E59F1D3-1FBE-11D0-8FF2-00A0D10038BC}\TypeLib									
	^	Name (Default) ab Version	Type REG_SZ REG_SZ	Data {0E59F1D2-1FBE-11D0-8FF2-00A0D10038BC} 1.0					

Figure 15: IScriptControl IID





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.





내 ITypeLib Viewer	— — »	<
<u>F</u> ile <u>V</u> iew		
🖬 🛍 💡		
 MSScriptControl (Microsof ^ Ms module ScriptControlCc Ms module ScriptControlCc Ms typedef enum ScriptCor dispinterface IScriptProcedu dispinterface IScriptProcedu dispinterface IScriptProcedu dispinterface IScriptProcedu dispinterface IScriptProcedu dispinterface IScriptModule dispinterface IScriptModule dispinterface IScriptModule dispinterface IScriptModule dispinterface IScriptTror dispinterface IScriptError dispinterface IScriptError dispinterface IScriptControl Gispinterface IScriptControl dispinterface DScriptCo 	<pre>// Generated .IDL file (by the OLE/COM Object Viewer) // // typelib filename: msscript.ocx [uuid(0E59F1D2-1FBE-11D0-8FF2-00A0D10038BC), version(1.0), helpstring("Microsoft Script Control 1.0"), helpfile("MSSCRIPT.HLP"), helpcontext(0x00113f4c)] library MSScriptControl { // TLib : OLE Automation : {00020430-0000-0000-c000-00000000046} importlib("stdole2.tlb"); // Forward declare all types defined in this typelib interface IScriptProcedure; interface IScriptProcedure; interface IScriptModule; interface IScriptModule; interface IScriptEror; interface IScriptControl; </pre>	~
кеаду		

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.





						_
👪 [N	o Name] + - GV	IM	_		\times	
<u>F</u> ile <u></u>	<u>E</u> dit <u>T</u> ools <u>S</u> y	ntax <u>B</u> uffers	<u>W</u> indo	w <u>H</u> elp	0	
48	n Dala	G X D	n de l	🔉 🂫	€	Ι.
		9 00 ~				' '
10	ypedef stru	ict IScript	Contr	olvtbl		^
2 {		0	F			
3	FHRPRUC	QueryInter	tace;			
4	FHKFKUL	HUUKEt;				
) x	F HKFKUU Fadddoc	Release;	ofour	+.		
U 7	F HRFRUC Eadddor	CotTupoInf	00000	ι,		
r Q	FARPROC	CotIncOfNa	moc.			
0	FARPROC	Inunke:	mes,			
16	FARPROC	net Lannua	ne :			
11	FARPROC	nut Langua	92, NP:			
12	FARPROC	get State:	9-,			
13	FARPROC	put State:				
14	FARPROC	put SitehW	Ind ;			
15	FARPROC	qet SitehW	nd;			
16	FARPROC	qet Timeou	it;			
17	FARPROC	put_Timeou	it;			
18	FARPROC	get_AllowU	Ι;			
19	FARPROC	put_AllowU	Ι;			
20	FARPROC	get_UseSaf	eSubs	et;		
21	FARPROC	put_UseSaf	eSubs	et;		
22	FARPROC	get_Module	s;			
23	FARPROC	get_Error;				
24	FARPROC	get_CodeOb	ject;			
25	FARPROC	get_Proced	ures;			
26	FARPROC	_AboutBox;				
27	FARPROC	AddObject;				
28	FARPROC	Reset;				
29	FARPROC	AddCode;				
30	FARPROC	Eval;				
31	FARPRUC	ExecuteSta	temen	τ;		
32	FARPRUC	Run;				
33 }	ISCRIPTION	Croivtbi;				*

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	<pre>eax, [ecx+IScriptControlVtbl.put_Language]</pre>
004044DD	call	<pre>eax ; IScriptControl->put_Language(this, BSTR)</pre>
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.





decode2.py (C:\Exclusions\src\c\fustuck\veh_sgx_wannabe\solution) - GVIM2	_		×
<u>F</u> ile <u>E</u> dit <u>T</u> ools <u>S</u> yntax <u>B</u> uffers <u>W</u> indow <u>H</u> elp			
Ө 묘 🖫 뵨 ୭ ଡ ୪ ा ा ा ฿ 원 원 ≛ ≛ 원 ĩ 🦚 💶 ? 원			
∧ # Read WinDbg breakpoint output, emit decoded UBScript	100a70004	27	00 ^
import re	1 <mark>00a70004</mark>	27	00
	100a70006	20	00
script = <mark>()</mark>	00a70008	: 20	00
prog = re.compile('([0-9a-fA-F]{8})\s+([0-9a-fA-F]{2}) [0-9a-fA-F]{2}')	00a7000a	20	00
with open('decode2.txt', 'r') as f:	00a7000c	: 20	00
<pre>for line in f.readlines():</pre>	00a7000e	20	00
m = prog.match(line)	00a70010	20	00
if m:	1 <mark>00a70012</mark>	20	00
addr = int(m.group(1), 16)	1 <mark>00a70014</mark>	20	00
n = int(m.group(2), 16)	100a70016	20	00
if n:	100a70018	20	00
script[addr] = chr(n)	100a7001a	20	00
vbs = ''	100a7001c	: 20	00
<pre>for k in sorted(script):</pre>	00a7001e	20	00
<pre>vbs += script[k]</pre>	100a70020	20	00
	00a70022	20	00
print(vbs)	100a70024	20	00
	00a70026	20	00
v decode2.py 4,11 A1	I <e2.txt< td=""><td>1,1</td><td>Top 🗸</td></e2.txt<>	1,1	Top 🗸

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.





6	[No Name] + - GVI	M4					_		×
<u>F</u> ile	<u>E</u> dit <u>T</u> ools <u>S</u> ynt	tax <u>B</u> uffers <u>N</u>	<u>W</u> indow <u>H</u> elp						
90	9 C 4 9 C	X 🗈 💼	🗞 🔁 🔁 📥	🎗 🕆 🛱 🚥	<u>የ የ</u>				
1 2			_a, _W#n	1,					^
3			Wmmn	1m/					
- 4	'BmmBmmBmm[Bmm	a#mmmn	ımB/	BmmBmmBm6a	3BmmBmmBm			
5	'mmm[mmm	j##mmmn	1mmm6	mmm -4mm[3mm[
6	'mBmLaaaa,	Bmm	JŴ#mmP 4	ImmmmL	mmBaaaa#mm [†]	3Bm6aaaa,			
7	'mmmP!!"?'	mmm	JWmmmP	4mmmBL	Bmm!4X##"	3mmP????'			
8	'Bmm[Bmmaaaaa	jWmmm?	4mmmBL	mmm !##L,	3BmLaasaa			
9	'mmm[mmm##Z#Z	_jWmmmmaaaaa	aa,]mBmm6.	mmB "#Bm/	3mmm#UZ#Z			
10			_WBmmmmm#Z#Z‡	ŧ! "mmmBm,					
11			??!??#mmmm#!	"??†??	>				
12			. JmmmP'						
13			_jmmP'						
-14			_JW?'						
15			"?						
16									
17	Di∎ Page(118)							
18	Page(0) = "<	!doctype	html>"						
19	Page(1) = "<	html>"							\sim
							1,1	Т	ор

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.

about:bla	nk			- Ç	Search		{	□ \$ \$ \$ \$	×
						- 11.			۹.
	'''	-=<[Cryp	optimized	s' ASCI L for TH	I Art	Editor]	>=-''·	'	\sim
You're on to s	something!	- But di	d vou thi	nkofa	atit'	le for vou	r master	piece?	
' BmmBmmBmm['mmm['mBmLaaa, 'mmmP!!"?' 'Bmm['mmm['	Bmm mmm Bmm Bmmaaaaa mmm##2#2	a#mmm j##mmm j##mmm JW#mmP jWmmm? jWmmmaaaaa WBmmmmm#Z#2 ??!??#nmmmm#Z#2 .JmmmP' jmmP' JW?' "?	A, m, mmB/ mmmB/ mmmmBL 4mmmBL 4mmmBL 4aa,]mBmm6. #! "mmmBm "??!?	BrumBrum mrum - mrumBaaaa Brum ! 4x mrum ! 4 mrum ! 4 ?	Bm6a -4mm[a#nm' ##" ##L, "#Bm/	3BmmBmmBm 3mm[3Bm6aaaa, 3mmP????' 3BmLaasaa 3mmm#UZ#Z			~

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.

Carlo Dachard ASCII A 1 5	Search	۲ م] 🏠	□ ☆ ŵ	× } 🙂
Crypto-Donkeys' ASCII Art E X		~			^
If I were to title this piece, it would be 'A_FLARE_fOr_th3_Dr4m4t1(C)'					

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.

Crypto-Donkeys' ASCII Art E × 1		▼ 🖒 Search	- ■ × ♪ ☆ @ ©
safipiling_slibaking	_emd_hH	Faking@Alar	e=on:cin
5 5 5 5 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7	λ λ λ δ4 λ	. 14000. . 191400042 ^ 24000400 * 2400040 * 240000* 24000* 24000* 24000* 24000* 24000* 24000* 24000* 24000* 24000* 24000* 24000* 24000* 2400* 24000* 24000* 2400	
3643004402440 3660064004 - 3460006 - 3460006 - 346000 - 346000 - 34600 - 35600 - 35600 - 3600 - 36000 - 36000	ktoristing ktoristing ktoristing ktoristing ktorig ktorig	rn.earna contact na Lea an Lea contact a contact a contact sector a contact a contact a contact a contact a contact sector a contact a contact a contact a contact a contact a contact sector a contact a contact a contact a contact a contact a contact sector a contact a contact a contact a contact a contact a contact sector a contact a contact a contact a contact a contact a contact sector a contact a contact a contact a contact a contact a contact a contact sector a contact	
		sanjeti	ng_sH-eking_

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 (<u>https://eatbrain.net/</u>) for allowing me to make use an ASCII of their logo to greet players who beat this challenge.

Figure 28: Sweet, sweet flag