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](image)

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](https://twitter.com/alex_k_polyakov/status/1042844336902299648) and [https://twitter.com/stuxxn/status/1042498255026716672](https://twitter.com/stuxxn/status/1042498255026716672)

² See [https://twitter.com/justadrawer2/status/104104082936726656](https://twitter.com/justadrawer2/status/104104082936726656)
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.
Basic Static Analysis

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

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.
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.
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.

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.
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.

```
00401900  ; Attributes: bp-based frame
00401990  sub_401990 proc near
00401990  arg_0= dword ptr 8
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
004019A4  mov   fptr_table+4, offset sub_401970
004019B1  mov   fptr_table+8, offset sub_402030
004019B8  mov   fptr_table+0Ch, offset sub_401B10
004019C5  mov   fptr_table+10h, offset sub_401AB0
004019D2  mov   fptr_table+14h, offset sub_4019F0
004019D8  mov   fptr_table+18h, offset sub_401830
004019E6  mov   dword ptr [eax], offset fptr_array
004019E8  xor   eax, eax
004019EE  pop   ebp
004019F0  ret
004019F0  sub_401990 endp
004019F8
```

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.

![Assembly code snippet](image)

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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Magic Number</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>createtextfile</td>
<td>0xCAFEBABE</td>
<td>RC4 decrypt and return a BSTR (OLE automation string type) associated</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Bitmask value</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>XOR</td>
</tr>
<tr>
<td>2</td>
<td>Incrementing XOR</td>
</tr>
<tr>
<td>4</td>
<td>RC4</td>
</tr>
<tr>
<td>$0x80$</td>
<td>Combination of three encodings (In source code, I called it Neapolitan)</td>
</tr>
</tbody>
</table>

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 (\texttt{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=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 el @esi  $$  (3)
0:000> g
Breakpoint 1 hit
eax=0053fdb0 ebx=00000000 ecx=73e44063 edx=00000000 esi=00840000 edi=75df5c20
eip=00840000 esp=0053fee0 iopl=0  nv up ei pl zr na pe nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b  efl=00000246
00840000 ??  ???
0:000> u @eip
00840000 ??  ???
  ^ Memory access error in 'u @eip'
0:000> p
eax=0053fdb0 ebx=00000000 ecx=73e44063 edx=00000000 esi=00840000 edi=75df5c20
eip=00840001 esp=0053fee0 iopl=0  nv up ei pl zr na pe nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b  efl=00000246
00840001 7512 jne 00840015  [br=0]
0:000> u @eip-1  $$  (4)
00840000 55  push ebp
```
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 IDA Python one-liner in Listing 3 can be used to apply this to the shellcode region.

```python
for n in range(0x70): PatchByte(here()+n, Byte(here()+n) ^ 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.

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 CIP](image)

**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.
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.

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.
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 20: Disabling MIDL warnings 2400 and 2401
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  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]
004044DB    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>)
6ab68310  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

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

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

![Image of marquee version of flag](image.png)

**Figure 28: Sweet, sweet flag**

The flag is scripting_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/](https://eatbrain.net/)) for allowing me to make use an ASCII of their logo to greet players who beat this challenge.