FireEye FLARE

Challenge #9 Solution

by Nick Harbour

This challenge is a new and improved version of challenges #1 and #2, but this time it is back for revenge. This program is riddled with anti-disassembly techniques and is designed to make static analysis painful. Use of a debugger is highly recommended.

If you look at the beginning of the code (at 0x401000) it contains what appears to be a program very similar to challenge #2. This code leads to what appears to be a comparison function called sub_401495 which is very long and complex. These are decoys, they never get called. The real program still performs similar behavior just not with this code.

Starting at the beginning of the "real" program, there is first a jump to a small block of code ending in a return. Detailed inspection will reveal that this code is building a pointer on the stack that the RETN instruction will return to. That pointer value is 0x401091 and is the real start of the program. The instruction at this location is "`xor eax, eax`" followed by a JZ instruction. In cases such as this, the JZ instruction is fake because the zero flag will always be set when it executes. Its role in this program is to attempt to throw off the disassembler.

```
.text:00401091                 xor     eax, eax
.text:00401093                 jz      short near ptr loc_401095+1
.text:00401095
.text:00401095 loc_401095:                             ; CODE XREF: .text:00401093↑j
.text:00401095                 call    near ptr 9B4D323h
.text:0040109A                 push    ebp
```

Figure 1: Anti-Disassembly fragment

The `jz` will always be taken but the target of the jump is inside of what IDA Pro thinks is a call instruction. To fix this, you can put your cursor on the line with the call instruction, then press 'd' to turn the instruction into data. Select the address that is the real jump target and press 'c' too turn the address into code. This trick is used many times throughout this challenge, with many variations and added complexities. If you are attempting to fix them in IDA for static analysis, the first few sequences should look as follows if you have correctly fixed the anti-disassembly:

```
.text:00401091                 xor     eax, eax
.text:00401093                 jz      short loc_401096
.text:00401093 ; ---------------------------------------------------------------------------
.text:00401095                 db 0E8h                 ; junk
.text:00401096 ; ---------------------------------------------------------------------------
.text:00401096
.text:00401096 loc_401096:                             ; CODE XREF: .text:00401093↑j
.text:00401096                 mov     edx, eax
.text:00401098                 jz      short loc_4010A3
.text:0040109A                 push    ebp
.text:0040109B                 mov     ebp, esp
.text:0040109D                 sub     esp, 10h
.text:0040109D ; ---------------------------------------------------------------------------
.text:004010A0                 db 0E8h                 ; junk
.text:004010A1 ; ---------------------------------------------------------------------------
.text:004010A1
.text:004010A1 loc_4010A1:                             ; CODE XREF: .text:loc_4010A3↓p
.text:004010A1                 jmp     short loc_4010A9
.text:004010A3 ; ---------------------------------------------------------------------------
.text:004010A3
.text:004010A3 loc_4010A3:                             ; CODE XREF: .text:00401098↑j
.text:004010A3                 call    loc_4010A1
```

Figure 2: Fixed anti-disassembly sequence

Another aspect to the program that makes it difficult to reverse engineer statically is that instead of performing its actions sequentially in a function it instead operates in a return-oriented fashion. This means that it builds the stack in a way such that the call to one function will directly return to the next function the program calls, with several "glue" functions in between where needed.

Another trick used extensively by this program is building and executing instructions on the stack. The following fragment shows a sequence of instructions which build and execute the instruction "`mov [esp+268], eax`" on the stack.

```
.text:0040116D                 mov     [esp+108h], eax
.text:00401174                 lea     eax, [ebp-8]
.text:00401177                 lea     ebx, [esp-10h]
.text:0040117B                 mov     dword ptr [ebx], 10248489h
.text:00401181                 mov     dword ptr [esp-0Ch], 0C3000001h
.text:00401189                 mov     [esp-8], ebx
.text:0040118D                 mov     dword ptr [esp-4], 119Dh
.text:00401195                 add     [esp-4], edi
.text:00401199                 sub     esp, 8
.text:0040119C                 retn
```

Figure 3: Executing instructions from the stack

Starting at 0x40175B is the computation function which is called once for each byte of the key, and compares it with the input string. It builds 4 tables on the stack, the first portion of which is not protected by anti-disassembly and is shown in the fragment below. (It actually builds 5 tables, one of them is just a decoy).

```
.text:00401768                 mov     dword ptr [ebx+0D8h], 9FAB3165h
.text:00401772                 mov     dword ptr [ebx+0D4h], 3960C446h
.text:0040177C                 mov     dword ptr [ebx+0D0h], 0A8EF74F1h
.text:00401786                 mov     dword ptr [ebx+0CCh], 0FF9FE6B2h
.text:00401790                 mov     dword ptr [ebx+0C8h], 0BDB54632h
.text:0040179A                 mov     dword ptr [ebx+0C4h], 0EF26678Eh
.text:004017A4                 mov     dword ptr [ebx+0C0h], 0EB0DC9EBh
.text:004017AE                 mov     dword ptr [ebx+0BCh], 4C96E345h
.text:004017B8                 mov     dword ptr [ebx+0B8h], 344A408Eh
```

Figure 4: Building tables on the stack

The rest of the tables are built in the same fashion, but with anti-disassembly code sprinkled throughout to obfuscate their construction. The four tables built on the stack are the xor key (at ESP+176), a lookup table (ESP+88), a rotate table (at ESP+132), and the answer key (at ESP+44).

For each index in the input, the lookup table is accessed twice. A table index for the xor key, answer key , and rotate table are generated by lookuptable[lookuptable[i]], where "i" is the index of the input byte. The input byte will be XOR'ed with the value at xorkey[index]. The byte is then rotated by the amount specified by rotatetable[index]. Then a value is fetched from the answer key (answerkey[index]) and compared with the computed byte. The comparison occurs in the form of a cmpxchg instruction which is executed off of the stack, as shown in the following fragment of code.

```
.text:00401B4F                 mov     dword ptr [esp-0Ch], 0C3D3B00Fh ; cmpxchg bl, dl
.text:00401B57                 push    5C841571h
.text:00401B5C                 push    esp
.text:00401B5D                 sub     dword ptr [esp+4], 5C43FA07h
.text:00401B65                 sub     dword ptr [esp], 8
.text:00401B69                 retn
```

Figure 5: Stack Execution of cmpxchg instruction

A single anti-debug step occurs following this, where the instruction "`mov ebx, [fs:0x30]`" is executed from the stack (at program offset 0x401B87-0x401BA1) then the NtGlobalFlag is accessed. Its value is added to the running total of correct values to the function, and thus a non-zero value (in the presence of a debugger) which would cause the program to produce incorrect results.

The following python script uses three of the four tables and deduces a key by using the inverse of the program logic.

```python
def decode9():
    key = [0x46, 0x15, 0xf4, 0xbd, 0xff, 0x4c, 0xef, 0x46, 0xeb, 0xe6,
           0xb2, 0xeb, 0xf1, 0xc4, 0x34, 0x67, 0x39, 0xb5, 0x8e, 0xef,
           0x40, 0x1b, 0x74, 0x0d, 0x60, 0x26, 0x45, 0xa8, 0x4a, 0x96,
           0xc9, 0x65, 0xe2, 0x32, 0x60, 0x64, 0x8c, 0x65, 0xe3, 0x8e,
           0x9f]
    t1 = [0xc3, 0xcc, 0xba, 0x4e, 0xf2, 0xeb, 0x27, 0x19, 0xc6, 0x42,
          0x06, 0x16, 0x5d, 0x53, 0x55, 0x0e, 0x66, 0xf4, 0xf9, 0x30,
          0x9a, 0x77, 0x56, 0x6b, 0xf0, 0x8e, 0xdc, 0x2e, 0x50, 0xe1,
          0x5a, 0x80, 0x48, 0x5d, 0x53, 0xc2, 0xb8, 0xd2, 0x01, 0xc3,
          0xbc]
    t2 = [0x56, 0xf5, 0xac, 0x1b, 0xb5, 0x93, 0x7e, 0xb8, 0x23, 0xda,
          0x0a, 0xf2, 0x01, 0x61, 0x5c, 0xc8, 0x4c, 0xd6, 0x16, 0x55,
          0x67, 0xb8, 0xc1, 0xf8, 0xbc, 0x11, 0xfa, 0x9b, 0x6b, 0xf9,
          0xd4, 0x75, 0x87, 0xca, 0xce, 0xbe, 0x4e, 0x6e, 0xf1, 0xb9,
          0x6e]

    result = []
    for i in xrange(len(key)):
        val = t1[i]
        val = ROR(val, t2[i], bits=8)
        val ^= key[i]
        result.append(chr(val))
    return ''.join(result)

def ROR(x, n, bits = 32):
    n = n % bits
    mask = (2L**n) - 1
    mask_bits = x & mask
    return (x >> n) | (mask_bits << (bits - n))

print decode9()
```

Figure 6: Decoding Script