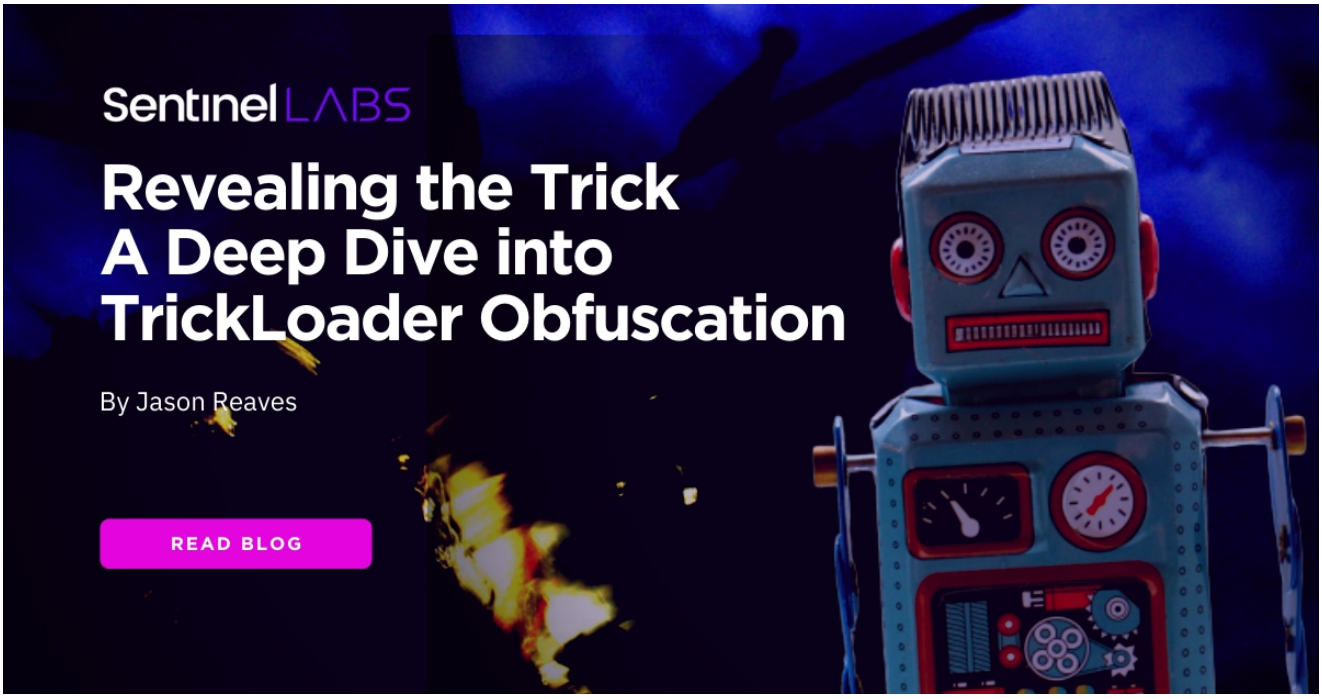
Revealing the Trick | A Deep Dive into TrickLoader Obfuscation

labs.sentinelone.com/revealing-the-trick-a-deep-dive-into-trickloader-obfuscation/

Jason Reaves



Within the TrickBot framework, there has historically been a loader component. This loader has had continued development over the years since TrickBot's first release where the ECS key and bot binary were stored in the resource section of the loader [1]. However, the function obfuscation has received relatively little treatment until now.

Executive Summary

- TrickBot developers have continued to be active over the years.
- Loader used by TrickBot has had continued development related to obfuscation for anti-analysis.
- The TrickLoader leverages 'minilzo' compression, which comes from the LZO library and its usage by these developers dates back to Dyre/Upatre timeframe.
- The goal is to detail the loader and aid additional automation efforts to process the TrickLoader.

Research Insight

TrickLoader obfuscation development timeline:

2017 – Started obfuscating the resource section name
2017 – Custom base64 of strings
2018 – Adds user account control (UAC) bypass [5], Heaven's Gate [2], function obfuscation and further hiding the configuration

Most of these have been reported on in detail with the exception of the function obfuscation, which has been mentioned but not really detailed. Researchers who write scripts for config retrieval have stopped putting them out as frequently as in the past, possibly due to the increased focus by TrickBot to obfuscate and hide the data.

Let's dive into the obfuscation. The function offsets are stored in a table. The first thing the loader does is execute a call over that table that will push the address of the table onto the stack for the next block of code to use.

```
                public start
start:
                push    ebp
                mov     ebp, esp
                call    loc_401090      ; ret address holds offset table
                test    dword ptr [eax], 2C8006Dh
                push    esp
                add     ah, ah
                add     [eax], ah
; ---------------------------------------------------------------------------
                db 0
                dd 1B00300h, 500040h, 0A40038h, 540074h, 0E40170h, 0B40104h
                dd 1BC01C0h, 0A40118h, 0E000D4h, 7C0030h, 70004Ch, 0B0001Ch
                dd 88001Ch, 224002Ch, 0FFF000E0h, 0FFF100C4h, 4800B4h
                dd 600064h, 5802D8h, 300040h, 1C40278h, 1D4009Ch, 0C8003Ch
                dd 2000058h, 440048h, 5C003Ch, 21C08BCh, 2683A78h, 300070h
                dd 5800F4h, 1Ch
; ---------------------------------------------------------------------------

loc_401090:                             ; CODE XREF: .text:00401003↑p
                pop     eax             ; ret address holds offset table
                mov     edi, eax
```

Figure 1: Call over offset table

The next section will then process the word values from the table in sequence by adding them to a value which is initially the start address of the table and then being pushed onto the stack.

```
                mov     eax, edi
                add     eax, 2A5FCh
                push    0FFF1h          ; negative is a flag
                pop     ecx
                mov     [ebp+4], eax
                mov     edx, edi
                mov     esi, edi
                dec     ecx

c_4010B6:                               ; CODE XREF: .text:004010D8↓j
                mov     eax, ecx
                lodsw                   ; load a word from table
                test    eax, eax
                jz      short loc_4010DA
                cmp     eax, ecx
                jb      short loc_4010D5 ; If >= 0xfff0 then:
                sub     eax, ecx
                shl     eax, 2
                push    ecx
                mov     ecx, edi
                add     ecx, eax        ; add to original offset of table
                add     ecx, 2B70Bh     ; add to that
                mov     eax, [ecx]      ; get dword to add
                pop     ecx

c_4010D5:                               ; CODE XREF: .text:004010C0↑j
                add     edx, eax        ; add to accumulating offset from previous start of offset table
                push    edx             ; push address onto stack
                jmp     short loc_4010B6
-------------------------------------------------------------------------

c_4010DA:                               ; CODE XREF: .text:004010BC↑j
                mov     [ebp+0Ch], eax
                mov     eax, ebp
                mov     ecx, 9
                shl     ecx, 2
                sub     eax, ecx
                mov     eax, [eax]
                mov     [ebp+8], eax
                push    33h
                mov     edx, eax
                pop     ecx
                call    edx             ; 401a7c
                mov     [eax+2], ebp
                push    0Ch
```

Figure 2: Overview of rebuilding addresses from table

Reconstructing this process into Python code allows us to create the same table as long as we can recover certain values from the binary.

```
off = 0x401008
start = 0x401008
orig = 0x401008
guard = 0xfff0
val = Word(off)
vals = []
while val != 0:
  if val >= guard:
    val -= guard
    val <<= 2
    temp = orig + val + 0x2b70b
    val = Dword(temp)
  start += val
  vals.append(start)
  off += 2
  val = Word(off)
```

Figure 3: Python code to demonstrate rebuilding the table manually

After the function table is rebuilt, a call is made to one of the functions that is responsible for decoding out the other functions and data blobs.

```
    mov     eax, ebp
    mov     ecx, 9
    shl     ecx, 2
    sub     eax, ecx
    mov     eax, [eax]
    mov     [ebp+8], eax
    push    33h
    mov     edx, eax
    pop     ecx
    call    edx             ; 401a7c
```

Figure 4: Decode function after rebuilding table

```
decode_401A7C   proc near
                push    ebp
                shl     ecx, 1
                pop     eax
                shl     ecx, 1
                push    ebx
                push    edi
                push    esi
                sub     eax, ecx
                push    eax
                mov     eax, [eax]
                pop     ecx
                mov     edi, eax
                dec     ecx
                dec     ecx
                dec     ecx
                dec     ecx
                mov     eax, [ecx]
                sub     eax, edi
                push    eax
                mov     ecx, eax
                cld
                push    edi
                push    327h
                mov     eax, [ebp+4]
                add     eax, 18h
                pop     ebx
                mov     esi, eax
                push    ebx

loc_401AA9:
                push    ecx
                mov     ecx, [edi]
                mov     eax, [esi]
                xor     eax, ecx
                mov     [edi], al
                inc     esi
                inc     edi
                dec     ebx
                pop     eax
```

Figure 5: Decode function

The function decodes the next function. The key is the last value in the rebuilt table address with 0x18 added to it, and the length of the key is 0x327 bytes. Using this we should be able to decode out all the addresses in the rebuilt table.

Figure 6: Decode all the objects from the table

After decoding all the objects, we can check the sizes of each by printing out the size of every element of the decoded_data list.

```
>>> [x[1] for x in decoded_data]
[109, 712, 84, 228, 32, 768, 432, 64, 80, 56, 164, 116, 84, 368, 228, 260, 180, 448, 444, 280, 164, 212, 224, 48,
124, 76, 112, 28, 176, 28, 136, 44, 548, 224, 72104, 196, 70620, 180, 72, 100, 96, 728, 88, 64, 48, 632, 452,
156, 468, 60, 200, 88, 512, 72, 68, 60, 92, 2236, 540, 14968, 616, 112, 48, 244, 88, 28]
```

Figure 7: Check decoded object sizes

Most of them look normal; however, there are a few that seem larger than what you would normally observe in the size of a single function.

```
>>> [x[1] for x in decoded_data].index(72104)
34
>>> decoded_data[34][2][:100]
bytearray(b'A\xe8\xcb\xf4\x00\x00AVVWSH\x81\xec\x08\x06\x00\x00\x8b\xfaH\x8b\xf1H\x8d\\
$X3\xd2A\xb8\xb0\x05\x00\x00H\x8b\xcb\xe85j\x01\x00\xc7D$(\x00A\x00HA\x01 ]
\x00\x00\x03\x08E3\xc9\x04\x07\x0c\xceL\x8b\xc3\xe8\xf46\x00\x00\x85\xc0\x0f\x84\x84\x02A\x04\x8b\x02
\x08XHu\x01vt\x01\x04\x8dt$h\xc7\x06')
>>> decoded_data[36][2][:100]
bytearray(b'\x1a\x00\x00\x80\x00\x01\x00\x00\x00\x04N\x00\xff\xff\x0e\x00\xb8\x00\xa2\x00@\x00
\x01\x01\x00hL\x00\x00\x01\x0e\x1f\xba\x0e\x00\xb4\t\xcd!\xb8\x01L\xcd!This H\x00\x00\ra PE
executable\r\n$PE\x00\x00L\x01\x03\x00\xe0\xf7\xb7]\x00\xc1\x00\xe0\x04\x0f\x04\x01\x0b\x01\n\x00\x00lE
\x10')
```

Figure 8: Compressed objects

These larger decoded objects are actually compressed data. It turns out there are at least 3 compressed objects: a 32 bit TrickBot binary, a large blob of 64-bit bytecode which is the 64 bit TrickBot binary, and a smaller 64-bit EXE file which is a loader for the 64-bit bytecode blob.

The compression is ‘minilzo’, which comes from the LZO library, and its usage by these developers dates back to Dyre/Upatre timeframe. After decompressing the 32-bit binary and fixing the missing ‘MZ’, we have the 32-bit TrickBot binary.

Now that we have the normal TrickBot binary, we can decode out the onboard configuration data which is hidden and XOR encoded inside the bot now. Taking an existing decoder from CAPE [4] and adjusting it a bit while adding in our deobfuscation works well!

Indicators of Compromise (IOCs)

SHA-256: ac27e0944ce794ebbb7e5fb8a851b9b0586b3b674dfa39e196a8cd47e9ee72b2

```
<mcconf>

<ver>1000480</ver>

<gtag>tot598</gtag>

<servs>

<srv>144.91.79.9:443</srv>
<srv>172.245.97.148:443</srv>
<srv>85.204.116.139:443</srv>
<srv>185.62.188.117:443</srv>
<srv>185.222.202.76:443</srv>
<srv>144.91.79.12:443</srv>
<srv>185.68.93.43:443</srv>
<srv>195.123.238.191:443</srv>
<srv>146.185.219.29:443</srv>
<srv>195.133.196.151:443</srv>
<srv>91.235.129.60:443</srv>
<srv>23.227.206.170:443</srv>
<srv>185.222.202.192:443</srv>
<srv>190.154.203.218:449</srv>
<srv>178.183.150.169:449</srv>
<srv>200.116.199.10:449</srv>
<srv>187.58.56.26:449</srv>
<srv>177.103.240.149:449</srv>
<srv>81.190.160.139:449</srv>
<srv>200.21.51.38:449</srv>
<srv>181.49.61.237:449</srv>
<srv>46.174.235.36:449</srv>
<srv>36.89.85.103:449</srv>
<srv>170.233.120.53:449</srv>
<srv>89.228.243.148:449</srv>
<srv>31.214.138.207:449</srv>
<srv>186.42.98.254:449</srv>
<srv>195.93.223.100:449</srv>
<srv>181.112.52.26:449</srv>
<srv>190.13.160.19:449</srv>
<srv>186.71.150.23:449</srv>
<srv>190.152.4.98:449</srv>
<srv>170.82.156.53:449</srv>
<srv>131.161.253.190:449</srv>
<srv>200.127.121.99:449</srv>
<srv>45.235.213.126:449</srv>
<srv>31.128.13.45:449</srv>
<srv>181.10.207.234:449</srv>
<srv>201.187.105.123:449</srv>
<srv>201.210.120.239:449</srv>
<srv>190.152.125.22:449</srv>
<srv>103.69.216.86:449</srv>
<srv>128.201.174.107:449</srv>
<srv>101.108.92.111:449</srv>
<srv>190.111.255.219:449</srv>

</servs>
```

```
<autorun>

<module name="systeminfo" ctl="GetSystemInfo"/>

<module name="pwgrab"/>

</autorun>

</mcconf>
```

References

1: https://www.fidelissecurity.com/threatgeek/archive/trickbot-we-missed-you-dyre/
2: http://www.hexacorn.com/blog/2015/10/26/heavens-gate-and-a-chameleon-code-x8664/
3: http://www.oberhumer.com/opensource/lzo/
4: https://github.com/ctxis/CAPE
5: https://sysopfb.github.io/malware/2018/04/16/trickbot-uacme.html