Dissecting the Dyre Loader

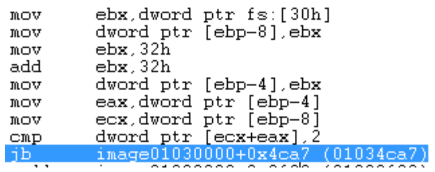
JASON REAVES

November 25, 2015

Abstract

Dyre or Dyreza, is a pretty prominent figure in the world of financial malware. The Dyre of today comes loaded with a multitude of modules and features while also appearing to be well maintained. The first recorded instance of Dyre I have found is an article in June 2014 and the sample in question is version 1001, while at the time of this report Dyre is already up to version 1166. While the crypters and packers have varied over time, for at least the past 6 months Dyre has used the same loader to perform it's initial checks and injection sequence. It is the purpose of this report to go through the various techniques and algorithms present in the loader, and at times reverse them to python proof of concepts.

Keywords - Reverse Engineering, Malware Analysis, Dyreza, Banking Trojan



```
mov     ebx,dword ptr fs:[30h]
mov     dword ptr [ebp-8],ebx
mov     ebx,32h
add     ebx,32h
mov     dword ptr [ebp-4],ebx
mov     eax,dword ptr [ebp-4]
mov     ecx,dword ptr [ebp-8]
cmp     dword ptr [ecx+eax],2
jb      image01030000+0x4ca7 (01034ca7)
```

Figure 1: Processor Check

1 Introduction

The Dyre banking trojan has evolved significantly since it's emergence in June of 2014 and, while it was by no means considered simple for it's time it has definitely grown in its capabilities. While some groups and bankers out there use more advanced techniques and tools any banking trojan has the goal of stealing enough information while utilizing enough tools in its arsenal to ultimately perform fraud against the institutions it is targeting. I would consider the Dyre of today to be among the more advanced forms of malware in the area of banking trojans. In this report we go through the loader used by Dyre, a loader is simply a program used to load various other things(code, other programs, DLLs, etc.).

2 Dyre Loader

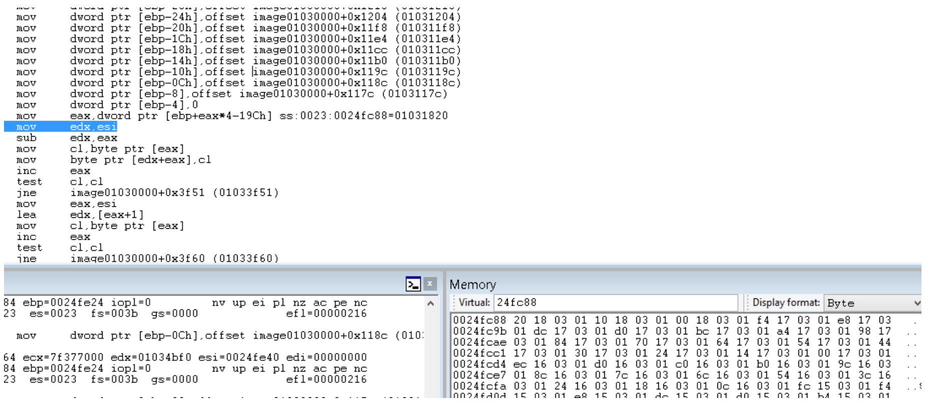
The loader first performs a simple check on the number of processors in the system which appears to be targeting sandboxes(Figure 1). This check was added around April 2015.

Next the loader begins decrypting the dll and function names that it will need. Each step the loader takes will be outlined below.

2.1 String Decrypt

The main function for the string decryption process is called with an index number as an argument indicating which string the calling code wants returned. This function when called puts every offset of every encoded string onto the stack. It then uses the index passed to it to then copy the encoded string into another section of memory, the end of the string is reached when a NULL byte is hit. We can this happening in Figure 2.

After this is done the code passes the section of memory with the encoded string and the length to the function responsible for decrypting it. In Figure 3 we can see the heart of what appears to be a single byte XOR loop over an 8 byte key unless the bytes are the same in which case that byte is left alone. The byte checking portion is turned on or off with flag that gets passed to the routine, it is an attempt at making it safe for unicode strings. However since the unicode strings have their null byte XORd it appears that same check is not done during the encoding process, making the check itself possibly useless code.



```
mov     dword ptr [ebp-24h],offset image01030000+0x1204 (01031204)
mov     dword ptr [ebp-20h],offset image01030000+0x11f8 (010311f8)
mov     dword ptr [ebp-1Ch],offset image01030000+0x11e4 (010311e4)
mov     dword ptr [ebp-18h],offset image01030000+0x11cc (010311cc)
mov     dword ptr [ebp-14h],offset image01030000+0x11b0 (010311b0)
mov     dword ptr [ebp-10h],offset image01030000+0x119c (0103119c)
mov     dword ptr [ebp-0Ch],offset image01030000+0x118c (0103118c)
mov     dword ptr [ebp-8],offset image01030000+0x117c (0103117c)
mov     dword ptr [ebp-4],0
mov     eax,dword ptr [ebp+eax*4-19Ch] ss:0023:0024fc88=01031820
mov     edx,esi
sub     edx,eax
mov     cl,byte ptr [eax]
mov     byte ptr [edx+eax],cl
inc     eax
test    cl,cl
jne     image01030000+0x3f51 (01033f51)
mov     eax,esi
lea     edx,[eax+1]
mov     cl,byte ptr [eax]
inc     eax
test    cl,cl
jne     image01030000+0x3f60 (01033f60)
```

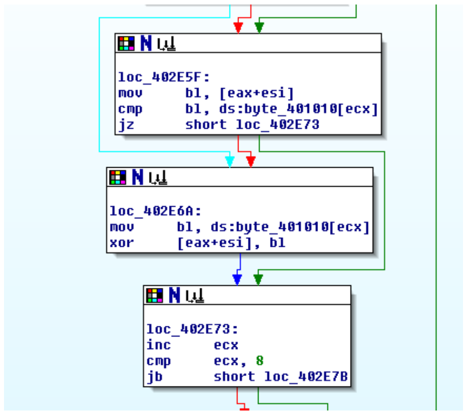
```
84 ebp=0024fe24 iopl=0         nv up ei pl nz ac pe nc
23  es=0023  fs=003b  gs=0000             efl=00000216

  mov     dword ptr [ebp-0Ch],offset image01030000+0x118c (010:

64 ecx=7f377000 edx=01034bf0 esi=0024fe40 edi=00000000
84 ebp=0024fe24 iopl=0         nv up ei pl nz ac pe nc
23  es=0023  fs=003b  gs=0000             efl=00000216
```

Memory

Virtual: 24fc88   Display format: Byte

```
0024fc88 20 18 03 01 10 18 03 01 00 18 03 01 f4 17 03 01 e8 17 03
0024fc9b 01 dc 17 03 01 d0 17 03 01 bc 17 03 01 a4 17 03 01 98 17
0024fcae 03 01 84 17 03 01 70 17 03 01 64 17 03 01 54 17 03 01 44
0024fcc1 17 03 01 30 17 03 01 24 17 03 01 14 17 03 01 00 17 03 01
0024fcd4 ec 16 03 01 d0 16 03 01 c0 16 03 01 b0 16 03 01 9c 16 03
0024fce7 01 8c 16 03 01 7c 16 03 01 6c 16 03 01 54 16 03 01 3c 16
0024fcfa 03 01 24 16 03 01 18 16 03 01 0c 16 03 01 fc 15 03 01 f4
```

Figure 2: Finding which string to decode



```
loc_402E5F:
mov     bl, [eax+esi]
cmp     bl, ds:byte_401010[ecx]
jz      short loc_402E73
```

```
loc_402E6A:
mov     bl, ds:byte_401010[ecx]
xor     [eax+esi], bl
```

```
loc_402E73:
inc     ecx
cmp     ecx, 8
jb      short loc_402E7B
```

Figure 3: Main string decoding section

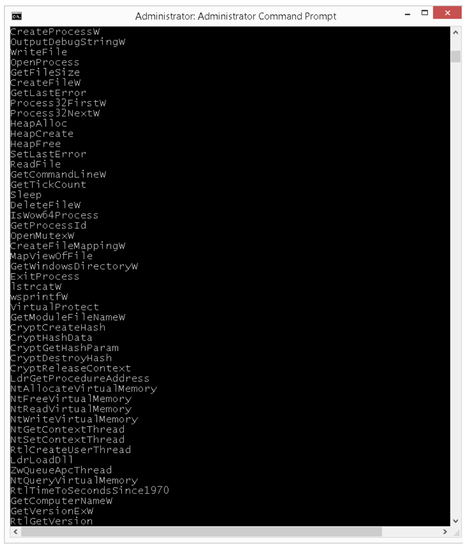
A proof of concept example of this can be seen in Figure 4, and decrypting all of the strings at every offset can give us insight into how the loader might operate(Figure 5).
Taking out the same byte check and running the script against the encoded unicode strings also gives us some interesting strings(Figure 6).

2.2 File Name Generation

Next the loader compares its own privilege level with the first svchost it finds in the process list, the check is performed by comparing the SIDs from the processes respective TOKEN_USER structures. If the comparison is successful then the loader checks if it's running from `C:\windows` if it's not successful then the loader checks if it's running from `%APPDATA%\local`. In either case a random 15 character filename is generated using a custom Psuedo-Random

```python
import binascii

key = bytearray(binascii.a2b_hex('1622f36a8541ca84'))
encoded = bytearray(binascii.a2b_hex('7d478104e02df9b638469f06'))

def decrypt_string(data, key):
        for i in range(len(data)):
                if data[i] != key[i%len(key)]:
                        data[i] ^= key[i%len(key)]
        print(data)

decrypt_string(encoded,key)
#>>> kernel32.dll
```

Figure 4: Loader String Decrypt Example

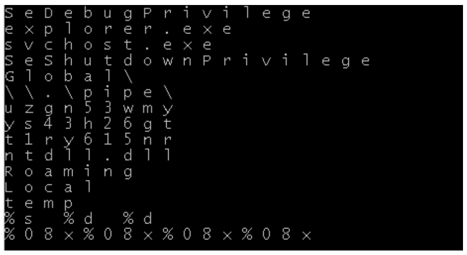


```
CreateProcessW
OutputDebugStringW
WriteFile
OpenProcess
GetFileSize
CreateFileW
GetLastError
Process32FirstW
Process32NextW
HeapAlloc
HeapCreate
HeapFree
SetLastError
ReadFile
GetCommandLineW
GetTickCount
Sleep
DeleteFileW
IsWow64Process
GetProcessId
OpenMutexW
CreateFileMappingW
MapViewOfFile
GetWindowsDirectoryW
ExitProcess
lstrcatW
wsprintfW
VirtualProtect
GetModuleFileNameW
CryptCreateHash
CryptHashData
CryptGetHashParam
CryptDestroyHash
CryptReleaseContext
LdrGetProcedureAddress
NtAllocateVirtualMemory
NtFreeVirtualMemory
NtReadVirtualMemory
NtWriteVirtualMemory
NtGetContextThread
NtSetContextThread
RtlCreateUserThread
LdrLoadDll
ZwQueueApcThread
NtQueryVirtualMemory
RtlTimeToSecondsSince1970
GetComputerNameW
GetVersionExW
RtlGetVersion
```

Figure 5: Decrypted strings

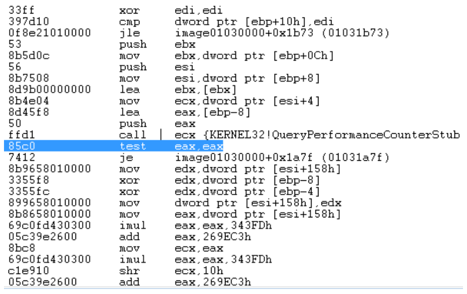
function based on the Microsoft variation LCG algorithm(Figure 7).

Breaking this routine down we can see that ultimately the routine is just generating a random number between 0 and 24 and depending on the outcome of the first loop being even or odd this will be an index into the ascii character set of either the lowercase or the uppercase alphabet. A proof of concept of this in python can be seen in Figure 8.
After copying itself the loader then excutes itself from the new location with its original location as the parameter.



```
SeDebugPrivilege
explorer.exe
svchost.exe
SeShutdownPrivilege
Global\
\\.\pipe\
uzgn53wmy
ys43h26gt
t1ry615nr
ntdll.dll
Roaming
Local
temp
%s  %d  %d
%08x%08x%08x%08x
```

Figure 6: Decrypted unicode strings



```
33ff            xor     edi,edi
397d10          cmp     dword ptr [ebp+10h],edi
0f8e21010000    jle     image01030000+0x1b73 (01031b73)
53              push    ebx
8b5d0c          mov     ebx,dword ptr [ebp+0Ch]
56              push    esi
8b7508          mov     esi,dword ptr [ebp+8]
8d9b00000000    lea     ebx,[ebx]
8b4e04          mov     ecx,dword ptr [esi+4]
8d45f8          lea     eax,[ebp-8]
50              push    eax
ffd1            call  | ecx {KERNEL32!QueryPerformanceCounterStub
85c0            test    eax,eax
7412            je      image01030000+0x1a7f (01031a7f)
8b9658010000    mov     edx,dword ptr [esi+158h]
3355f8          xor     edx,dword ptr [ebp-8]
3355fc          xor     edx,dword ptr [ebp-4]
899658010000    mov     dword ptr [esi+158h],edx
8b8658010000    mov     eax,dword ptr [esi+158h]
69c0fd430300    imul    eax,eax,343FDh
05c39e2600      add     eax,269EC3h
8bc8            mov     ecx,eax
69c0fd430300    imul    eax,eax,343FDh
c1e910          shr     ecx,10h
05c39e2600      add     eax,269EC3h
```

Figure 7: Pseudo-Random filename generate function

2.3 Mutex Generation

After starting from either `%APPDATA%\local` or `C:\Windows` the loader goes through the same checks and then checks if it temp is in it's path. If not it starts building out it's mutex value. The mutex is based on the following information

1. GetCompuerNameW
2. RtlGetVersion - Build Number

Passes the computer name, 0x31 and the machines build number to a wsprintfW call producing the following unicode string: $< computername > 49 < buildnumber >$.

A SHA1 hash is then performed on the unicode string but it only takes the first 16 bytes of the output and then passes it to wsprintfW with the format string "%08x%08x%08x%08x". This string is appended to `Global\` and checked using OpenMutexW(Figure 9).

2.4 Rsrc Decoding and Injection

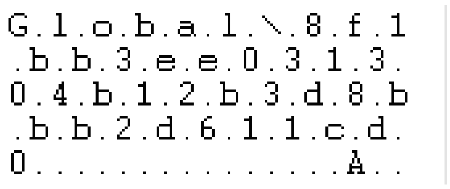
Statically looking at the loader we can see 3 resource sections(Figure 10), first it loads the smaller of the three resource sections which is 256 bytes in length, the next resource section loaded depends on if the system is 32 bit or 64 bit.

```
temp = 0
val = c_int64()

resp = ""
for i in range(15):
        for j in range(2):
                windll.Kernel32.QueryPerformanceCounter(byref(val))
                perf = val.value

                temp ^= perf>>32
                temp ^= perf & 0xFFFFFFFF

                temp *= int('343fd',16)
                temp = temp & 0xFFFFFFFF

                temp = temp + int('269ec3',16)
                temp2 = temp
                temp = (temp * int('343fd',16)) & 0xFFFFFFFF
                temp2 >>= 16
                temp += int('269ec3',16)
                if j == 0:
                        if temp2 % 2 == 1:
                                even = True
                        else:
                                even = False

        temp = temp & 0xFFFF0000
        temp = temp | temp2
        remain = temp % 25

        if even:
                remain += int('61',16)
        else:
                remain += int('41',16)

        resp += chr(remain)

print(resp)
```

Figure 8: Pseudo-Random filename generation

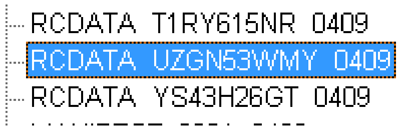
Depending on the outcome of that check the loader loads in one of the remaining resource sections.

After loading the proper resource the loader will find the appropriate process



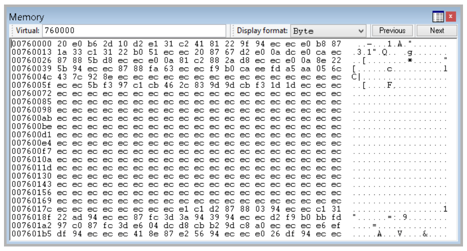
```
G.l.o.b.a.l.\.8.f.1
.b.b.3.e.e.0.3.1.3.
0.4.b.1.2.b.3.d.8.b
.b.b.2.d.6.1.1.c.d.
0...............A..
```

Figure 9: Mutex



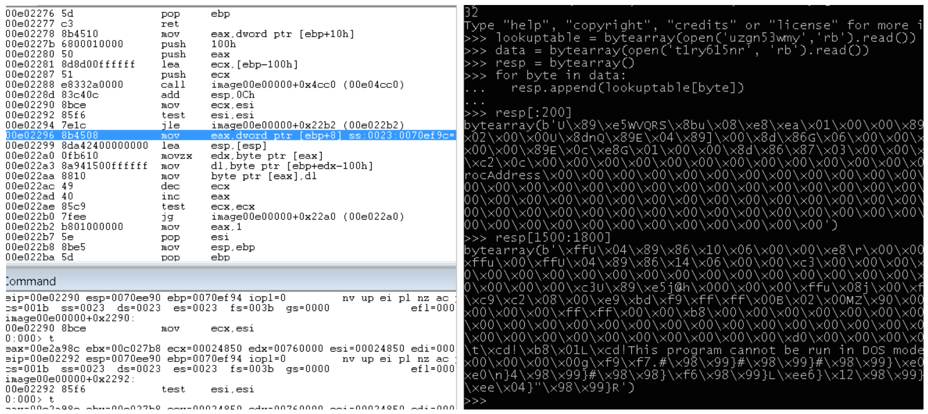
```
RCDATA T1RY615NR 0409
RCDATA UZGN53WMY 0409
RCDATA YS43H26GT 0409
```

Figure 10: Resource Sections



```
00760000 20 e0 b6 2d 10 d2 e1 31 c2 41 81 22 9f 94 ec ec e0 b8 87   ..-...1.A."........
00760013 1a 33 c1 31 22 b0 51 ec ec 20 87 67 d2 e0 0a dc e0 ca ec   .3.1".Q...g........
00760026 87 88 5b d8 ec ec e0 0a 81 c2 88 2a d8 ec ec e0 0a 8e 22   ..[........*......"
00760039 5b 94 ec ec 87 88 fa 63 ec ec f9 b0 ca ee fd a5 aa 05 6c   [......c..........l
0076004c 43 7c 92 8e ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec   C|.................
0076005f ec ec 5b f3 97 c1 cb 46 2c 83 9d 9d cb f3 1d 1d ec ec ec   ..[....F,..........
00760072 ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec   ...................
00760085 ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec   ...................
00760098 ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec   ...................
007600ab ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec   ...................
007600be ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec   ...................
007600d1 ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec   ...................
007600e4 ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec   ...................
007600f7 ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec   ...................
0076010a ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec   ...................
0076011d ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec   ...................
00760130 ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec   ...................
00760143 ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec   ...................
00760156 ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec   ...................
00760169 ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec ec   ...................
0076017c ec ec ec ec ec ec ec ec e1 c1 d2 87 88 03 94 ec ec c1 31   ..................1
0076018f 22 ad 94 ec ec 87 fc 3d 3a 94 39 94 ec ec d2 f9 b0 bb fd   "......=:.9........
007601a2 97 c0 87 fc 3d e6 04 dc d8 cb b2 9d c8 a0 ec ec ec e6 ef   ....=..............
007601b5 df 94 ec ec ec 41 8e 87 e2 56 94 ec ec e0 26 df 94 ec ec   .....A...V...&....
```

Figure 11: Large Resource



```
00e02276 5d              pop     ebp
00e02277 c3              ret
00e02278 8b4510          mov     eax,dword ptr [ebp+10h]
00e0227b 6800010000      push    100h
00e02280 50              push    eax
00e02281 8d8d00ffffff    lea     ecx,[ebp-100h]
00e02287 51              push    ecx
00e02288 e8332a0000      call    image00e00000+0x4cc0 (00e04cc0)
00e0228d 83c40c          add     esp,0Ch
00e02290 8bce            mov     ecx,esi
00e02292 85f6            test    esi,esi
00e02294 7e1c            jle     image00e00000+0x22b2 (00e022b2)
00e02296 8b4508          mov     eax,dword ptr [ebp+8] ss:0023:0070ef9c=
00e02299 8da42400000000  lea     esp,[esp]
00e022a0 0fb610          movzx   edx,byte ptr [eax]
00e022a3 8a941500ffffff  mov     dl,byte ptr [ebp+edx-100h]
00e022aa 8810            mov     byte ptr [eax],dl
00e022ac 49              dec     ecx
00e022ad 40              inc     eax
00e022ae 85c9            test    ecx,ecx
00e022b0 7fee            jg      image00e00000+0x22a0 (00e022a0)
00e022b2 b801000000      mov     eax,1
00e022b7 5e              pop     esi
00e022b8 8be5            mov     esp,ebp
00e022ba 5d              pop     ebp
```

Command

```
eip=00e02290 esp=0070ee90 ebp=0070ef94 iopl=0         nv up ei pl nz ac
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=000
image00e00000+0x2290:
00e02290 8bce            mov     ecx,esi
0:000> t
eax=00e2a98c ebx=00c027b8 ecx=00024850 edx=00760000 esi=00024850 edi=000
eip=00e02292 esp=0070ee90 ebp=0070ef94 iopl=0         nv up ei pl nz ac
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=000
image00e00000+0x2292:
00e02292 85f6            test    esi,esi
0:000> t
```

```
32
Type "help", "copyright", "credits" or "license" for more i
>>> lookuptable = bytearray(open('uzgn53wmy','rb').read())
>>> data = bytearray(open('t1ry615nr', 'rb').read())
>>> resp = bytearray()
>>> for byte in data:
...   resp.append(lookuptable[byte])
...
>>> resp[:200]
bytearray(b'U\x89\xe5WVQRS\x8bu\x08\xe8\xea\x01\x00\x00\x89
x02\x00\x00U\x8dnQ\x89E\x04\x89]\x00\x8d\x86G\x06\x00\x00\x
x00\x00\x89E\x0c\xe8G\x01\x00\x00\x8d\x86\x87\x03\x00\x00\x
\xc2\x0c\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
rocAddress\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
>>> resp[1500:1800]
bytearray(b'\xffU\x04\x89\x86\x10\x06\x00\x00\xe8\r\x00\x00
xffu\x00\xffU\x04\x89\x86\x14\x06\x00\x00\xc3\x00\x00\x00\x
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
0\x00\x00\x00\xc3U\x89\xe5j@h\x000\x00\x00\xffu\x08j\x00\xf
\xc9\xc2\x08\x00\xe9\xbd\xf9\xff\xff\x00B\x02\x00MZ\x90\x00
x00\x00\x00\xff\xff\x00\x00\xb8\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xd0\x00\x00\x00\x0
\t\xcd!\xb8\x01L\xcd!This program cannot be run in DOS mode
x00\x00\x00\x00g\xf9\xf7.#\x98\x99}#\x98\x99}#\x98\x99}\xe0
xe0\n}4\x98\x99}#\x98\x98}\xf6\x98\x99}L\xee6}\x12\x98\x99}
\xee\x04}"\x98\x99}R')
>>>
```

Figure 12: Resource Section Decode POC

to inject. In the event the loader is running from APPDATA then it will inject explorer.exe, if however the loader is running from the Windows directory then it will inject svchost.exe.

The loader will perform the injection by creating a handle to a empty file mapping object using CreateFileMappingW and attain the base address with MapViewOfFile. The encoded data(Figure 11) is then copied over to this memory section before the loader maps the section into the remote process using ZwMapViewOfSection. Next an APC thread is created using the processes main thread id, this is attained using NtQuerySystemInformation.

The loader calls NtQuerySystemInformation for the SystemProcessInformation option which will pull in a giant linked list of SYSTEM_PROCESS_INFORMATION structures. After enumerating this list to find its target by comparing process ids, the loader will then check if the number of threads is <= 0 and if so it will continue enumerating the list. If number of threads is < 0 however then it will jump 0xDC bytes into the structure which lands you at 4 bytes into the CLIENT_ID structure within the SYSTEM_THREAD_INFORMATION structure which is located at the bottom of the relevant SYSTEM_PROCESS_INFORMATION structure. The loader checks that the threadState is 5 and then reads in the thread id from the CLIENT_ID structure.

After queueing the APC thread the loader will decode the injected code. The decoding is done using the smaller resource section as a lookup table. The two larger resource sections are the 32 bit and 64 bit encoded injects respectively and this can be proven with a simple proof of concept as in Figure 12. In the previous figure we can see the decoded inject appears to be a dll wrapped in shellcode.

3 Conclusions

Sample SHA256: ffd0c9571d4a76618c8a970f71bb17a7b0e3b9e2244704ced368bfe276614e63

References

[1] Hex-Rays Decompiler, http://www.hex-rays.com/products/decompiler/index.shtml.

[2] Python, https://www.python.org/