Dissecting the Dyre Loader

JASON REAVES

November 17, 2015

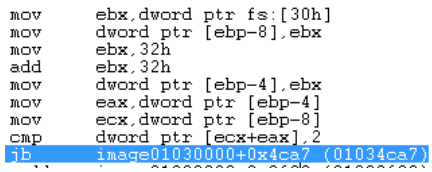

Figure 1: Processor Check

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.

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. A proof of concept example of this can be seen in Figure 4, and decrypting

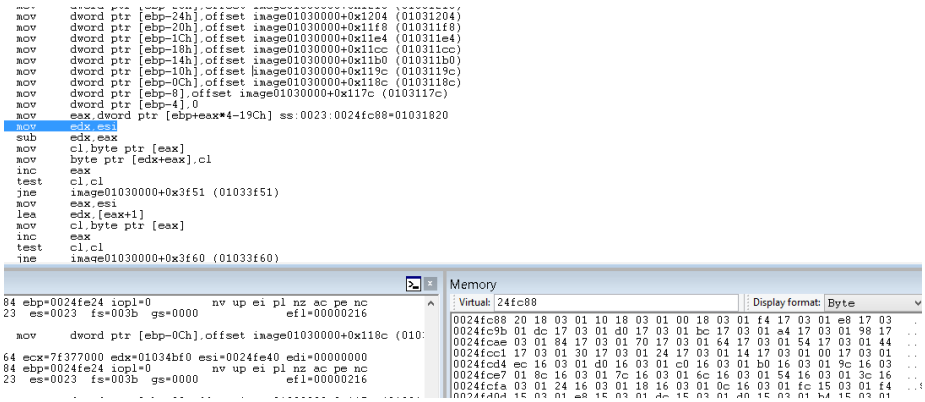

Figure 2: Finding which string to decode

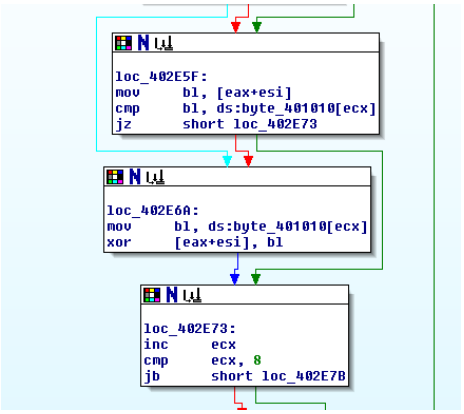

Figure 3: Main string decoding section

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

```
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

Figure 5: Decrypted strings

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.

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

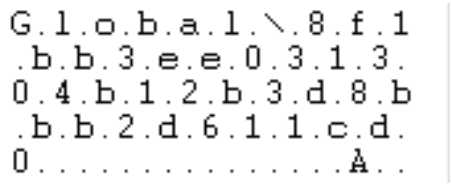

Figure 9: Mutex

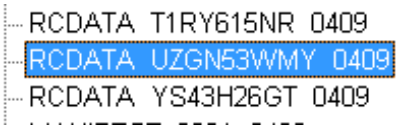

Figure 10: Resource Sections

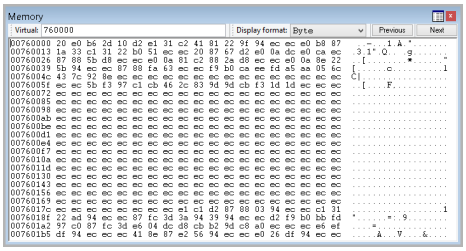

Figure 11: Large Resource

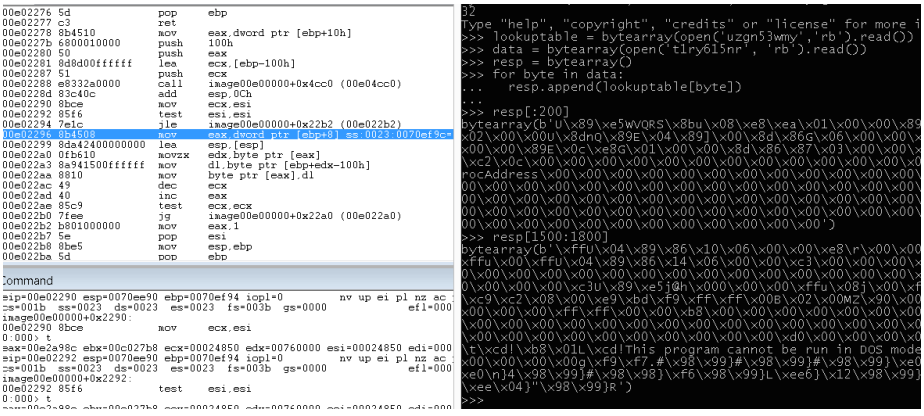

Figure 12: Resource Section Decode POC

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