ARCADIA
Cyber Defense

SPECIAL RESEARCH REPORT

Reverse engineering of the intrusion behavior of malware z13.exe

SNOW Special Research Group
Mickaël Paradis & Benoit Hamelin, Ph.D
Winter 2015

Reverse engineering of the intrusion behavior of malware z13.exe

Mickaël Paradis and Benoit Hamelin, Arc4dia Labs

2015 03 19

Introduction

We report here on a reverse engineering effort to better understand how z13.exe infests a system. From our analysis, this malware bears the signature of a banker trojan. Although we have not yet confirmed its insertion in a web browser to steal banking credentials, we can see that it leverages methods and algorithms seen in Zeus and the most recent version of Tinba (Tiny Banker). In addition, it it is reported by most antivirus software as miscellaneous banker variations. Since the leak of the Zeus source code in 2011, forks of well-known versions of banker trojans have become commonplace.
We first look at the dropper program, which unwraps the malware behaviour from encrypted modules. Second, we show how it transfers its attack code to a hollowed-out child process. Third, we follow its injection into Windows Explorer, using the so-called Heaven's Gate technique. Fourth, we examine the various tricks the malware does to cloak itself, as well as to persist on the infected system.

Malware Identity		
File name	z13.exe	
Observed Location		
Architecture	x86 (supports x64 injections)	
Arc4dia Indicators		
Dropped Files	z13.exe	
Autoruns	Key	HKCU\Software\Microsoft\windows\currentVersion\Run
	Value	C:\%UserDir%\AppData\Roaming\Identities\wnisxpeo.exe lorem ipsum dolor
Architecture	C:\%UserDir%\AppData\Roaming\Identities\wnisxpeo.exe	

Table Lorem ipsum

Entry Point

At first glance, the malware doesn't reveal much with static and dynamic analysis. Only with careful tracing does it reveal all its tricks. It starts as a common MFC program. It sets up a dialog named *chat tool* from a loaded resource segment. It's from the initialization of this dialog that the code Link starts to act like a malware.
It loads in memory a copy of the file on disk and then seeks a value into this memory that indicates the start of an encrypted block. The block is decrypted on the heap as follows:

```
004012DC   B8 0F 82 AE 34          mov eax,34AE820F
004012E1   F7 EE                   imul esi
004012E3   C1 FA 06                sar edx,6
004012E6   8B C2                   mov eax,edx
004012E8   81 C6 6E 02 00 00       add esi,26E
004012EE   C1 E8 1F                shr eax,1F
004012F1   03 D0                   add edx,eax
004012F3   03 D7                   add edx,edi
004012F5   8A 85 48 E8 FF FF       mov al,byte ptr ss:[ebp-17B8]
004012FB   8A 94 0A B3 0C 00 00    mov dl,byte ptr ds:[edx+ecx+CB3]
00401302   80 F2 FB                xor dl,FB
00401305   32 D0                   xor dl,al
00401307   88 13                   mov byte ptr ds:[ebx],dl
00401309   43                      inc ebx
0040130A   81 FE EE 57 1B 00       cmp esi,1B57EE
00401310   0F 8E 25 FF FF FF       jng z13.40123B
```

Block decryption

Once decrypted, the execution jumps into this new code segment, which executes various recon tasks to determine whether the host is already infected. At the end of this code segment, another memory block is decrypted using the same procedure. This second block contains antiVM behaviour and an interesting forking process detailed in the next section. This type of architecture shows modularity in the conception of the malware. The developers make independent block with different features and those blocks, revealed for execution one at a time, determine the malware behaviour.

```
02800A45   6A 14                   push 14
02800A47   8D 85 34 FF FF FF       lea eax,dword ptr ss:[ebp-CC]
02800A4D   57                      push edi
02800A4E   50                      push eax
02800A4F   8D 85 7C C0 FF FF       lea eax,dword ptr ss:[ebp-3F84]
02800A55   50                      push eax
02800A56   E8 00 F9 FF FF          call 280035B                    Decipher function
02800A5B   83 C4 10                add esp,10
02800A5E   6A 40                   push 40
02800A60   68 00 30 00 00          push 3000
02800A65   57                      push edi
02800A66   53                      push ebx
02800A67   FF 95 48 FF FF FF       call dword ptr ss:[ebp-B8]      [ebp-B8]:VirtualAlloc
02800A6D   8B F0                   mov esi,eax                     Move newly allocated space ptr ( 0x3D0000 ) in ESI
02800A6F   8D 85 7C C0 FF FF       lea eax,dword ptr ss:[ebp-3F84]
02800A75   57                      push edi
02800A76   50                      push eax
02800A77   56                      push esi
02800A78   E8 B3 F7 FF FF          call 2800230                    Move data to allocated space
02800A7D   83 C4 0C                add esp,C
02800A80   FF D6                   call esi                        Jump into new code
02800A82   5F                      pop edi
02800A83   5E                      pop esi
02800A84   33 C0                   xor eax,eax
02800A86   5B                      pop ebx
02800A87   C9                      leave
02800A88   C3                      ret
02800A89   CC                      int3
```

```
003D0000   00004000                                PRV  ERW--  ERW--
00400000   00001000   z13.exe                      IMG  -R---  ERWC-
00401000   00002000    ".text"                     IMG  ER---  ERWC-
00403000   00001000    ".rdata"                    IMG  -R---  ERWC-
00404000   00001000    ".data"                     IMG  -RW--  ERWC-
00405000   00001000    ".idata"                    IMG  -RW--  ERWC-
00406000   0001E000    ".rsrc"                     IMG  -R---  ERWC-
00424000   00001000    ".reloc"                    IMG  -R---  ERWC-
```

Code modularity

The imports for each code segment are found by first pushing the names of all routines it needs on the stack. Then, the string *kernel32.dll* is retrieved from the process environment block (PEB). The module is found among those loaded, so the code fetches *LoadLibrary* and *GetProcAddress* from its the export table. Those two functions are then used to resolve all other function names that were previously pushed on the stack.

```
02800740   C6 85 7C FF FF FF 2E   mov byte ptr ss:[ebp-84],2E
02800747   C6 85 7D FF FF FF 65   mov byte ptr ss:[ebp-83],65
0280074E   C6 85 7E FF FF FF 78   mov byte ptr ss:[ebp-82],78
02800755   C6 85 7F FF FF FF 65   mov byte ptr ss:[ebp-81],65
0280075C   C6 45 80 2E            mov byte ptr ss:[ebp-80],2E
02800760   C6 45 81 5C            mov byte ptr ss:[ebp-7F],5C
02800764   88 5D 82               mov byte ptr ss:[ebp-7E],bl
02800767   E8 EC FA FF FF         call 2800258                          Load GetProcAddress and LoadLibrary
0280076C   59                     pop ecx
0280076D   8D 85 60 FF FF FF      lea eax,dword ptr ss:[ebp-A0]
02800773   59                     pop ecx
02800774   50                     push eax
02800775   8D 45 E0               lea eax,dword ptr ss:[ebp-20]
02800778   50                     push eax
02800779   FF 55 FC               call dword ptr ss:[ebp-4]             [ebp-4]:LoadLibraryA
0280077C   50                     push eax
0280077D   FF 55 F8               call dword ptr ss:[ebp-8]             [ebp-8]:GetProcAddress
02800780   89 85 28 FF FF FF      mov dword ptr ss:[ebp-D8],eax         [ebp-D8]:eFileNameW
02800786   8D 45 B8               lea eax,dword ptr ss:[ebp-48]
02800789   50                     push eax
0280078A   8D 45 E0               lea eax,dword ptr ss:[ebp-20]
0280078D   50                     push eax
0280078E   FF 55 FC               call dword ptr ss:[ebp-4]             [ebp-4]:LoadLibraryA
02800791   50                     push eax
02800792   FF 55 F8               call dword ptr ss:[ebp-8]             [ebp-8]:GetProcAddress
02800795   89 45 F0               mov dword ptr ss:[ebp-10],eax
02800798   8D 45 AC               lea eax,dword ptr ss:[ebp-54]
0280079B   50                     push eax
0280079C   8D 45 E0               lea eax,dword ptr ss:[ebp-20]
0280079F   50                     push eax
028007A0   FF 55 FC               call dword ptr ss:[ebp-4]             [ebp-4]:LoadLibraryA
028007A3   50                     push eax
028007A4   FF 55 F8               call dword ptr ss:[ebp-8]             [ebp-8]:GetProcAddress
```

Import resolving

Malicious behaviour transfer

The malware is using a wellknown but interesting technique called process hollowing, which consists in rewriting the memory of another process without touching the PEB. This technique is usually used to cloak a malicious process into a legitimate process. For example, “C:\malware.exe” could hide into “C:\windows\notepad.exe”, but in this case, the hollowed process is the same as the original one. It is therefore not used for cloaking but as a way to slow down the reversing.

```
push eax
push ebx
call dword ptr ss:[ebp-588]                [ebp-588]:CreateProcessW
test eax,eax
je 39233D
lea eax,dword ptr ss:[ebp-10C0]
mov dword ptr ss:[ebp-10C0],10007
push eax
push dword ptr ss:[ebp-28]
call dword ptr ss:[ebp-558]                [ebp-558]:GetThreadContext
push ebx
lea eax,dword ptr ss:[ebp-1C]              [ebp-1C]:RegQueryValueExA
push 4
push eax
push dword ptr ss:[ebp-1010]
push dword ptr ss:[ebp-2C]
call dword ptr ss:[ebp-574]                [ebp-574]:ZwReadVirtualMemory
push dword ptr ss:[ebp-1C]                 [ebp-1C]:RegQueryValueExA
push dword ptr ss:[ebp-2C]
call dword ptr ss:[ebp-80]                 [ebp-80]:NtUnmapViewOfSection
push dword ptr ss:[ebp-1C]                 [ebp-1C]:RegQueryValueExA
push dword ptr ss:[ebp-2C]
call dword ptr ss:[ebp-80]                 [ebp-80]:NtUnmapViewOfSection
push dword ptr ss:[ebp-1C]                 [ebp-1C]:RegQueryValueExA
push dword ptr ss:[ebp-2C]
call dword ptr ss:[ebp-80]                 [ebp-80]:NtUnmapViewOfSection
push dword ptr ds:[edi+34]
push dword ptr ss:[ebp-2C]
call dword ptr ss:[ebp-80]                 [ebp-80]:NtUnmapViewOfSection
push dword ptr ds:[edi+34]
push dword ptr ss:[ebp-2C]
call dword ptr ss:[ebp-80]                 [ebp-80]:NtUnmapViewOfSection
push 40
push 3000
push dword ptr ds:[edi+50]
push dword ptr ds:[edi+34]
push dword ptr ss:[ebp-2C]
call dword ptr ss:[ebp-570]                [ebp-570]:VirtualAllocEx
push ebx
```

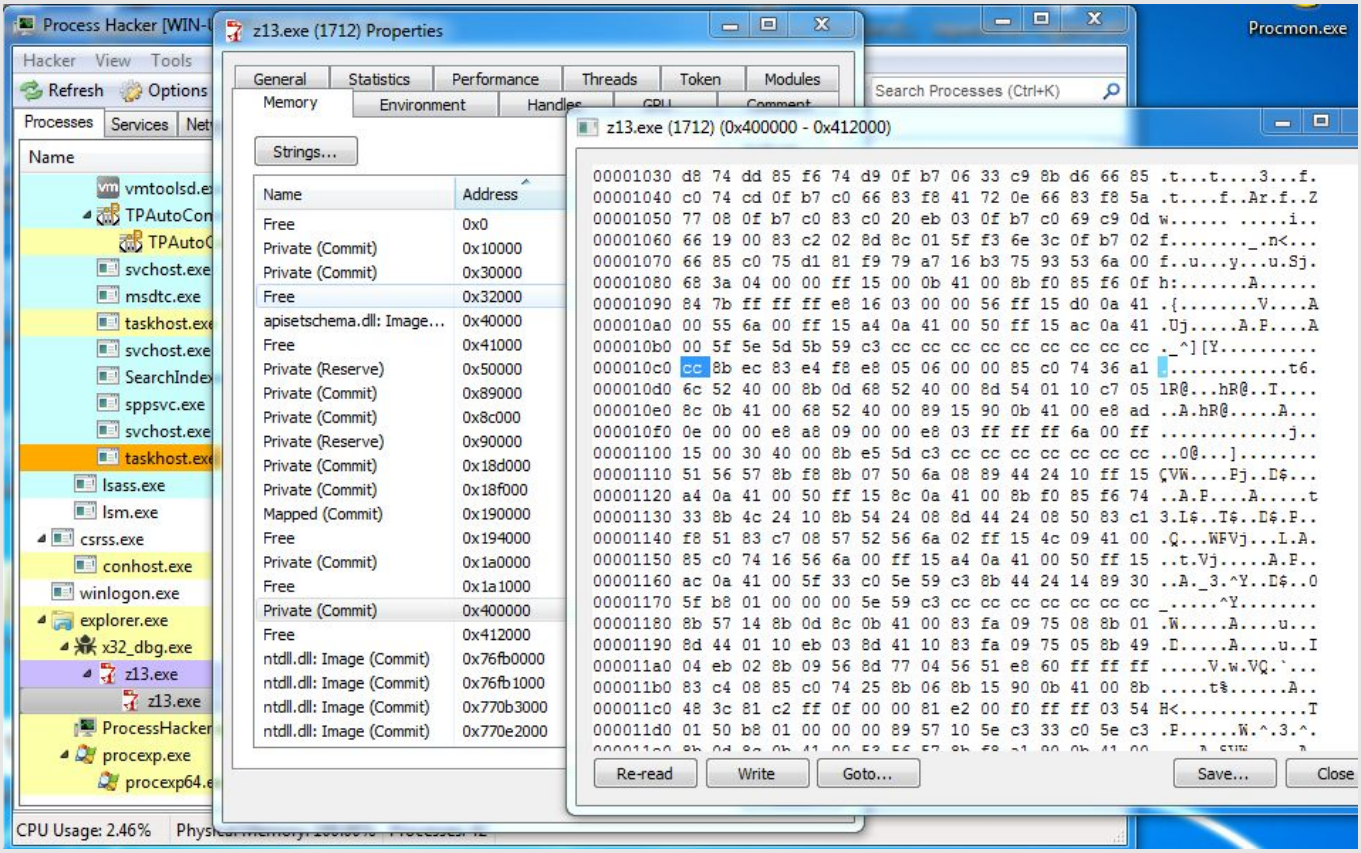
```
push 4
push eax
mov eax,dword ptr ss:[ebp-101C]
add eax,8
push eax
push dword ptr ss:[ebp-2C]
call dword ptr ss:[ebp-4AC]          [ebp-4AC]:WriteProcessMemory
mov eax,dword ptr ds:[edi+28]
add eax,dword ptr ss:[ebp-48]        [ebp-48]:RegOpenKeyExA
mov dword ptr ss:[ebp-1008],eax
mov dword ptr ss:[ebp-1010],eax
lea eax,dword ptr ss:[ebp-10C0]
push eax
push dword ptr ss:[ebp-28]
call dword ptr ss:[ebp-55C]          [ebp-55C]:SetThreadContext
push dword ptr ss:[ebp-28]
call dword ptr ss:[ebp-560]          [ebp-560]:ResumeThread
cmp dword ptr ss:[ebp+8],7
jnz 392347
```

Process hollowing implementation

The implementation of process hollowing starts by creating a suspended target process. The original memory sections of the target process are first unmapped with *NtUnmapViewOfSection* . The new sections are then written into the hollowedout process. The suspended process is now ready to resume except for one detail: the thread EP (entry point) is not valid anymore. Hence the parent process gets the main thread context with *GetThreadContext* , corrects the EP at offset 0xB0 and sets back the thread context with SetThreadContext . The hollowedout process is then resumed with new program code.

At this point, the malicious computation has transferred into the child process. In order to continue our trace study, we must attach to the child process. This can be done by manually placing a breakpoint on the new EP with a memory patching tool like ProcessHacker. The EP address is found at offset 0xB0 in the thread context structure just before the *SetThreadContext* function.

Address	Hex	ASCII
00177DDC	07 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00177DEC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00177DFC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00177E0C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00177E1C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00177E2C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00177E3C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00177E4C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00177E5C	00 00 00 00 00 00 00 00 00 00 00 00 2B 00 00 00	............+...
00177E6C	53 00 00 00 2B 00 00 00 2B 00 00 00 00 00 00 00	S...+...+.......
00177E7C	00 00 00 00 00 E0 FD 7E 00 00 00 00 00 00 00 00	.....àý~........
00177E8C	C0 10 40 00 00 00 00 00 C0 10 40 00 23 00 00 00	À.@.....À.@.#...
00177E9C	02 02 00 00 F0 FF 18 00 2B 00 00 00 41 50 49 2E	....ðÿ..+...API.
00177EAC	64 6C 6C 00 00 00 00 00 00 00 00 00 00 00 00 00	dll.............
00177EBC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....



Hollowed process EP

With the breakpoint set in the child process, we set our debugger up as JIT and resume execution of the parent process. The latter will resume the hollowedout process, so the JIT debugger will break on the EP.
The main task of the child process is to inject a remote thread into the Windows Explorer. To prepare for this, it simply asserts debugging privileges, so it may attach as a debugger to all application running as the same user.

Injection in Windows Explorer

The malware finds its injection target by listing the system processes with the function *NtQuerySystemInformation*, called with the enum SYSTEM_INFORMATION_CLASS set to 0x5 (*SystemProcessInformation*) . It then performs the same hash comparison procedure used to resolve the imports in order to find the SYSTEM_PROCESS_INFORMATION structure related to explorer.exe. It gets from that structure the PID of explorer.exe, of which it retrieves the handle using *OpenProcess*.

```
0040105A   0F B7 C0              movzx eax,ax
0040105D   69 C9 0D 66 19 00     imul ecx,ecx,19660D
00401063   83 C2 02              add edx,2
00401066   8D 8C 01 5F F3 6E 3C  lea ecx,dword ptr ds:[ecx+eax+3C6EF35F]
0040106D   0F B7 02              movzx eax,word ptr ds:[edx]
00401070   66 85 C0              test ax,ax
00401073   75 D1                 jnz z13.401046
00401075   81 F9 79 A7 16 B3     cmp ecx,B316A779                Compare with explorer.exe
0040107B   75 93                 jnz z13.401010
0040107D   53                    push ebx
0040107E   6A 00                 push 0
00401080   68 3A 04 00 00        push 43A
00401085   FF 15 00 0B 41 00     call dword ptr ds:[<&OpenProcess>]
```

Explorer.exe hash comparison

When ready to inject, the malware tests the remote process architecture with *isWow64Process*. The injection occurs either ways, but with the difference that it passes through the Heaven's gate for a 64bits process.
The Heaven's gate is a simple far jump with the code selector register (CS) set to 0x33, indicating x64 instructions. Its a simple CPU switch between 32bit and 64bit mode. In *sysWoW64*, this switch is implemented in the function *wow64cpu!X86SwitchTo64BitMode* FS:[0xC0]. In order to keep control over the return address, the injector implemented a widget with a call to offset 0x0 and setting up the return address 0x5 higher on the stack, yielding a return following immediately this fake procedure, into the rest of the injected code..

```
00402143   6A 33              push 33                          CS
00402145   E8 00 00 00 00     call z13.40214A
0040214A   83 04 24 05        add dword ptr ss:[esp],5         Return address
0040214E   CB                 retf                             Far return into x64 instructions
0040214F   B8 60 00 00 00     mov eax,60                       Start of x64 instructions
00402154   67 65 4C           dec esp
00402157   8B 10              mov edx,dword ptr ds:[eax]
00402159   4D                 dec ebp
0040215A   8B 52 18           mov edx,dword ptr ds:[edx+18]
0040215D   4D                 dec ebp
0040215E   8B 52 10           mov edx,dword ptr ds:[edx+10]    [edx+10]:\t_'B
00402161   49                 dec ecx
00402162   8B 42 30           mov eax,dword ptr ds:[edx+30]
00402165   48                 dec eax
00402166   85 C0              test eax,eax
00402168   74 43              je z13.4021AD
0040216A   4D                 dec ebp
```

Heaven's gate entrance

Once in 64bits mode, the injection is completed with *CreateRemoteThread*. The inverse mechanism is used to return to x86 mode, setting the CS to 0x23.

```
004021A8   8B 52 34           mov edx,dword ptr ds:[edx+34]
004021AB   EB 04              jmp z13.4021B1
004021AD   33 C0              xor eax,eax
004021AF   33 D2              xor edx,edx
004021B1   E8 00 00 00 00     call z13.4021B6
004021B6   C7 44 24 04 23 00 00 00  mov dword ptr ss:[esp+4],23   CS set back to 0x23
004021BE   83 04 24 0D        add dword ptr ss:[esp],D
004021C2   CB                 retf                             Return into x86 mode
004021C3   5D                 pop ebp
004021C4   C3                 ret
004021C5   CC                 int3
004021C6   CC                 int3
```

Heavens gate exit

The use of this hybrid type of malware is increasingly used by malware authors to adapt to the dual architecture Windows ecosystem.
The main thread injected into explorer.exe is behaving as we would expect from any banker trojan. It starts by hooking *ZwResumeThread*, which is called each time a new process is created by explorer.exe.

```
0000000077001828   0F 05              syscall
000000007700182A   C3                 ret
000000007700182B   0F 1F 44 00 00     nop dword ptr ds:[rax+rax]
0000000077001830   E9 23 E9 FE BF     jmp 36FF0158                 Hook on ZwResumeThread
0000000077001835   CC                 int3
0000000077001836   CC                 int3
0000000077001837   CC                 int3
0000000077001838   0F 05              syscall
000000007700183A   C3                 ret
000000007700183B   0F 1F 44 00 00     nop dword ptr ds:[rax+rax]
0000000077001840   4C 8B D1           mov r10,rcx
```

ntdll.ZwResumeThread hooking

The added hook is straightforward. It injects code into each new process if the access rights allow it. Note that the injection still supports both 32bits and 64bits architectures. The main thread injects also into every currently running process.
The injected thread in explorer.exe also tries to beacon its C&C using the new popular thing in the banker world, DGA (domain generator algorithm), with the difference that it generates IP addresses instead of domain names. The malware owners set up a large range of possible IP addresses for their C&C. The DGA queries thousands of IPs until it receives a valid answer from an active C&C. This method of communication is now very popular, as it is very effective against C&C takedowns.

Persistence and cloaking

The malware is using common sandbox evasion behaviours. It searches indicators specific to virtual environments like vmtoolsd.exe (VMWare), VBoxService.exe (VirtualBox) or SbieDll.dll (Sandboxie). It does so by iterating over the module and process list using *CreateToolhelp32Snapshot* . This is easily worked around with memory patching or a VM setup without virtualisation tools.
Before infecting the host, the mutex “qazwsx” is tested to avoid reinfection. It also attempts to open the file “C:\myapp.exe” with C reateFile. If the returned handle is valid, the process is terminated without infecting the host. This could be a way for the authors to easily avoid infecting the host in the development process. It tests the string “C:\windows\explorer.exe.\” similarly.
Persistence is achieved by dropping a copy of the malware in the AppData folder. The name of this new file is an obfuscated value that terminates in *wnisxpeo.exe* . Here is the deobfuscation algorithm:

```
00401C2F   B8 1F 85 EB 51        mov eax,51EB851F
00401C34   F7 E6                 mul esi
00401C36   8B CA                 mov ecx,edx
00401C38   B8 A3 8B 2E BA        mov eax,BA2E8BA3
00401C3D   F7 E6                 mul esi
00401C3F   8B C2                 mov eax,edx
00401C41   C1 E8 05              shr eax,5
00401C44   33 D2                 xor edx,edx
00401C46   BF 19 00 00 00        mov edi,19
00401C4B   F7 F7                 div edi
00401C4D   8B C6                 mov eax,esi
00401C4F   C1 E8 05              shr eax,5
00401C52   C1 E9 03              shr ecx,3
00401C55   83 C2 61              add edx,61
00401C58   52                    push edx
00401C59   33 D2                 xor edx,edx
00401C5B   F7 F7                 div edi
00401C5D   B8 25 49 92 24        mov eax,24924925
00401C62   83 C2 61              add edx,61
00401C65   52                    push edx
00401C66   F7 E6                 mul esi
00401C68   8B C6                 mov eax,esi
00401C6A   2B C2                 sub eax,edx
00401C6C   D1 E8                 shr eax,1
00401C6E   03 C2                 add eax,edx
00401C70   C1 E8 04              shr eax,4
00401C73   33 D2                 xor edx,edx
00401C75   F7 F7                 div edi
00401C77   8B C1                 mov eax,ecx
00401C79   6B C9 19              imul ecx,ecx,19
00401C7C   83 C2 61              add edx,61
00401C7F   52                    push edx
00401C80   33 D2                 xor edx,edx
00401C82   F7 F7                 div edi
00401C84   B8 C9 42 16 B2        mov eax,B21642C9
00401C89   83 C2 61              add edx,61
00401C8C   52                    push edx
00401C8D   F7 E6                 mul esi
```

Deobfuscation algorithm for wnisxpeo.exe

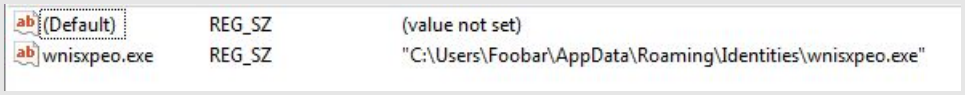
```
Address   Hex                                              ASCII
0018F708  43 00 3A 00 5C 00 55 00 73 00 65 00 72 00 73 00  C.:.\.U.s.e.r.s.
0018F718  5C 00 46 00 6F 00 6F 00 62 00 61 00 72 00 5C 00  \.F.o.o.b.a.r.\.
0018F728  41 00 70 00 70 00 44 00 61 00 74 00 61 00 5C 00  A.p.p.D.a.t.a.\.
0018F738  52 00 6F 00 61 00 6D 00 69 00 6E 00 67 00 5C 00  R.o.a.m.i.n.g.\.
0018F748  49 00 64 00 65 00 6E 00 74 00 69 00 74 00 69 00  I.d.e.n.t.i.t.i.
0018F758  65 00 73 00 5C 00 77 00 6E 00 69 00 73 00 78 00  e.s.\.w.n.i.s.x.
0018F768  70 00 65 00 6F 00 2E 00 65 00 78 00 65 00 00 00  p.e.o...e.x.e...
0018F778  D0 34 35 75 44 00 00 00 20 02 00 00 02 00 00 00  Ð45uD... .......
0018F788  A4 F7 18 00 C9 14 30 75 40 35 35 75 08 00 00 00  ¤÷..É.0u@55u....
0018F798  30 00 00 00 B4 F7 18 00 01 00 00 00 AC F8 18 00  0...´÷......¬ø..
0018F7A8  C8 3E 51 00 00 32 35 75 B7 0E 30 75 DC F7 18 00  È>Q..25u·.0uÜ÷..
0018F7B8  99 49 2F 75 00 00 2E 75 01 00 00 00 00 00 00 00  .I/u...u........
0018F7C8  AC F8 18 00 01 00 00 00 F0 F7 18 00 00 00 00 00  ¬ø......ð÷......
0018F7D8  88 F8 00 01 FC F7 18 00 30 99 1C 77 00 00 2E 75  .ø..ü÷..0..w...u
0018F7E8  01 00 00 00 00 00 00 00 01 00 00 00 AC F8 18 00  ............¬ø..
0018F7F8  C8 3E 51 00 88 F8 18 00 A9 D8 1C 77 E5 49 2F 75  È>Q..ø..@Ø.wåI/u
```

The hiding path

The file time of the dropped file wnisxpeo.exe is copied from svchost.exe. This is a common approach to lower the file suspicion.

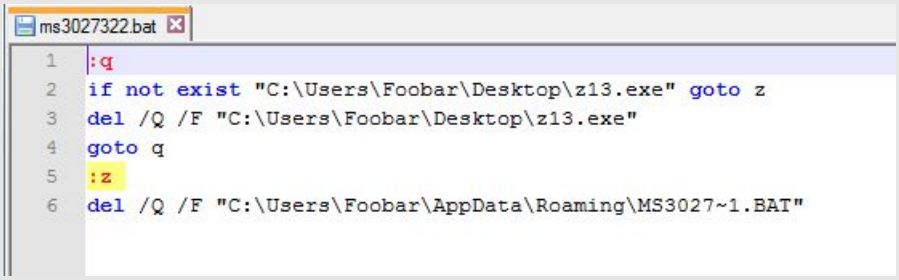
```
6A 03             push 3
6A 00             push 0
6A 01             push 1
68 80 00 00 00    push 80
8D 4C 24 30       lea ecx,dword ptr ss:[esp+30]
51                push ecx                                  ecx:L"C:\\Windows\\system32\\svchost.exe"
FF 15 BC 0A 41 00 call dword ptr ds:[<&CreateFileW>]
8B F0             mov esi,eax
83 FE FF          cmp esi,FFFFFFFF
74 2A             je z13.401D8D
8D 54 24 10       lea edx,dword ptr ss:[esp+10]
52                push edx
6A 00             push 0
8D 44 24 10       lea eax,dword ptr ss:[esp+10]
50                push eax
56                push esi
FF 15 74 0B 41 00 call dword ptr ds:[<&GetFileTime>]      Get svchost.exe file time
85 C0             test eax,eax
74 13             je z13.401D8D
8D 4C 24 10       lea ecx,dword ptr ss:[esp+10]
51                push ecx                                  ecx:L"C:\\Windows\\system32\\svchost.exe"
6A 00             push 0
8D 54 24 10       lea edx,dword ptr ss:[esp+10]
52                push edx
57                push edi
FF 15 78 0B 41 00 call dword ptr ds:[<&SetFileTime>]      Set wnisxpeo.exe file time
56                push esi
FF 15 D0 0A 41 00 call dword ptr ds:[<&CloseHandle>]
57                push edi
```

An autorun registry key is created by the injected explorer.exe process to secure persistence after reboot.



Autorun key

Finally, the dropper is deleted from disk once its task is done. It does so by creating a simple batch file starting by “ms” and followed by a random numerical value. The script is placed in the path “C:\%UserDir%\AppData\Roaming” and executed.



Auto delete script