Stuxnet drivers: detailed analysis

There has passed already a lot of time since the publication of various detailed researches about Stuxnet and its components. All top AV vendors wrote own comprehensive papers, which reveal major information about destructive Stuxnet features. Some information about Stuxnet rootkits were published by Kaspersky here, Symantec here, ESET here. However, the published information is not complete, because each of these documents covers only a specific sample of the rootkit and describes some of its functions. For example, Kaspersky analysis tries to summarize information about known Stuxnet drivers, but it doesn't contain any technical info about it. Another mentioned report from ESET contains information about two Stuxnet drivers, but this is not sufficient for complete summarizing.

First of all, it is need to be clear that from point of view of undocumented Windows kernel exploration, there are no something really interesting in Stuxnet drivers. I mean nothing interesting comparing with such advanced & sophisticated "civilian" rootkits like ZeroAccess or TDL4. These instances can be deeply embedded into a system, bypassing anti-rootkits and deceive low-level disk access tools. In contrast to them, authors of Stuxnet rootkits do not use such deep persistence into a compromised system. This analysis tries to summarize technical information about Stuxnet drivers.

As a starting point of our research, we can take already published information about Stuxnet drivers by Kaspersky. Their analysis Stuxnet/Duqu: The Evolution of Drivers summarizes some information about drivers that have been used by Stuxnet authors in cyber attacks.

Driver 1
File name: MRxCls.sys
SHA256: 817a7f28a0787509c2973ce9ae85a95beb979e30b7b08e64c66d88372aa3da86
File size: 19840 bytes
Signed: No

Timestamp: 2009-01-01 18:53:25
Device object name: \Device\MRxClsDvX
Main purpose: code injection
AV detection ratio: 53/61

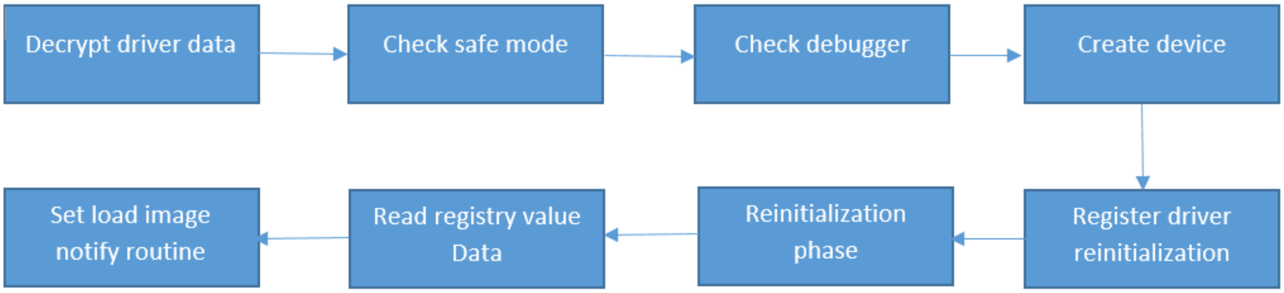
First driver contains sensitive text information such as rootkit device name and path to its service into registry as encrypted data. After starting, the driver performs decryption of this data and we can extract it. Note that name of rootkit service is almost matches its device object name. First dword of decrypted data is also interesting, because it stores some flags, which have an impact on driver behaviour. For example, first bit of this dword restricts the work of rootkit code into Windows safe mode, while second is used as anti-debug trick. If second bit and *ntoskrnl!KdDebuggerEnabled* are active, the driver will not load.

Decrypted rootkit data also stores name of registry value (Data) that is used by the rootkit to determining what files should be injected into processes. So, these decrypted data are stored in the next sequence.

\REGISTRY\MACHINE\SYSTEM\CurrentControlSet\Services\MRxCls
Data
\Device\MRxClsDvX

```
.text:0001041E BE 78 02 00 00                 mov     esi, 278h
.text:00010423 B9 99 3E 01 00                 mov     ecx, offset dwFlag
.text:00010428 E8 15 18 00 00                 call    fnDecryptData   ; ecx->data; esi->size
.text:0001042D 88 1D 98 3E 01+                mov     byte_13E98, bl
.text:00010433
.text:00010433                jCheckFlag:                             ; CODE XREF: start+70↑j
.text:00010433 A1 99 3E 01 00                 mov     eax, dwFlag
.text:00010438 A8 01                          test    al, 1
.text:0001043A 74 10                          jz      short jCheckRemoteDebugger
.text:0001043C A1 B0 23 01 00                 mov     eax, ds:InitSafeBootMode
.text:00010441 39 18                          cmp     [eax], ebx
.text:00010443 74 07                          jz      short jCheckRemoteDebugger
.text:00010445
.text:00010445                jRetWithError:                          ; CODE XREF: start+B2↓j
.text:00010445 B8 01 00 00 C0                 mov     eax, 0C0000001h
.text:0001044A EB 14                          jmp     short loc_10460
.text:0001044C                ; ---------------------------------------------------------------------------
.text:0001044C
.text:0001044C                jCheckRemoteDebugger:                   ; CODE XREF: start+90↑j
.text:0001044C                                                        ; start+99↑j
.text:0001044C A1 99 3E 01 00                 mov     eax, dwFlag
.text:00010451 A8 02                          test    al, 2
.text:00010453 74 09                          jz      short loc_1045E
.text:00010455 A1 AC 23 01 00                 mov     eax, ds:KdDebuggerEnabled
.text:0001045A 38 18                          cmp     [eax], bl
.text:0001045C 75 E7                          jnz     short jRetWithError
```

As driver is registered by Stuxnet with "boot" loading type, it can't perform whole initialization in *DriverEntry*, because neither NT kernel nor file system is ready to perform requests from clients. So, it calls API *IoRegisterDriverReinitialization* and delays own initialization.
After *ReInitialize* rootkit function gets control, it checks Windows NT version and fills some dynamic imports. It also doing some preparatory operations and calls *PsSetLoadImageNotifyRoutine* for registering own handler on load image. This handler will response for code injection into processes. Below you can see scheme of rootkit initialization.



The driver registers following IRP handlers.

- IRP_MJ_CREATE
- IRP_MJ_CLOSE
- IRP_MJ_DEVICE_CONTROL (*fnDispatchIrpMjDeviceControl*)

If you are not familiar with Windows NT drivers development, it is worth to note that any driver that allows to open handles on its device, registers, at least, two IRP handlers: IRP_MJ_CREATE for supporting operations *ZwCreateFile* and IRP_MJ_CLOSE for *ZwClose*. Our driver supports *ZwDeviceIoControl* interface, that's why it registers IRP_MJ_DEVICE_CONTROL handler.

```
.text:00010A04                         fnDispatchIrpMjDeviceControl proc near  ; DATA XREF: DriverEntry+F3↑o
.text:00010A04
.text:00010A04                         var_1C          = dword ptr -1Ch
.text:00010A04                         var_4           = dword ptr -4
.text:00010A04                         IRP             = dword ptr  0Ch
.text:00010A04
.text:00010A04                         esi_IRP = esi
.text:00010A04 6A 0C                                   push    0Ch
.text:00010A06 68 E8 3D 01 00                          push    offset unk_13DE8
.text:00010A0B E8 F0 17 00 00                          call    fnSEH
.text:00010A0B
.text:00010A10 B9 02 00 00 C0                          mov     ecx, 0C0000002h
.text:00010A15 89 4D E4                                mov     [ebp+var_1C], ecx
.text:00010A18 83 65 FC 00                             and     [ebp+var_4], 0
.text:00010A1C 8B 75 0C                                mov     esi_IRP, [ebp+IRP]
.text:00010A1F 8B 46 60                                mov     eax, [esi_IRP+60h] ; IRP->IoStack
.text:00010A22 81 78 0C 00 38+                         cmp     dword ptr [eax+0Ch], 223800h ; IoStack->DeviceIoControl.IoControlCode
.text:00010A29 74 04                                   jz      short jDispatchIOCTL
.text:00010A29
.text:00010A2B 8B C1                                   mov     eax, ecx
.text:00010A2D EB 06                                   jmp     short loc_10A35
.text:00010A2D
.text:00010A2F                         ; ---------------------------------------------------------------------------
.text:00010A2F
.text:00010A2F                         jDispatchIOCTL:                         ; CODE XREF: fnDispatchIrpMjDeviceControl+25↑j
.text:00010A2F 56                                      push    esi_IRP
.text:00010A30 E8 13 10 00 00                          call    fnDispatchIOCTL
.text:00010A30
.text:00010A35
.text:00010A35                         loc_10A35:                              ; CODE XREF: fnDispatchIrpMjDeviceControl+29↑j
.text:00010A35 8B F8                                   mov     edi, eax
.text:00010A37 89 7D E4                                mov     [ebp-1Ch], edi
.text:00010A3A 83 4D FC FF                             or      dword ptr [ebp-4], 0FFFFFFFFh
.text:00010A3E EB 11                                   jmp     short jCheckOnStatusPending ; STATUS_PENDING
```

handler supports pending I/O operation

Handler *fnDispatchIrpMjDeviceControl* serves only for one purpose: to call undocumented Windows NT function *ZwProtectVirtualMemory*. Client should send to driver special IOCTL code 0x223800 for that (*DeviceIoControl*) and provide a special prearranged structure with parameters for API call. The driver uses buffered I/O.

```
.text:00011A7D
.text:00011A7D                 jGetZwProtectVirtualMemoryAddr:         ; CODE XREF: fnDispatchIOCTL+2C↑j
.text:00011A7D 83 65 FC 00                     and     [ebp+var_4], 0
.text:00011A81 56                              push    esi
.text:00011A82 57                              push    edi
.text:00011A83 8D 75 FC                        lea     esi, [ebp+var_4]
.text:00011A86 8D 7D F8                        lea     edi, [ebp+pZwProtectVirtualMemory]
.text:00011A89 E8 94 EE FF FF                  call    fnGetZwProtectVirtualMemoryAddr
.text:00011A89
.text:00011A8E 8B 45 FC                        mov     eax, [ebp+var_4]
.text:00011A91 85 C0                           test    eax, eax
.text:00011A93 5F                              pop     edi
.text:00011A94 5E                              pop     esi
.text:00011A95 75 26                           jnz     short jRet_
.text:00011A95
.text:00011A97 8D 43 20                        lea     eax, [ebx+20h]
.text:00011A9A 50                              push    eax             ; OldProtect
.text:00011A9B FF 30                           push    dword ptr [eax] ; NewProtectWin32
.text:00011A9D 8D 43 18                        lea     eax, [ebx+18h]
.text:00011AA0 50                              push    eax             ; RegionSize
.text:00011AA1 8D 43 10                        lea     eax, [ebx+10h]
.text:00011AA4 50                              push    eax             ; BaseAddress
.text:00011AA5 FF 73 08                        push    dword ptr [ebx+8] ; ProcessHandle
.text:00011AA8 8B 45 F8                        mov     eax, [ebp+pZwProtectVirtualMemory]
.text:00011AAB FF 10                           call    dword ptr [eax] ; ZwProtectVirtualMemory
.text:00011AAD 85 C0                           test    eax, eax
.text:00011AAF 75 0C                           jnz     short jRet_
```

```
NTSTATUS
NtProtectVirtualMemory(
    __in HANDLE ProcessHandle,
    __inout PVOID *BaseAddress,
    __inout PSIZE_T RegionSize,
    __in WIN32_PROTECTION_MASK NewProtectWin32,
    __out PULONG OldProtect
    )

/*++

Routine Description:

    This routine changes the protection on a region of committed pages
    within the virtual address space of the subject process.  Setting
    the protection on a range of pages causes the old protection to be
    replaced by the specified protection value.

    Note if a virtual address is locked in the working set and the
    protection is changed to no access, the page is removed from the
    working set since valid pages can't be no access.

Arguments:

     ProcessHandle - An open handle to a process object.
```

As we know, function *ZwProtectVirtualMemory* is not exported by the Windows kernel and this is another task which authors of MRXCLS.sys have been solved. For example, in case of Windows 2000, they try to find function signature with analysis of executable sections of ntoskrnl image. This signature you can see below.

```
.rdata:00012450                 ; ---------------------------------------------------------------------------
.rdata:00012450
.rdata:00012450                 Win2k_ZwProtectVirtualMemory_Pattern:   ; DATA XREF: fnCheckNtServiceOnAllocateVirtualMemory+11↑r
.rdata:00012450                                                         ; fnCheckNtServiceOnAllocateVirtualMemory+27↑o
.rdata:00012450 B8 77 00 00 00                  mov     eax, 77h        ; SSDT index
.rdata:00012455 8D 54 24 04                     lea     edx, [esp+4]
.rdata:00012459 CD 2E                           int     2Eh             ; DOS 2+ internal - EXECUTE COMMAND
.rdata:00012459                                                         ; DS:SI -> counted CR-terminated command string
.rdata:0001245B C2 14 00                        retn    14h
.rdata:0001245B
.rdata:0001245B                 ; ---------------------------------------------------------------------------
```

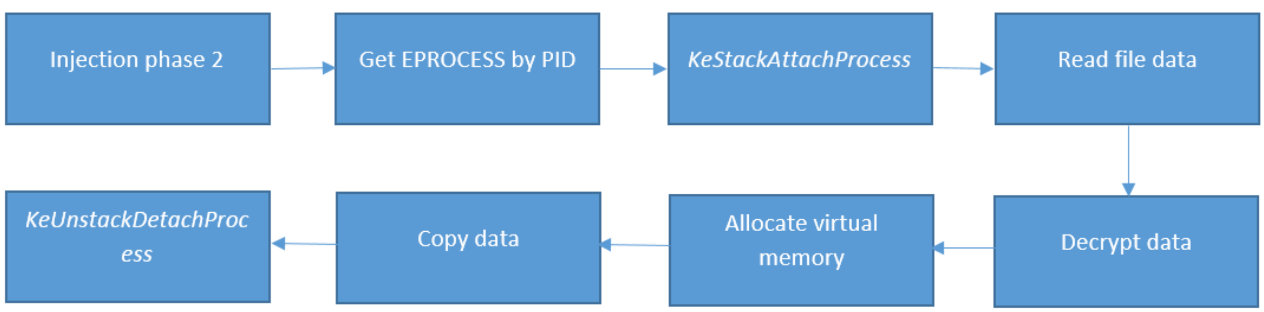
Authors are trying to enumerate all useful ntoskrnl sections and for each of it call special function that performs searching *ZwAllocateVirtualMemory* by signatures on Windows 2000 or little harder on Windows XP.

```
.text:000119DD
.text:000119DD                 jNextNtoskrnlSection:                   ; CODE XREF: sub_11994+90↓j
.text:000119DD 0F B7 C3                        movzx   eax, bx
.text:000119E0 6B C0 28                        imul    eax, 28h
.text:000119E3 8D 14 30                        lea     edx, [eax+esi]
.text:000119E6 E8 43 FD FF FF                  call    fnCheckSectionByFlagsame
.text:000119E6
.text:000119EB 84 C0                           test    al, al
.text:000119ED 74 31                           jz      short jNextIteration
.text:000119ED
.text:000119EF 8B 42 08                        mov     eax, [edx+8]
.text:000119F2 8B 4A 10                        mov     ecx, [edx+10h]
.text:000119F5 3B C1                           cmp     eax, ecx
.text:000119F7 73 02                           jnb     short loc_119FB
.text:000119F7
.text:000119F9 8B C8                           mov     ecx, eax
.text:000119F9
.text:000119FB
.text:000119FB                 loc_119FB:                              ; CODE XREF: sub_11994+63↑j
.text:000119FB 8B 42 0C                        mov     eax, [edx+0Ch]
.text:000119FE 03 45 F4                        add     eax, [ebp+NTKernelBaseAddress]
.text:00011A01 8D 55 FF                        lea     edx, [ebp-1]
.text:00011A04 52                              push    edx
.text:00011A05 8D 55 F8                        lea     edx, [ebp+var_8]
.text:00011A08 52                              push    edx
.text:00011A09 03 C8                           add     ecx, eax
.text:00011A0B 51                              push    ecx
.text:00011A0C 50                              push    eax
.text:00011A0D 8D 45 D8                        lea     eax, [ebp+var_28]
.text:00011A10 50                              push    eax
.text:00011A11 C6 45 FF 01                     mov     [ebp+var_1], 1
.text:00011A15 E8 6A FE FF FF                  call    fnFindZwAllocateVirtualMemory
.text:00011A15
```

Main purpose of this Stuxnet rootkit is code injection. As we can see from its code, the driver tries to read configuration data of injection either from registry parameter *Data*, either from file, if its name is present into malware sample. In analyzed sample, name of configuration file is absent. Injection configuration data is prepared by user mode part of malware. Injection mechanism was perfectly described by ESET in their paper. The driver performs injection into process in two phases: first phase is preparatory and second is major.

```
.text:000112B7 8D 75 D8                    lea     esi, [ebp+var_28]
.text:000112BA 89 5D FC                    mov     [ebp+var_4], ebx
.text:000112BD E8 7A 0A 00 00              call    fnLookupProcessAndAttachToIt
.text:000112BD
.text:000112C2 39 5D FC                    cmp     [ebp+var_4], ebx
.text:000112C5 75 14                       jnz     short loc_112DB
.text:000112C5
.text:000112C7 FF 75 14                    push    [ebp+arg_C]
.text:000112CA 8B 4D 18                    mov     ecx, [ebp+arg_10]
.text:000112CD FF 75 0C                    push    [ebp+arg_4]
.text:000112D0 8B 55 10                    mov     edx, [ebp+arg_8]
.text:000112D3 FF 75 08                    push    [ebp+arg_0]
.text:000112D6 E8 95 FF FF FF              call    fnReadConfigFileAllocateMemAndWriteData_
.text:000112D6
.text:000112DB
.text:000112DB             loc_112DB:                              ; CODE XREF: fnInjectionPhase2+1F↑j
.text:000112DB 38 5D D8                    cmp     byte ptr [ebp+var_28], bl
.text:000112DE 74 0D                       jz      short loc_112ED
.text:000112DE
.text:000112E0 8D 45 E4                    lea     eax, [ebp+var_1C]
.text:000112E3 50                          push    eax
.text:000112E4 88 5D D8                    mov     byte ptr [ebp+var_28], bl
.text:000112E7 FF 15 F0 23 01+             call    ds:KeUnstackDetachProcess
.text:000112E7 00
```

On second phase it tries to read content of file, decrypts it and injects it into process address space. File names for injection are stored into configuration file or registry parameter *Data*.



As we can see from the code analysis, authors have developed rootkit for injection malicious code into processes. Data for injection is prepared by Stuxnet user mode code. The driver registers handler for image load notify and performs injection into two phases. It also supports one IOCTL command for changing protection for virtual memory pages of process with help of *ZwProtectVirtualMemory*. For finding this unexported and undocumented function into ntoskrnl, it uses raw bytes search based on special signatures.

Driver 2

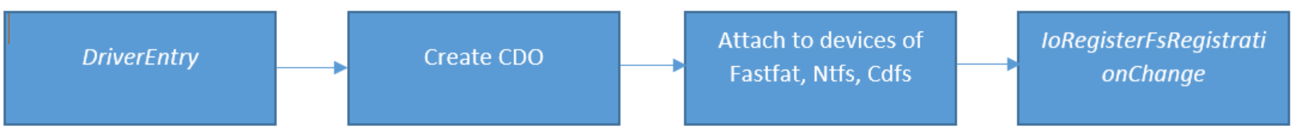
File name: Mrxnet.sys
SHA256: 0d8c2bcb575378f6a88d17b5f6ce70e794a264cdc8556c8e812f0b5f9c709198
File size: 17400 bytes
Signed: Yes
Timestamp: 2010-01-25 14:39:24
Device object name: none
Main purpose: malicious files hiding

AV detection ratio: 51/61

Unlike first driver MRXCLS.sys, authors of Mrxnet.sys don't perform anti-analysis checks in the start function of driver. Mrxnet.sys is a FS filter driver that controls some file operations. The rootkit tries to hide some file types by controlling IRP_MJ_DIRECTORY_CONTROL request and removes information about it from buffer.

```
.text:00010765                jNextHandler:                ; CODE XREF: DriverEntry-188B↓j
.text:00010765 8B 0D 98 1F 01+            mov     ecx, DriverObject
.text:0001076B C7 04 08 86 04+            mov     dword ptr [eax+ecx], offset fnIrpMjHandlerCallNextDriver
.text:00010772 83 C0 04                   add     eax, 4
.text:00010775 3D A4 00 00 00             cmp     eax, 0A4h
.text:0001077A 7E E9                      jle     short jNextHandler
.text:0001077A
.text:0001077C A1 98 1F 01 00             mov     eax, DriverObject
.text:00010781 C7 40 6C 96 04+            mov     dword ptr [eax+6Ch], offset fnIrpMjHandlerFileSystemControl
.text:00010788 A1 98 1F 01 00             mov     eax, DriverObject
.text:0001078D C7 40 68 EC 04+            mov     dword ptr [eax+68h], offset fnIrpMjHandlerDirectoryControl
.text:00010794 E8 A9 FD FF FF             call    fnInitFastIoDispatch
.text:00010794
.text:00010799 E8 20 FF FF FF             call    fnAttachToExistingDevsOfFastfatNtfsCdfs
.text:00010799
.text:0001079E 68 EC 09 01 00             push    offset fnDriverFsNotification
.text:000107A3 FF 35 98 1F 01+            push    DriverObject
.text:000107A9 FF 15 08 1C 01+            call    ds:IoRegisterFsRegistrationChange
.text:000107AF 8B F0                      mov     esi, eax
.text:000107B1 85 F6                      test    esi, esi
.text:000107B3 7D 1D                      jge     short loc_107D2
.text:000107B3
```

The rootkit initialization steps.



As we can see from code, the driver plays with two types of devices: firstly its own CDO (Control Device Object) that represents FS filter and secondly devices that were created to filter files related operations on specific volumes. In case of CDO, the rootkit dispatches widely known request IRP_MJ_FILE_SYSTEM_CONTROL and operation IRP_MN_MOUNT_VOLUME. This operation is used by Windows kernel in case of mounting new volume into a system. After got this request, the rootkit creates new device object, registers completion routine and attaches device to newly mounted device. This method allows for driver to monitor appearance in a system new volumes, for example, volume of removable drive.

```
.text:00010496                 fnIrpMjHandlerFileSystemControl proc near
.text:00010496                                                         ; DATA XREF: DriverEntry-1884↓o
.text:00010496 6A 08                           push    8
.text:00010498 68 D8 1D 01 00                  push    offset unk_11DD8
.text:0001049D E8 22 13 00 00                  call    fnSEH
.text:0001049D
.text:000104A2 83 65 FC 00                     and     dword ptr [ebp-4], 0
.text:000104A6 8B 4D 0C                        mov     ecx, [ebp+0Ch]
.text:000104A9 8B 41 60                        mov     eax, [ecx+60h]
.text:000104AC 51                              push    ecx
.text:000104AD FF 75 08                        push    dword ptr [ebp+8]
.text:000104B0 80 78 01 01                     cmp     byte ptr [eax+1], 1 ; IoStack->MinorFunction -- IRP_MN_MOUNT_VOLUME
.text:000104B4 75 0E                           jnz     short jCallNextDriver
.text:000104B4
.text:000104B6 E8 4B 04 00 00                  call    fnCreateDeviceAndAttachToDeviceStack
.text:000104B6
.text:000104BB
.text:000104BB                 jRet_:                                  ; CODE XREF: fnIrpMjHandlerFileSystemControl+33↓j
.text:000104BB C7 45 FC FE FF+                 mov     dword ptr [ebp-4], 0FFFFFFFEh
.text:000104C2 EB 1A                           jmp     short jRet
.text:000104C2
.text:000104C4                 ; ---------------------------------------------------------------------------
.text:000104C4
.text:000104C4                 jCallNextDriver:                        ; CODE XREF: fnIrpMjHandlerFileSystemControl+1E↑j
.text:000104C4 E8 7B 06 00 00                  call    fnCallNextDriver
.text:000104C4
.text:000104C9 EB F0                           jmp     short jRet_
.text:000104C9
.text:000104C9                 fnIrpMjHandlerFileSystemControl endp
```

As you can see from the picture above, the driver also calls *IoRegisterFsRegistrationChange* I/O manager API for registering its handler that Windows kernel will call each time, when new file system driver CDO is registered into a system. In this handler, the driver creates new device and attaches it to passed CDO or removes device in case of file system driver deletion.

Major purpose of Mrxnet.sys driver is hiding Stuxnet malicious files. Windows kernel provides *ZwQueryDirectoryFile* API for requesting information about files in directory. This API function calls driver handler of IRP_MJ_DIRECTORY_CONTROL operation. So, the rootkit registers own IRP_MJ_DIRECTORY_CONTROL handler and sets completion routine when such request is passed through handler. In this completion routine it analyzes buffer with data and checks file names in it. It erases from buffer files with extension .LNK and .TMP. It also imposes additional restrictions on hiding. For example, in case of .LNK file, its size should be equal 0x104B.

```
.text:00011708 6A 04                          push    4
.text:0001170A 8D 44 73 F8                    lea     eax, [ebx+esi*2-8]
.text:0001170E 50                             push    eax                               check .LNK
.text:0001170F B8 98 1B 01 00                 mov     eax, offset a_lnk ; ".LNK"
.text:00011714 E8 C1 FD FF FF                 call    fnStrCmpi                         extension
.text:00011714
.text:00011719 84 C0                          test    al, al
.text:0001171B 75 12                          jnz     short jRemoveInfoFromBuf
.text:0001171B
.text:0001171D
.text:0001171D                  loc_1171D:                                ; CODE XREF: sub_11688+70↑j
.text:0001171D                                                            ; sub_11688+79↑j ...
.text:0001171D FF 75 F4                       push    [ebp+var_C]
.text:00011720 FF 75 F0                       push    [ebp+var_10]
.text:00011723 56                             push    esi
.text:00011724 8B F3                          mov     esi, ebx
.text:00011726 E8 CB FE FF FF                 call    fnCheckFileNameInList ; check on .TMP
.text:00011726
.text:0001172B 84 C0                          test    al, al                            check .TMP
.text:0001172D 74 1C                          jz      short loc_1174B
.text:0001172D                                                                          extension
.text:0001172F
.text:0001172F                  jRemoveInfoFromBuf:                       ; CODE XREF: sub_11688+93↑j
.text:0001172F 8B 45 0C                       mov     eax, [ebp+arg_4]
.text:00011732 85 C0                          test    eax, eax
.text:00011734 74 2F                          jz      short loc_11765
.text:00011734
.text:00011736 8B 4D FC                       mov     ecx, [ebp+var_4]
.text:00011739 2B C8                          sub     ecx, eax
.text:0001173B 51                             push    ecx
.text:0001173C 03 C7                          add     eax, edi
.text:0001173E 50                             push    eax                      erase info
.text:0001173F 57                             push    edi
.text:00011740 FF 15 5C 1C 01+                call    ds:memmove
.text:00011746 83 C4 0C                       add     esp, 0Ch
```

It should be noted that such technique of files hiding were described in famous book "Rootkits: Subverting the Windows kernel" by Hoglund, Butler.

Driver 3

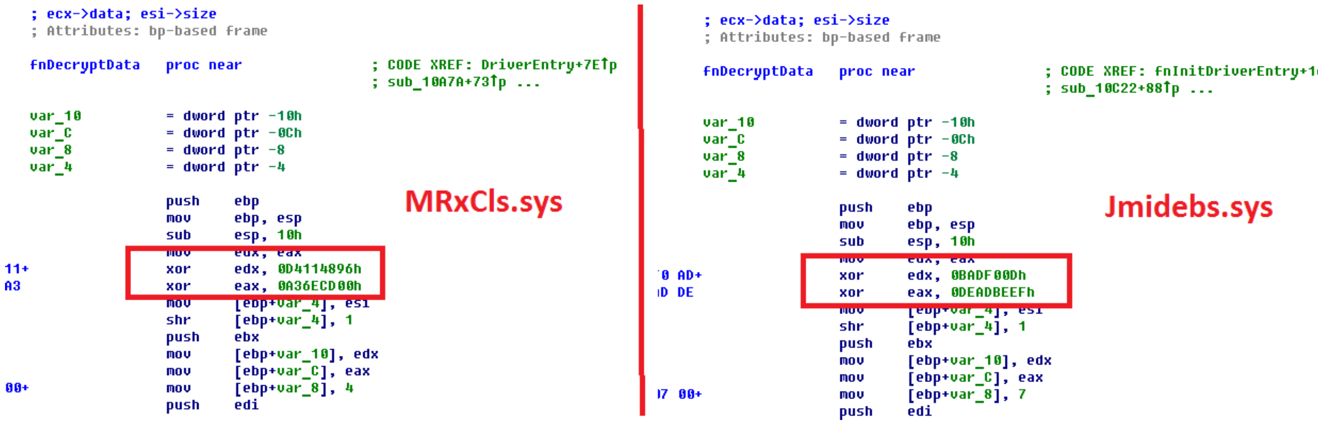
File name: Jmidebs.sys
SHA256: 63e6b8136058d7a06dfff4034b4ab17a261cdf398e63868a601f77ddd1b32802
File size: 25552 bytes
Signed: Yes
Timestamp: 2010-07-14 09:05:36
Device object name: \Device\{3093983-109232-29291}
Main purpose: code injection
AV detection ratio: 50/61

This driver is pretty similar to MRxCls.sys and serves only for code injection into processes. The following properties distinguish it from original MRxCls.sys.

- New device object name - \Device\{3093983-109232-29291}
- New registry service name - jmidebs
- New registry service parameter name (injection data) - IDE
- New IOCTL code for reading (caching) configuration data
- New constants in decryption routine.

Decrypted strings.

\REGISTRY\MACHINE\SYSTEM\CurrentControlSet\Services\jmidebs
IDE
\Device\{3093983-109232-29291}



The rootkit contains additional IOCTL function, which specializes in caching configuration data. This data are used for injection malicious Stuxnet code.

```
.text:00010B68                  fnDispatchDeviceControlRequest proc near
.text:00010B68                                                  ; CODE XREF: fnDispatchIrpMjFunctions+30↓p
.text:00010B68
.text:00010B68                  var_4           = dword ptr -4
.text:00010B68
.text:00010B68 55                               push    ebp
.text:00010B69 8B EC                            mov     ebp, esp
.text:00010B6B 51                               push    ecx
.text:00010B6C 8B 41 60                         mov     eax, [ecx+60h]
.text:00010B6F 8B 40 0C                         mov     eax, [eax+0Ch]
.text:00010B72 2D 00 38 22 00                   sub     eax, 223800h
.text:00010B77 74 20                            jz      short loc_10B99
.text:00010B77
.text:00010B79 83 E8 04                         sub     eax, 4
.text:00010B7C 74 07                            jz      short jDispatchNewIOCTLFunc
.text:00010B7C
.text:00010B7E B8 02 00 00 C0                   mov     eax, 0C0000002h
.text:00010B83 C9                               leave
.text:00010B84 C3                               retn
.text:00010B84
.text:00010B85                  ; ---------------------------------------------------------------------------
.text:00010B85
.text:00010B85                  jDispatchNewIOCTLFunc:          ; CODE XREF: fnDispatchDeviceControlRequest+14↑j
.text:00010B85 56                               push    esi
.text:00010B86 8D 75 FC                         lea     esi, [ebp+var_4]
.text:00010B89 E8 54 FF FF FF                   call    fnInit
.text:00010B89
.text:00010B8E 8B 45 FC                         mov     eax, [ebp+var_4]
.text:00010B91 E8 8C 00 00 00                   call    fnReadConfigurationData
.text:00010B91
```

new IOCTL function

Driver 4

File name: MRxCls.sys

SHA256: 1635ec04f069ccc8331d01fdf31132a4bc8f6fd3830ac94739df95ee093c555c

File size: 26616 bytes

Signed: Yes

Timestamp: 2009-01-01 18:53:25

Device object name: \Device\MRxClsDvX

Main purpose: code injection

AV detection ratio: 50/61

This sample is identical to driver 1, but signed with digital certificate. Both samples have identical timestamp value in PE header and identical code inside.

Conclusion

As we can see from the analysis, authors of Stuxnet Ring 0 part were interested in code injection and malicious files hiding. Driver MRxCls.sys has two instances, one unsigned and another with digital signature. Both drivers are identical and contain same compilation date. Driver Jmidebs.sys was compiled later than these two and I can call it "MRxCls.sys v2", because it contains some differences inside, but serves for same purpose. Driver Mrxnet.sys is a typical legacy FS filter driver that is used by attackers for hiding files in Windows.

	Driver 1	Driver 2	Driver 3	Driver 4
File name	MRxCls.sys	MRxCls.sys	Mrxnet.sys	Jmidebs.sys
File size	19840 bytes	26616 bytes	17400 bytes	25552 bytes
Signed	No	Yes	Yes	Yes
Timestamp	**2009-01-01 18:53:25**	**2009-01-01 18:53:25**	2010-01-25 14:39:24	2010-07-14 09:05:36
Main purpose	Code injection	Code injection	Malicious files hiding	Code injection
Uses encryption	Yes	Yes	No	Yes
Device name	MRxClsDvX	MRxClsDvX	CDO/Unnamed	{3093983-109232-29291}