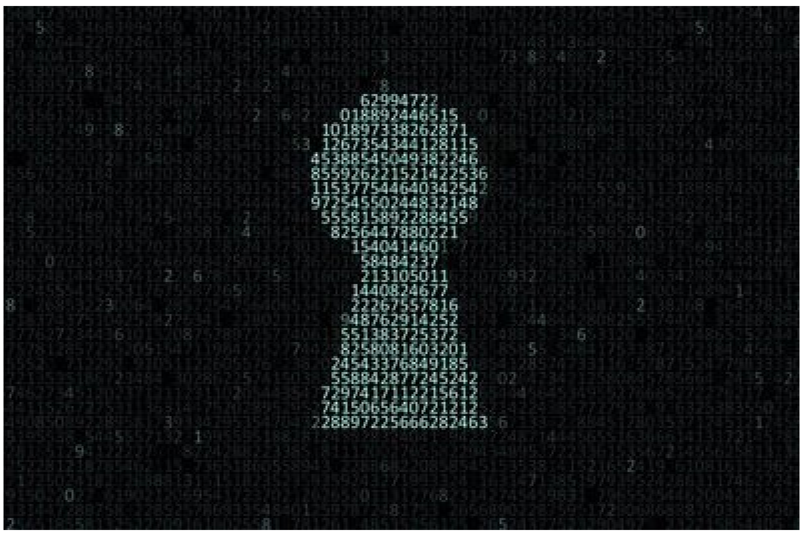
Remsec driver analysis - Part 2

artemonsecurity.blogspot.com/2016/10/remsec-driver-analysis-part-2.html

In previous blog post I've described 32-bit driver that has been used by attackers who are behind Strider cybergroup. I also pointed that from my point of view the driver was developed by skilled guys, but it contains two flaws. Firstly, authors forget to turn on SMEP again, after executing user mode code and they disable it each time when client tries to call 0x1173000C IOCTL code. Secondly, they try to unload driver dynamically in separate system thread that can lead to code execution from invalid memory (*fnThreadStartFunction*).



The dropper also contains one more driver (and its x64 clone) that is also interesting for research. I should make one clarification about information I posted before. The dropper itself doesn't contain rootkit driver as whole file inside, instead it stores only its PE-sections. This means that rootkit PE-file is generated by dropper on-the-fly. So, aswfilt.dll and 32-bit code of another driver as well as its 64-bit clone are stored only as PE-sections. And this is answer on question, why aswfilt.dll has one unnamed section and zeroed timestamp in PE-header. On screenshot below, you can see how the dropper initializes PE-header of aswfilt.dll driver before it was written to FS as executable file.

```
mov     [esp+1F0h+PeHdr.OptionalHeader.SectionAlignment], ebx
mov     [esp+1F0h+PeHdr.OptionalHeader.FileAlignment], ebx
mov     [esp+1F0h+PeHdr.OptionalHeader.MajorOperatingSystemVersion], si
mov     [esp+1F0h+PeHdr.OptionalHeader.MajorSubsystemVersion], si
mov     [esp+1F0h+PeHdr.OptionalHeader.Subsystem], bx
mov     [esp+1F0h+PeHdr.OptionalHeader.DataDirectory.VirtualAddress+8], ecx
push    8
mov     [esp+1F4h+PeHdr.OptionalHeader.SizeOfHeaders], ecx
mov     [esp+1F4h+var_8C], ecx
mov     [esp+1F4h+var_84], ecx
mov     [esp+1F4h+PeHdr.OptionalHeader.SizeOfImage], edx
pop     ebx
lea     eax, [ecx+46h]
mov     [esp+1F0h+var_48], eax
mov     [esp+1F0h+var_40], eax
mov     [esp+1F0h+var_194], 40h
mov     [esp+1F0h+PeHdr.Signature], 'EP'
mov     dword ptr [esp+1F0h+PeHdr.FileHeader.SizeOfOptionalHeader], 10E00E0h
mov     dword ptr [esp+1F0h+PeHdr.OptionalHeader.Magic], 9010Bh
mov     [esp+1F0h+PeHdr.OptionalHeader.ImageBase], 400000h
mov     [esp+1F0h+PeHdr.OptionalHeader.SizeOfStackReserve], 40000h
mov     [esp+1F0h+PeHdr.OptionalHeader.SizeOfHeapReserve], 100000h
mov     [esp+1F0h+PeHdr.OptionalHeader.NumberOfRvaAndSizes], 10h
mov     [esp+1F0h+PeHdr.OptionalHeader.DataDirectory.Size+8], 62h
mov     [esp+1F0h+PeHdr.OptionalHeader.DataDirectory.VirtualAddress+28h], 1C4h
mov     [esp+1F0h+PeHdr.OptionalHeader.DataDirectory.Size+28h], ebx
mov     [esp+1F0h+var_74], 0E8000020h
mov     [esp+1F0h+var_64], 198h
mov     [esp+1F0h+var_70], 188h
mov     [esp+1F0h+var_60], 190h
mov     esi, offset aNtoskrnl_exe ; "ntoskrnl.exe"
```

Driver (Ring 0 code) has following properties:

- It has compact size and its code is stored into two PE sections inside dropper.
- It has dynamic imports that are stored into special context structure.
- It has 64-bit clone inside the dropper.
- It has no *DriverEntry* function.
- It serves for one purpose: execute code from ptr that was passed from user mode to FastIoDeviceControl handler.
- It uses undocumented Windows kernel API.

Code and data of aswfilt.dll driver are stored into a separate section with name ".rwxdrv", as you can see on screenshot below. Another two sections with names ".krwkr32", ".krdrv32" and ".krwkr64", ".krdrv64" are used for storing mentioned above 32-bit Ring 0 code and its x64 analog.

Analysis [Section Headers]

Name	VirtualSize	VirtualAddress	SizeOfRawData	PointerToRawData
.text	0000D77F	00001000	0000D800	00000400
.krwkr64	00000469	0000F000	00000600	0000DC00
.krdrv64	000000AA	00010000	00000200	0000E200
.krwkr32	000002FE	00011000	00000400	0000E400
.krdrv32	0000009D	00012000	00000200	0000E800
.vdmbios	000004D6	00013000	00000600	0000EA00
.rwxdrv	00000A40	00014000	00000C00	0000F000
.rdata	000A7A24	00015000	000A7C00	0000FC00
.data	00000804	000BD000	00000400	000B7800
.reloc	00001754	000BE000	00001800	000B7C00

Like aswfilt.dll, kernel mode code from above mentioned sections uses special context structure where dynamically loaded imports are located. Format of this structure you can see below.

```
struct RootkitStruct {
PVOID pExAllocatePool;
PVOID pExFreePool;
PVOID pExQueueWorkItem;
PVOID pIofCompleteRequest;
PVOID pIoCreateDevice;
PVOID pIoDeleteDevice;
PVOID pIoDriverObjectType;
PVOID pIoGetCurrentProcess;
PVOID pKeInitializeEvent;
PVOID pKeSetEvent;
PVOID pKeStackAttachProcess;
PVOID pKeUnstackDetachProcess;
PVOID pKeWaitForSingleObject;
PVOID pObCreateObject;
PVOID pObInsertObject;
PVOID pObQueryNameString;
PVOID pObfDereferenceObject;
PVOID pZwClose;
PVOID pZwCreateFile;
PVOID pBuffer;
ULONG cbBuffer;
ULONG field1;
PVOID pProcessForAttach;
};
```

A problem is that start function of new kernel mode code doesn't contain *DriverEntry* function, showing for us that, in first, this code is loaded into Ring 0 not by Windows functions like *ZwLoadDriver* and in second that it can be loaded into memory with exploit. Anyway, start function of this kernel mode code, where the control will be passed at the beginning of its execution, gets already initialized context with corresponding function ptrs. There is no function inside Ring 0 code, which is responsible for filling context with dynamic loaded functions ptrs.

```
.krwkr32:100111D2 fnRootkitStartFunction proc near       ; DATA XREF: fnCreateDriverFromSections+8F↑o
.krwkr32:100111D2                                        ; fnCreateDriverFromSections+F9↑o
.krwkr32:100111D2
.krwkr32:100111D2 var_50          = dword ptr -50h
.krwkr32:100111D2 ObjAttr         = OBJECT_ATTRIBUTES ptr -38h
.krwkr32:100111D2 var_20          = byte ptr -20h
.krwkr32:100111D2 var_18          = dword ptr -18h
.krwkr32:100111D2 punDeviceName   = dword ptr -14h
.krwkr32:100111D2 var_10          = dword ptr -10h
.krwkr32:100111D2 pDrvObj1        = dword ptr -0Ch
.krwkr32:100111D2 var_8           = dword ptr -8
.krwkr32:100111D2 pContext1_hDevice= dword ptr  8
.krwkr32:100111D2
.krwkr32:100111D2                 push    ebp
.krwkr32:100111D3                 mov     ebp, esp
.krwkr32:100111D5                 sub     esp, 50h
.krwkr32:100111D8                 push    ebx
.krwkr32:100111D9                 xor     ecx, ecx
.krwkr32:100111DB                 push    esi
.krwkr32:100111DC                 mov     esi, [ebp+pContext1_hDevice]
.krwkr32:100111DF                 mov     [ebp+pDrvObj1], ecx
.krwkr32:100111E2                 mov     eax, [esi+RootkitStruct.field_24]
.krwkr32:100111E5                 mov     [ebp+punDeviceName], ecx
.krwkr32:100111E8                 mov     [ebp+pContext1_hDevice], ecx
.krwkr32:100111EB                 mov     [ebp+var_10], ecx
.krwkr32:100111EE                 push    edi
.krwkr32:100111EF                 lea     edx, [ebp+pDrvObj1] ; ppDrvObj1
.krwkr32:100111F2                 mov     ecx, esi        ; pContext1
.krwkr32:100111F4                 mov     [ebp+var_18], eax
.krwkr32:100111F7                 call    fnCreateDriverObject
```

First action, which *fnRootkitStartFunction* does, it is creating driver object for loaded Ring 0 code (*fnCreateDriverObject*). This function (*fnCreateDriverObject*) allocates an object in memory with help of *ObCreateObject*, initializes it and does it visible for Windows kernel by inserting it into objects list with *ObInsertObject*.

```
.krwkr32:100110DE                 mov     ecx, [ebp+pDrvObj]
.krwkr32:100110E1                 push    IO_TYPE_DRIVER
.krwkr32:100110E3                 lea     eax, [ecx+(size DRIVER_OBJECT)]
.krwkr32:100110E9                 mov     [ecx+DRIVER_OBJECT.DriverExtension], eax
.krwkr32:100110EC                 mov     ecx, [ebp+pDrvObj]
.krwkr32:100110EF                 mov     eax, [ecx+DRIVER_OBJECT.DriverExtension]
.krwkr32:100110F2                 mov     [eax], ecx
.krwkr32:100110F4                 mov     eax, [ebp+pDrvObj]
.krwkr32:100110F7                 pop     ecx
.krwkr32:100110F8                 mov     [eax+DRIVER_OBJECT.Type], cx
.krwkr32:100110FB                 mov     eax, [ebp+pDrvObj]
.krwkr32:100110FE                 mov     ecx, size DRIVER_OBJECT
.krwkr32:10011103                 mov     [eax+DRIVER_OBJECT.Size], cx
.krwkr32:10011107                 mov     eax, [ebp+pDrvObj]
.krwkr32:1001110A                 mov     [eax+DRIVER_OBJECT.Flags], 6 ; DRVO_LEGACY_DRIVER | DRVO_BUILTIN_DRIVER
.krwkr32:10011111                 mov     ecx, [ebp+pDrvObj]
.krwkr32:10011114                 mov     eax, [ecx+DRIVER_OBJECT.DriverExtension]
.krwkr32:10011117                 add     eax, 24h
.krwkr32:1001111A                 mov     [ecx+DRIVER_OBJECT.FastIoDispatch], eax
.krwkr32:1001111D                 mov     eax, [ebp+pDrvObj]
.krwkr32:10011120                 mov     eax, [eax+DRIVER_OBJECT.FastIoDispatch]
.krwkr32:10011123                 mov     [eax+FAST_IO_DISPATCH.SizeOfFastIoDispatch], size FAST_IO_DISPATCH
.krwkr32:10011129                 lea     eax, [ebp+hDriver]
.krwkr32:1001112C                 push    eax
.krwkr32:1001112D                 push    ebx
.krwkr32:1001112E                 push    1
.krwkr32:10011130                 push    1
.krwkr32:10011132                 push    ebx
.krwkr32:10011133                 push    [ebp+pDrvObj]
.krwkr32:10011136                 call    [esi_pContext+RootkitStruct.pObInsertObject]
```

Next, it does copying of prepared data with already initialized ptr from user mode buffer to system buffer and saves ptr to it into DriverExtension->ServiceKeyName.Buffer.

```
.krwkr32:10011159 jCopyDataToSystemPoolFromProcess:       ; CODE XREF: fnCreateDriverObject+CE↑j
.krwkr32:10011159                 mov     eax, [ebp+pDrvObj]
.krwkr32:1001115C                 mov     ebx, edi_pNewBuffer
.krwkr32:1001115E                 mov     eax, [eax+DRIVER_OBJECT.DriverExtension]
.krwkr32:10011161                 mov     [eax+DRIVER_EXTENSION1.ServiceKeyName.Buffer], edi_pNewBuffer
.krwkr32:10011164                 sub     ebx, [esi_pContext+74h]
.krwkr32:10011167                 lea     eax, [ebp+ApcState]
.krwkr32:1001116A                 push    eax
.krwkr32:1001116B                 push    [esi_pContext+RootkitStruct.pProcess]
.krwkr32:10011171                 call    [esi_pContext+RootkitStruct.pKeStackAttachProcess]
.krwkr32:10011171
.krwkr32:10011174                 mov     edx, [esi_pContext+RootkitStruct.cbBufferSize]
.krwkr32:10011177                 mov     ecx, [esi_pContext+RootkitStruct.pBuffer]
.krwkr32:1001117A                 test    edx, edx
.krwkr32:1001117C                 jz      short loc_10011187
.krwkr32:1001117C
.krwkr32:1001117E
.krwkr32:1001117E jNextByte:                              ; CODE XREF: fnCreateDriverObject+101↓j
.krwkr32:1001117E                 mov     al, [ecx]
.krwkr32:10011180                 mov     [edi_pNewBuffer], al
.krwkr32:10011182                 inc     edi_pNewBuffer
.krwkr32:10011183                 inc     ecx
.krwkr32:10011184                 dec     edx
.krwkr32:10011185                 jnz     short jNextByte
.krwkr32:10011185
.krwkr32:10011187
.krwkr32:10011187 loc_10011187:                           ; CODE XREF: fnCreateDriverObject+F8↑j
.krwkr32:10011187                 lea     eax, [ebp+ApcState]
.krwkr32:1001118A                 push    eax
.krwkr32:1001118B                 call    [esi_pContext+RootkitStruct.pKeUnstackDetachProcess]
```

The driver leverages interesting way for dispatching DeviceControl request. Unlike other drivers that are using IRP_MJ_DEVICE_CONTROL handler in such case, it registers FastIo function DriverObject->FastIoDispatch.FastIoDeviceControl.

```
.krwkr32:1001103D                 mov     esi, [ebp+arg_0]
.krwkr32:10011040                 push    1Bh
.krwkr32:10011042                 mov     eax, offset fnDispatchIrp
.krwkr32:10011047                 add     eax, esi
.krwkr32:10011049                 pop     ecx
.krwkr32:1001104A                 lea     edi, [ebx+DRIVER_OBJECT.MajorFunction]
.krwkr32:1001104D                 rep stosd
.krwkr32:1001104F                 mov     eax, offset fnDispatchIrpAndReturnSuccess
.krwkr32:10011054                 add     eax, esi
.krwkr32:10011056                 mov     [ebx+DRIVER_OBJECT.MajorFunction], eax
.krwkr32:10011059                 mov     [ebx+(DRIVER_OBJECT.MajorFunction+8)], eax
.krwkr32:1001105C                 mov     eax, offset fnDriverUnload
.krwkr32:10011061                 add     eax, esi
.krwkr32:10011063                 mov     [ebx+DRIVER_OBJECT.DriverUnload], eax
.krwkr32:10011066                 mov     eax, [ebx+DRIVER_OBJECT.FastIoDispatch]
.krwkr32:10011069                 mov     ecx, offset fnFastIoDeviceControl ; BOOLEAN
.krwkr32:10011069                                         ; (*PFAST_IO_DEVICE_CONTROL) (
.krwkr32:10011069                                         ;     IN struct _FILE_OBJECT *FileObject,
.krwkr32:10011069                                         ;     IN BOOLEAN Wait,
.krwkr32:10011069                                         ;     IN PVOID InputBuffer OPTIONAL,
.krwkr32:10011069                                         ;     IN ULONG InputBufferLength,
.krwkr32:10011069                                         ;     OUT PVOID OutputBuffer OPTIONAL,
.krwkr32:10011069                                         ;     IN ULONG OutputBufferLength,
.krwkr32:10011069                                         ;     IN ULONG IoControlCode,
.krwkr32:10011069                                         ;     OUT PIO_STATUS_BLOCK IoStatus,
.krwkr32:10011069                                         ;     IN struct _DEVICE_OBJECT *DeviceObject
.krwkr32:10011069                                         ;     );
.krwkr32:10011069                                         ;
.krwkr32:1001106E                 add     ecx, esi
.krwkr32:10011070                 mov     [eax+FAST_IO_DISPATCH.FastIoDeviceControl], ecx
.krwkr32:10011073                 mov     eax, [ebp+pDeviceObject]
.krwkr32:10011076                 and     [eax+DEVICE_OBJECT.Flags], 11111111111111111111111101111111b ; clear DO_DEVICE_INITIALIZING
.krwkr32:1001107D                 xor     eax, eax
```

As we already saw in case of aswfilt.dll, DeviceControl handler is responsible for performing only one task: execute function by ptr that has been passed to it from user mode client.

```
.krdrv32:10012000 fnFastIoDeviceControl proc near        ; DATA XREF: fnCreateDeviceAndInitDrvObject+69↑o
.krdrv32:10012000
.krdrv32:10012000 FileObject      = dword ptr  8
.krdrv32:10012000 Wait            = dword ptr  0Ch
.krdrv32:10012000 InputBuffer     = dword ptr  10h
.krdrv32:10012000 InputBufferLength= dword ptr  14h
.krdrv32:10012000 OutputBuffer    = dword ptr  18h
.krdrv32:10012000 OutputBufferLength= dword ptr  1Ch
.krdrv32:10012000 IoControlCode   = dword ptr  20h
.krdrv32:10012000 pIoStatus       = dword ptr  24h
.krdrv32:10012000 DeviceObject    = dword ptr  28h
.krdrv32:10012000
.krdrv32:10012000                 push    ebp
.krdrv32:10012001                 mov     ebp, esp
.krdrv32:10012003                 cmp     [ebp+IoControlCode], Rootkits_IOCTLs_ExecuteFunction
.krdrv32:1001200A                 jnz     short jInvalidDeviceRequest
.krdrv32:1001200A
.krdrv32:1001200C                 cmp     [ebp+InputBufferLength], 0Ch
.krdrv32:10012010                 jb      short jInvalidDeviceRequest
.krdrv32:10012010
.krdrv32:10012012                 push    esi
.krdrv32:10012013                 mov     esi, [ebp+InputBuffer]
.krdrv32:10012016                 push    [esi+Struct1.Argument]
.krdrv32:10012019                 call    [esi+Struct1.pFunction]
.krdrv32:10012019
.krdrv32:1001201B                 mov     ecx, [esi+8]
.krdrv32:1001201E                 pop     esi
.krdrv32:1001201F                 mov     [ecx], eax
.krdrv32:10012021                 mov     eax, [ebp+pIoStatus]
.krdrv32:10012024                 and     [eax+IO_STATUS_BLOCK.anonymous_0.Status], 0
.krdrv32:10012027                 mov     [eax+IO_STATUS_BLOCK.Information], 0Ch
.krdrv32:1001202E                 jmp     short loc_1001203D
```

Unfortunately, I haven't a lot of free time for reconstructing logic of Ring 3 code (dropper part) and how it forms context for kernel mode code. I know that IOCTL with code 0x839200BF (Rootkits_IOCTLs_ExecuteFunction) is used by the dropper only one time in function *fnSendIOCTL_839200BF*. It passes to driver ptr to function that located at address 0x1000741A.

The dropper also contains some code for exploiting NT Virtual DOS Machine (NTVDM), it contains section with name .vdmbios and it imports function *NtVdmControl*. The code also uses context structre for calling dynamic imports.