Remsec driver analysis - Part 3

artemonsecurity.blogspot.com/2016/10/remsec-driver-analysis-part-3.html

In two previous blog posts I've described 32-bit plugin that was mentioned by Kaspersky in their technical analysis. The plugin is called *kgate* and it has some interesting features, including, exploiting 32-bit Agnitum driver to run rootkit driver, run 32-bit or 64-bit kernel mode code by non-standart way. It's hard to say how stable this code works on live system, because authors use undocumented Windows kernel functions like *ObCreateObject* and *ObInsertObject* for creating new DriverObject.

There is one more 64-bit plugin that is called *xkgate* and it is used for compromising 64-bit Windows versions. Unlike *kgate* plugin, *xkgate* contains valid timestamp in PE header - 20 Aug 2014 (08:34:04). Both plugins contain code with identical functions in their .krwkr64 and .krdrv64 sections, but looks like *xkgate* plugin was written later than *kgate*.
Although, *kgate* plugin has zeroed timestamp in PE-header, its file contains one timestamp inside - Oct 28 2013. This means that operators have switched from *kgate* plugin, which was developed to load Ring 0 code for both x32 and x64 platforms, to special edition for x64 that is called *extended kgate*.

Like *kgate* plugin, *xkgate* also contains timestamp inside its file - Aug 19 2014. This timestamp confirms for us that date of compilation in PE header of *xkgate* is valid, although they differ by one day. I think the reason for this difference is that timestamp inside file contains time data for debug purpose and was set by authors in manual mode. Screenshot below shows this fact.

```
push    offset aOct282014 ; "Oct 28 2013"
push    offset aS         ; "(%s)"
lea     eax, [ebp+Dst]
push    103h              ; Count
push    eax               ; Dest
call    ds:_snwprintf
```

KGATE

```
lea     r9, aAug192014 ; "Aug 19 2014"
lea     r8, Format     ; "(%s)"
lea     rcx, [rsp+258h+Dst] ; Dest
mov     edx, 103h      ; Count
call    cs:_snwprintf
```

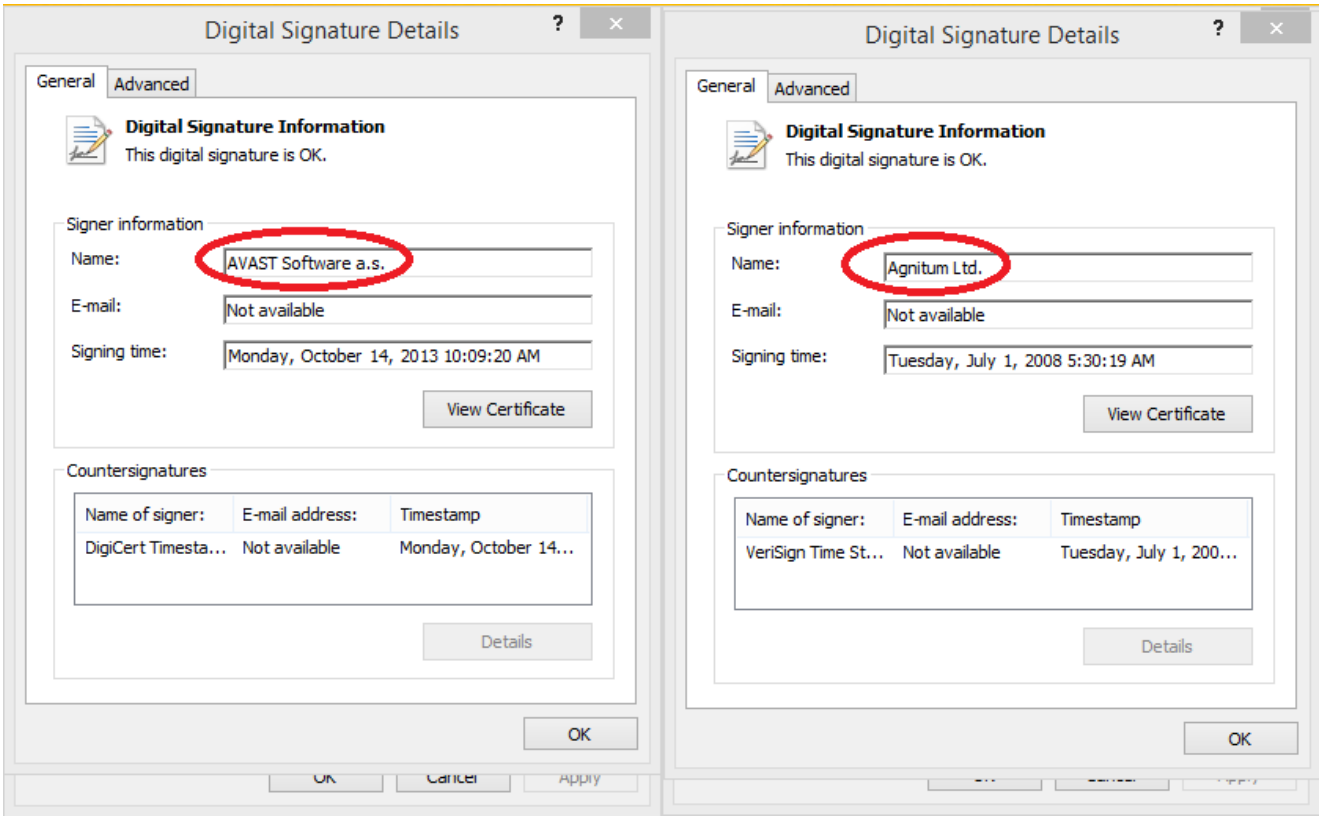
XKGATE

The plugin also contains Ring 0 code in two separate sections named .krwkr64 and .krdrv64. Section .rdata stores whole AVAST! Virtualization Driver file aswsnx.sys, which is used by xkgate for loading own kernel mode code. Section .avit contains code for communication with AVAST driver from "trusted" process.

Analysis [Section Headers]

Name	VirtualSi…	VirtualAd…	SizeOfRaw…	PointerTo…	PointerTo…	PointerTo…	NumberOfR…	NumberOfL…	Character…
.text	0000691F	00001000	00006A00	00000400	00000000	00000000	0000	0000	60000020
.krwkr64	00000469	00008000	00000600	00006E00	00000000	00000000	0000	0000	E0000020
.krdrv64	000000AC	00009000	00000200	00007400	00000000	00000000	0000	0000	E0000020
.avit	00000303	0000A000	00000400	00007600	00000000	00000000	0000	0000	E0000020
.rdata	000FEB64	0000B000	000FEC00	00007A00	00000000	00000000	0000	0000	40000040
.data	00000248	0010A000	00000200	00106600	00000000	00000000	0000	0000	C0000040
.pdata	00000498	0010B000	00000600	00106800	00000000	00000000	0000	0000	40000040
.reloc	00000114	0010C000	00000200	00106E00	00000000	00000000	0000	0000	42000040

Below you can see information about digital signatures of legitimate drivers Outpost and AVAST! that have been used by Remsec authors for loading Ring 0 code.



According to the language that has been used in debug comments for both plugins, authors were native English speakers. But it is not clear why they don't remove debug information from it. Below you can see strings, which are present into *xkgate*.

Load error: Access Denied
Load error: Unsupported OS
Load error: Invalid plugin image format!
Load error: Plugin entry point not found!
Load error: Failed to resolve kernel functions!
Load error: Out of memory!
Load error: Status unsuccessful!
Load error: Failed to run plugin (%#x)
Unsupported OS! Only Windows 2000 and later supported!
Unable to determine 32/64-bit OS!
Invalid plugin name: path not allowed, try using -n.
Invalid plugin name: suffix not allowed, try using -n.
Unable to load kernel plugin %s!
Unable to build argv!
Plugin successfully executed!
GateDriver is currently disabled on x64 systems due to driver signing restrictions!

In table below you can see characteristics of both plugins.

Plugin / Feature	*kgate*	*xkgate (extended kgate)*
Cross-platform (contains Ring 0 code for both x32 and x64)	+	-
Uses legitimate AV driver to run Ring code	Agnitum	AVAST!
Storage section for AV driver	.rdata	.rdata
Drops Ring 0 code (driver) on disk	+	-
Refs to IOCTLs inside plugin body	0x10009C4D 0x10009C98 0x10006B78	0x180003B7E
Ring 0 code primary purpose	LPE – run arbitrary code with SYSTEM privileges	LPE – run arbitrary code with SYSTEM privileges
Contains identical Ring 0 code	krwkr64 and krdrv64	krwkr64 and krdrv64
Uses offsets obfuscation	+	+
Uses dynamic imports	+	+
Timestamp	28 Oct 2013	19 Aug 2014
	Both use same plugin architecture	

Ring 0 code	IOCTL	Description
aswfilt.dll	0x11730004	Copy data (is not used by plugin)
	0x11730008	Copy data (is not used by plugin)
	0x1173000C	Execute function in Ring 0 with SMEP bypass
	0x117300C8	Set driver unload function
	0x117300CC	Set timer interval (is not used by plugin)
krwkr[32/64] + krdrv[32/64]	0x839200BF	Execute function in Ring 0

From the table above you can see that some IOCTLs functions are not used by attackers.

As I mentioned above, *xkgate* leverages Avast driver for executing Ring 0 code which

doesn't drop to FS. There is function *fnLoadAvast* that is responsible for loading Avast driver in proper way. It also does some actions for reproducing correct environment for it. Unlike exploiting Agnitum driver, this situation is more hard for exploitation. Let's look more detailed.

First action that *fnLoadAvast* does, it tries to create mutex with name Global\yRg7d3x and checks a result of *WaitForSingleObject* to prevent doing same actions again.

```
4C 8D 05 09 7D+                 lea     r8, Name        ; "Global\\yRg7d3x"
48 8D 4C 24 78                  lea     rcx, [rsp+268h+pSecurityDescriptor] ; lpMutexAttributes
33 D2                           xor     edx, edx        ; bInitialOwner
FF 15 FC 65 00+                 call    cs:CreateMutexA
00
4C 8B E8                        mov     r13, rax
48 85 C0                        test    rax, rax
74 1F                           jz      short loc_180004B13

BA 88 13 00 00                  mov     edx, 5000       ; dwMilliseconds
48 8B C8                        mov     rcx, rax        ; hHandle
FF 15 5E 66 00+                 call    cs:WaitForSingleObject
00
85 C0                           test    eax, eax
0F 84 8F 00 00+                 jz      jContinue
00
49 8B CD                        mov     rcx, r13        ; hObject
FF 15 75 65 00+                 call    cs:CloseHandle
00

                loc_180004B13:                          ; CODE XREF: fnLoadAvast+CA↑j
BB 19 00 00 00                  mov     ebx, 19h
```

Next, the code calls *fnDropAvastDriverAndPrepareEnv* function that does following actions:

- Creates directory \SystemRoot\Temp\aswSnx for dropping AVAST related files.
- Drops aswSnx.sys to FS.
- Drops snx_lconfig.xml to FS.
- Drops snx_gconfig.xml.
- Creates empty file snxhk.dll.
- Creates aswSnx.exe and writes to it content of notepad.exe.

```
.text:0000000180004C1A
.text:0000000180004C1A jDropFiles:                              ; CODE XREF: fnLoadAvast+1AD↑j
.text:0000000180004C1A                  lea     r9, [rbp+1A0h+h_snxhk64.dll]
.text:0000000180004C21                  lea     r8, [rbp+1A0h+h_snxhk.dll]
.text:0000000180004C28                  lea     rdx, [rbp+1A0h+ApplicationName_aswSnx.exe]
.text:0000000180004C2C                  mov     ecx, r12d
.text:0000000180004C2F                  call    fnDropAvastDriverAndPrepareEnv
.text:0000000180004C2F
.text:0000000180004C34                  mov     r15, [rbp+1A0h+h_snxhk.dll]
.text:0000000180004C3B                  mov     rsi, [rbp+1A0h+h_snxhk64.dll]
.text:0000000180004C42                  test    eax, eax
.text:0000000180004C44                  jz      jCleanupAndRet
.text:0000000180004C44
.text:0000000180004C4A                  mov     [rbp+1A0h+var_4C], dil
.text:0000000180004C51                  test    r12d, r12d
.text:0000000180004C54                  jz      loc_180004D19
.text:0000000180004C54
.text:0000000180004C5A                  call    fnCreateAvastServiceAndLoadDriver
.text:0000000180004C5A
.text:0000000180004C5F                  xor     r12d, r12d
.text:0000000180004C62                  test    eax, eax
.text:0000000180004C64                  jz      jCleanupAndRet
.text:0000000180004C64
.text:0000000180004C6A                  lea     rcx, aNtdll     ; "ntdll"
.text:0000000180004C71                  call    cs:GetModuleHandleA
```

Next, *fnLoadAvast* calls *fnCreateAvastServiceAndLoadDriver* for creating AVAST service key in registry.

- It creates key System\CurrentControlSet\Services\aswSnx.
- Creates parameter ImagePath with path to \SystemRoot\Temp\aswSnx\aswSnx.sys.
- Creates parameter Type.
- Creates subkey Parameters.
- Creates parameter DataFolder inside subkey with value \?? \Global\GLOBALROOT\SystemRoot\Temp\aswSnx.
- Creates parameter ProgramFolder with value \?? \Global\GLOBALROOT\SystemRoot\Temp\aswSnx.
- Creates key subkey Instances.
- Creates parameter DefaultInstance inside Instances key.
- Creates parameters Altitude and Flags.
- Loads aswSnx.sys with *NtLoadDriver*.

After aswSnx.sys was loaded, *fnLoadAvast* removes snxhk.dll and snxhk64.dll files from FS with help of *NtSetInformationFile*. Next step is creating process aswSnx.exe, which actually is notepad.exe. After that it opens handle on AVAST device \Device\aswSnx.

```
.text:0000000180004D43
.text:0000000180004D4A
.text:0000000180004D4A jCreateProcess:                          ; CODE XREF: fnLoadAvast+2FB↑j
.text:0000000180004D4A                  lea     rax, [rsp+268h+hProcess]
.text:0000000180004D4F                  lea     rdx, [rbp+1A0h+ApplicationName_aswSnx.exe] ; lpCommandLine
.text:0000000180004D53                  lea     rcx, [rbp+1A0h+ApplicationName_aswSnx.exe] ; lpApplicationName
.text:0000000180004D57                  mov     [rsp+268h+lpProcessInformation], rax ; lpProcessInformation
.text:0000000180004D5C                  lea     rax, [rbp+1A0h+StartupInfo]
.text:0000000180004D60                  xor     r9d, r9d        ; lpThreadAttributes
.text:0000000180004D63                  mov     [rsp+268h+lpStartupInfo], rax ; lpStartupInfo
.text:0000000180004D68                  mov     [rsp+268h+lpCurrentDirectory], r12 ; lpCurrentDirectory
.text:0000000180004D6D                  mov     [rsp+268h+hTemplateFile], r12 ; lpEnvironment
.text:0000000180004D72                  xor     r8d, r8d        ; lpProcessAttributes
.text:0000000180004D75                  mov     [rsp+268h+dwFlagsAndAttributes], CREATE_SUSPENDED ; dwCreationFlags
.text:0000000180004D7D                  mov     [rsp+268h+dwCreationDisposition], r12d ; bInheritHandles
.text:0000000180004D82                  call    cs:CreateProcessA
.text:0000000180004D82
.text:0000000180004D88                  test    eax, eax
.text:0000000180004D8A                  jz      jCleanupAndRet
```

After creating aswSnx.exe (notepad) process in suspended state, it duplicates handle to \Device\aswSnx from current plugin process into new process aswSnx.exe. As you already guessed, next step is copying code for communicating with AVAST driver into aswSnx.exe. Copied code is located in special .avit section and performs *DeviceIoControl* that triggers code execution of Ring 0 rootkit code from AVAST driver.

```
.avit:000000018000A210                  mov     [rbp+57h+var_s28], 0FFFFFFFFFFF0BDC0h
.avit:000000018000A218                  mov     rcx, [r15+8]
.avit:000000018000A21C                  xor     r8d, r8d
.avit:000000018000A21F                  add     r9, rax
.avit:000000018000A222                  mov     [rsp+28h], ebx
.avit:000000018000A226                  mov     [rsp+20h], r15
.avit:000000018000A22B                  call    [rbp+57h+pSetWaitableTimer]
.avit:000000018000A22B
.avit:000000018000A22E                  test    eax, eax
.avit:000000018000A230                  jz      loc_18000A162
.avit:000000018000A230
.avit:000000018000A236                  mov     rcx, [r15]      ; hAvastDevice
.avit:000000018000A239                  mov     [rsp+88h+pExitThread], rbx
.avit:000000018000A23E                  lea     rax, [rbp+57h+var_s18]
.avit:000000018000A242                  mov     [rsp+88h+pSleepEx], rax
.avit:000000018000A247                  xor     r9d, r9d
.avit:000000018000A24A                  xor     r8d, r8d
.avit:000000018000A24D                  mov     edx, [redacted]  ; IOCTL code
.avit:000000018000A252                  mov     [rsp+28h], ebx
.avit:000000018000A256                  mov     [rsp+20h], rbx
.avit:000000018000A25B                  call    [rbp+57h+pDeviceIoControl]
.avit:000000018000A25B
```

Into loaded Ring 0 code, we can see already known to us function for dispatching FastDeviceControl request.

```
.krdrv64:0000000180009000 fnDispatchFastIoDeviceControl proc near ; DATA XREF: sub_180008000+8B↑o
.krdrv64:0000000180009000                                         ; .pdata:0000000180108468↓o
.krdrv64:0000000180009000
.krdrv64:0000000180009000 IoControlCode   = dword ptr  38h
.krdrv64:0000000180009000 arg_38          = qword ptr  40h
.krdrv64:0000000180009000
.krdrv64:0000000180009000                 push    rbx
.krdrv64:0000000180009002                 sub     rsp, 20h
.krdrv64:0000000180009006                 cmp     [rsp+28h+IoControlCode], 839200BFh ; Rootkit_IOCTLs_ExecuteFunction
.krdrv64:000000018000900E                 mov     rbx, r8
.krdrv64:0000000180009011                 jnz     short loc_180009038
.krdrv64:0000000180009011
.krdrv64:0000000180009013                 cmp     r9d, 18h
.krdrv64:0000000180009017                 jb      short loc_180009038
.krdrv64:0000000180009017
.krdrv64:0000000180009019                 mov     rcx, [r8+8]     ; Function argument
.krdrv64:000000018000901D                 call    qword ptr [r8]  ; pUserFunction
.krdrv64:000000018000901D
.krdrv64:0000000180009020                 mov     rcx, [rbx+10h]
.krdrv64:0000000180009024                 mov     [rcx], eax
.krdrv64:0000000180009026                 mov     rax, [rsp+28h+arg_38]
.krdrv64:000000018000902B                 and     [rax+IO_STATUS_BLOCK.anonymous_0.Status], 0
.krdrv64:000000018000902E                 mov     [rax+IO_STATUS_BLOCK.Information], 18h
.krdrv64:0000000180009036                 jmp     short loc_180009048
```

As you can see, both Agnitum and AVAST! drivers became exploitable for Remsec authors, because both don't use check of caller process based on digital signature properly. Although, notepad.exe is signed with Microsoft digital certificate that means AVAST! can check digital signature of caller process in IRP_MJ_CREATE handler, but doesn't check the name of signer.