Notes from a simple Malware Analysis – 20160103 v.1.1

Dear readers, how are you? First all, Happy New Year! I wish an excellent 2016 for you! Unfortunately, as you should know, our subject is related to small concepts about malwares and it is never a pleasurable topic. Anyway, this time, I am going to show a short and summarized analysis of an educational malware, where I will not be showing a complete analysis (this practice would be only for a class) . The shortened procedure follows:

1. This malware was composed for two files: malw.exe and malw.sys.

2. It was made a quick static analysis before starting to analyzing the drive file (not shown here).

3. The malware presented a complicated issue. The executable (malw.exe) loaded the driver file (malw.sys), but it immediately unloaded it. Thus, there was a small issue because if we tried to analyze the malw.sys by using the kernel debugger (WinDbg) without executing the malw.exe, so the driver would not be in the memory and it will would be impossible to analyze it. Nonetheless, if we tried to analyze the malw.sys after the executable (malw.exe) having finished, the driver would already have been unloaded from memory by the malw.exe file. What was the solution? We needed to set a breakpoint after the driver has been loaded, but before it having been unloaded. It can be done inside the virtual machine where we were running the malware by using WinDbg because we could choose an appropriate address to breakpoint (from static analysis)

4. After having set the breakpoint, we run the malware and waited it to hit the breakpoint.

5. Now an interesting trick: from host machine (outside virtual machine), we executed WinDbg (a kernel debugger) and connected to virtual machine.

6. To acquire more information about the driver file, we executed the following command from WinDbg:

   kd> **!drvobj malw1**

   ```
   Driver object (8496f030) is for:
   *** ERROR: Module load completed but symbols could not be loaded for Malw1.sys
    \Driver\Malw1
   Driver Extension List: (id , addr)
   Device Object list:
   ```

7. As we are able to see, there was not any device associated to this drive.

8. To get a list of drivers objects found in the memory, we executed:

   kd> **!object \Driver**

   ```
   Object: e15ae030  Type: (84bf03b0) Directory
      ObjectHeader: e15ae018 (old version)
      HandleCount: 0  PointerCount: 80
   ```

```
Directory Object: e10005d8  Name: Driver
Hash Address            Type         Name
----  -------           ----        ---------------
 00   8484a408          Driver       Beep
      84b62770          Driver       NDIS
      84b95390          Driver       KSecDD
 01   84984ca8          Driver       Mouclass
      84955a28          Driver       Raspti
      84960be0          Driver       es1371
 02   84965408          Driver       vmx_svga
 03   84867f38          Driver       Fips
      84ac99d8          Driver       Kbdclass
 04   8484b408          Driver       VgaSave
      84904030          Driver       NDProxy
      84b8f9a8          Driver       Compbatt
 05   84907030          Driver       Ptilink
      84ba2c48          Driver       MountMgr
      849c02b0          Driver       wdmaud
 06   8496f030          Driver       Malw1
 07   84ba1030          Driver       dmload
      84b909f0          Driver       isapnp
      ----snip------
```

9. Making an overlay by composing the driver object with the driver object address and the _DRIVER_OBJECT_ structure, we got:

```
kd>  dt _DRIVER_OBJECT 8496f030
         ntdll!_DRIVER_OBJECT
           +0x000 Type            : 0n4
           +0x002 Size           : 0n168
           +0x004 DeviceObject     : (null)
           +0x008 Flags           : 0x12
           +0x00c DriverStart      : 0xf7cad000 Void
           +0x010 DriverSize       : 0xe80
           +0x014 DriverSection    : 0x8484be78 Void
           +0x018 DriverExtension  : 0x8496f0d8 _DRIVER_EXTENSION
           +0x01c DriverName       : _UNICODE_STRING "\Driver\Malw1"
           +0x024 HardwareDatabase : 0x80670ae0 _UNICODE_STRING
         "\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"
           +0x028 FastIoDispatch   : (null)
           +0x02c DriverInit       : 0xf7cad959     long  +0
           +0x030 DriverStartIo    : (null)
           +0x034 DriverUnload     : 0xf7cad486     void  +0
           +0x038 MajorFunction    : [28] 0x804f354a     long
         nt!IopInvalidDeviceRequest+0
```

10. As we knew the address of the function (DriverUnload) where the malware was unloaded (where we missed the chance of analyzing the driver and a service was installed – shown from static analysis), so we set a breakpoint:

```
kd> bp 0xf7cad486
```

11. We resumed the kernel in the WinDbg from host (as shown below) and, afterwards, we resumed it from WinDbg from virtual machine (not showed):

```
kd> g

Breakpoint 0 hit
Malw+0x486:
f7cad486 8bff          mov     edi,edi
```

12. Our code hit the breakpoint set in the Unload function. Next, we returned to WinDbg from host (outside virtual machine) by executing the WinDbg in step-in mode (**t** command) and step-over mode (**p** command):

```
kd> t
Malw+0x489:
f7cad489 8bec          mov     ebp,esp
kd> t
Malw+0x48b:
f7cad48b 51            push    ecx
kd> t
Malw+0x48c:
f7cad48c 53            push    ebx
kd> t
Malw+0x48d:
f7cad48d 56            push    esi
kd> t
Malw+0x48e:
f7cad48e 8b3580d7caf7    mov     esi,dword ptr [Malw+0x780 (f7cad780)]

(snip)

kd> t
nt!RtlCreateRegistryKey:
805ddafe 8bff          mov     edi,edi
kd> t
nt!RtlCreateRegistryKey+0x2:
805ddb00 55            push    ebp

(snip)

kd> t
nt!RtlCreateRegistryKey+0x11:
805ddb0f e8b6f4ffff      call    nt!RtlpGetRegistryHandle (805dcfca)
kd> t
nt!RtlpGetRegistryHandle:
805dcfca 8bff          mov     edi,edi

(snip)

kd> t
nt!RtlAppendUnicodeToString:
8052804a 8bff          mov     edi,edi
```

```
kd> t
nt!RtlAppendUnicodeToString+0x2:
8052804c 55             push    ebp
kd> t
nt!RtlAppendUnicodeToString+0x3:
8052804d 8bec           mov     ebp,esp
```

(snip)

```
kd> t
nt!RtlInitUnicodeString+0x14:
8052ad80 7422           je      nt!RtlInitUnicodeString+0x38 (8052ada4)
kd> t
nt!RtlInitUnicodeString+0x16:
8052ad84 83c9ff         or      ecx,0FFFFFFFFh
kd> t
nt!RtlInitUnicodeString+0x19:
8052ad85 33c0           xor     eax,eax
```

(snip)

```
kd> p
nt!RtlAppendUnicodeToString+0x1c:
80528066 8b4df4         mov     ecx,dword ptr [ebp-0Ch]
kd> p
nt!RtlAppendUnicodeToString+0x1f:
80528069 8b7508         mov     esi,dword ptr [ebp+8]
```

(snip)

13. We found fews functions:

    - **RtlCreateRegistryKey** → it adds a key object in the registry along a given relative path.
    - **RtlpGetRegistryHandle** → it gets a handle to a registry key.
    - **RtlAppendUnicodeToString** → it concatenates a buffered unicode string and a `'\0'` terminated unicode string.
    - **RtlInitUnicodeString** → it initializes a counted string of Unicode characters.

14. Analyzing these functions above, we could make a well educated guess and understood that the malware driver was trying to create few registry entries. Thus, we had to find some keys and it was the easy part because we only need to examine the arguments pushed on stack by the malware to find them:

```
kd> p
Malw+0x4a2:
f7cad4a2 6840d6caf7     push    offset Malw+0x640 (f7cad640)
kd> p
Malw+0x4a7:
f7cad4a7 57             push    edi
kd> p
Malw+0x4a8:
```

```
f7cad4a8 ffd6          call    esi
kd> p
Malw+0x4aa:
f7cad4aa 68a8d5caf7      push    offset Malw+0x5a8 (f7cad5a8)
kd> p
Malw+0x4af:
f7cad4af 57            push    edi
kd> p
Malw+0x4b0:
f7cad4b0 ffd6          call    esi
```

(snip)

```
kd> du f7cad5a8
f7cad5a8  "\Registry\Machine\SOFTWARE\Polic"
f7cad5e8  "ies\Microsoft\WindowsFirewall\Do"
f7cad628  "mainProfile"

kd> du f7cad640
f7cad640  "\Registry\Machine\SOFTWARE\Polic"
f7cad680  "ies\Microsoft\WindowsFirewall"
```

15. Looking for these registry keys on the Internet we could find that they are responsible for Windows XP Firewall and that the malware was trying to disable the Windows Firewall.

16. Finally, we listed the kernel modules to find the address of the malware because its address in WinDbg is different from disassembler (IDA Pro) and we needed to rebased it in the IDA Pro. Therefore, we executed:

```
kd> lm

start    end        module name
7c900000 7c9af000   ntdll      (pdb symbols)
c:\symbols\ntdll.pdb\1751003260CA42598C0FB326585000ED2\ntdll.pdb
804d7000 806cf580   nt         (pdb symbols)
c:\symbols\ntkrnlpa.pdb\30B5FB31AE7E4ACAABA750AA241FF3311\ntkrnlpa.pdb
806d0000 806f0300   hal        (deferred)
```

(snip)

```
f7b7d000 f7b7ea80   ParVdm     (deferred)
7baf000 f7bafc00   audstub    (deferred)
f7c18000 f7c18d00   dxgthk     (deferred)
f7cad000 f7cade80   Malw   (no symbols)
f7cdd000 f7cddb80   Null       (deferred)
Unloaded modules:
f421d000 f4248000   kmixer.sys
f7d18000 f7d19000   drmkaud.sys
f4248000 f426b000   aec.sys
f76b3000 f76c0000   DMusic.sys
f76a3000 f76b1000   swmidi.sys
```

```
f7b29000 f7b2b000   splitter.sys
f7643000 f764e000   imapi.sys
f7633000 f763c000   processr.sys
f78fb000 f7900000   Cdaudio.SYS
f6c22000 f6c25000   Sfloppy.SYS
```

17. Using this value, we could calculate the this offset to rebase the malware in the IDA Pro:

    ```
    kd> ? (0xf7cad486 - 0xf7cad000)
    Evaluate expression: 0x486
    ```

18. It’s perfect! We lauched the IDA Pro and opened the Malw.sys driver file. Afterwards, we could have either mentally added this offset during the our analysis or made the rebasing by going to **Edit → Segment → Rebase Program** by changing the base address value with the start address of Malw.sys driver (f7cad000).

Please, let’ s remember that it was an educational malware, but you can repeat the same procedure during your real cases. Additionally, it was a very simple case, which I omitted several points (I didn’ t show the static analysis, for example).

Have a nice day.

Alexandre Borges.
(LinkedIn: http://www.linkedin.com/in/aleborges)