ERNW
providing security.

The TPM: Communication Interfaces

Aleksandar Milenkoski✉
amilenkoski@ernw.de

This work is part of the *Windows Insight* series. This series aims to assist efforts on analysing inner working principles, functionalities, and properties of the Microsoft Windows operating system. For general inquiries contact Aleksandar Milenkoski (amilenkoski@ernw.de) or Dominik Phillips (dphillips@ernw.de). For inquiries on this work contact the corresponding author (✉).

The content of this work has been created in the course of the project named 'Studie zu Systemaufbau, Protokollierung, Härtung und Sicherheitsfunktionen in Windows 10 (SiSyPHuS Win10)' (ger.) - 'Study of system design, logging, hardening, and security functions in Windows 10' (eng.). This project has been contracted by the German Federal Office for Information Security (ger., Bundesamt für Sicherheit in der Informationstechnik - BSI).

Required Reading

In addition to referenced work, related work focussing on the Trusted Platform Module (TPM), part of the *Windows Insight* series, are relevant for understanding concepts and terms mentioned in this document.

Technology Domain

The operating system in focus is Windows 10, build 1607, 64-bit, long-term servicing branch (LTSB).

The TPM standard in focus is version 2.0.

1 Introduction

In this work, we discuss how the different components of the Windows 10 operating system deployed in user-land (Section 2) and in kernel-land (Section 3), use the TPM. We focus on the communication interfaces between Windows 10 and the TPM, which we depict in Figure 1. In addition, we discuss the construction of TPM usage profiles, that is, information on system entities communicating with the TPM as well as on communication patterns and frequencies (Section 3.1).

2 TPM Communication Interfaces: User-land

The components of the Windows 10 system deployed in user-land (referred to as Executable in Figure 1) can communicate with the TPM in two ways: direct (direct TPM communication in Figure 1) or abstracted (abstracted TPM communication in Figure 1).

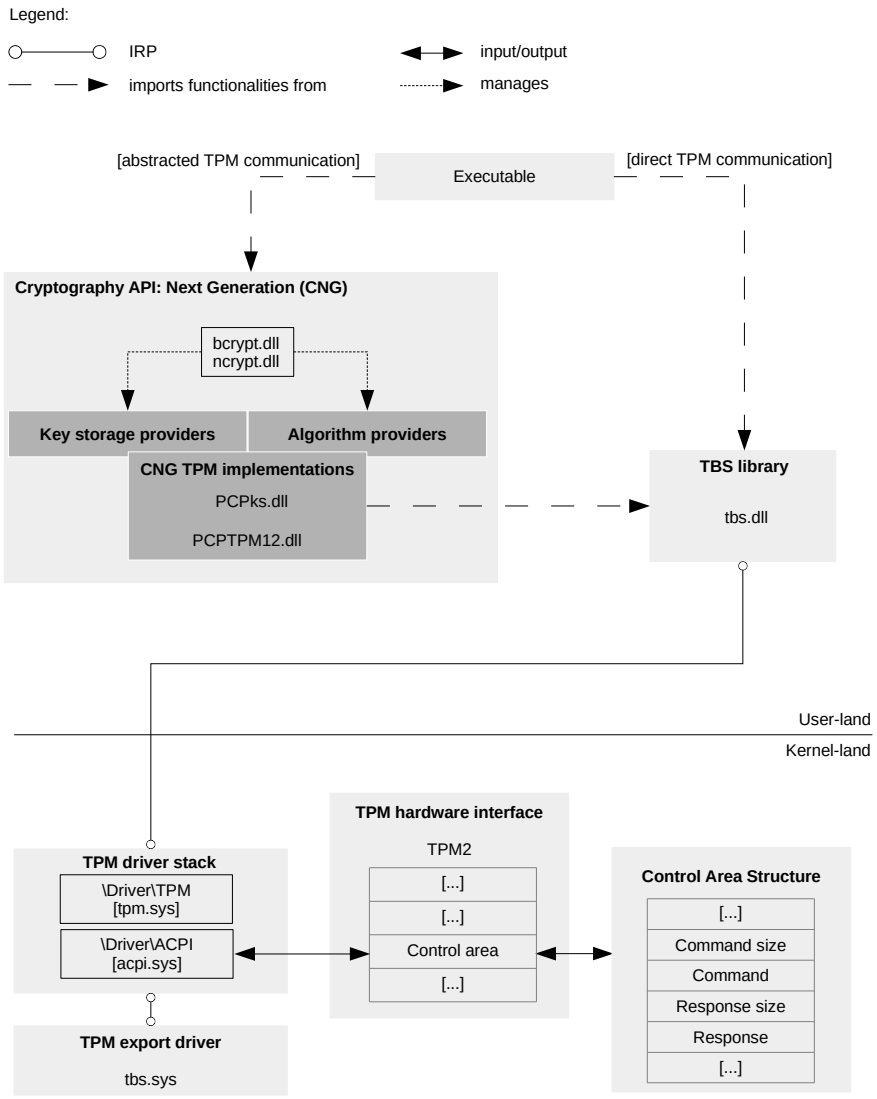

Figure 1: Interfaces for communicating with the TPM

2.1 Direct TPM communication

The direct TPM communication involves executing functions declared as part of the TPM Base Services (TBS) library file named *tbs.dll* (TBS library in Figure 1). This library file implements a number of functions, structures, and data types for communicating with the TPM.[1] Example functions are *Tbsi_GetDeviceInfo* for obtaining relevant information about the TPM device and *Tbsi_Get_OwnerAuth* for obtaining the owner authorization value. Most of the functions implemented as part of the TBS library perform TPM operations by constructing TPM command buffers (see Section 1.3) and submitting them to the TPM device by invoking the *Tbsip_Submit_Command* function.[2] From the perspective of user-land system entities, this function represents the communication interface to the TPM at the lowest-level; that is, it submits commands to the TPM device in their raw form, as byte sequences.

The submission of commands from the TBS library to the TPM involves issuing a system call to the TPM driver,

[1] https://msdn.microsoft.com/de-de/library/windows/desktop/aa446794(v=vs.85).aspx [Retrieved: 22/9/2017]

[2] https://msdn.microsoft.com/de-de/library/windows/desktop/aa446799(v=vs.85).aspx [Retrieved: 22/9/2017]

passing the TPM command byte sequence in the form of (input/output) I/O request packets (IRPs).[3] The system call may be issued, for example, using the *NtDeviceIoControlFile* Windows application programming interface (API) function.[4]

The TPM driver is implemented in the *%SystemRoot%\System32\drivers\tpm.sys* driver executable file. This driver submits commands passed to it from the TBS library to the TPM device as discussed next.

Windows drivers may be structured into driver stacks, where drivers at higher levels process submitted IRPs and submit them to drivers at lower levels. The driver at the lowest level communicates with the actual device to which the submitted IRP is destined.[5] In a given driver stack, there may be: a single function driver, which is a driver developed by the vendor of the device handling the majority of submitted IRPs; filter drivers performing auxiliary roles in IRP processing; and bus drivers communicating with the actual device.

With each driver that is part of a driver stack is associated a driver object and a device object of the physical device.[6] The device object is a representation of the device at the level at which the driver resides. For example, there are functional device objects (FDOs), which are associated with functional drivers, and physical device objects (PDOs), which are associated with bus drivers. The driver and device objects have names associated with them so that user-land system entities can reference them in program code.[7]

The TPM driver is the upper layer of the TPM driver stack. On Advanced Configuration and Power Interface (ACPI)-enabled platforms, this stack consists of the functional driver *tpm.sys* and the bus ACPI driver *acpi.sys*. A functional device object is associated with the functional driver *tpm.sys* (driver object named *\Driver\TPM*) and a physical device object is associated with the ACPI driver *acpi.sys* (driver object named *\Driver\ACPI*).

Following the hierarchy of the TPM driver stack, when a command in the form of an IRP is submitted to the TPM driver *tpm.sys*, it passes the procession of the IRP to the ACPI driver *acpi.sys*. According to the Trusted Computing Group (TCG) ACPI Specification, version 1.2, revision 8 (this is the latest TCG ACPI specification at the time of writing [Tru17]), *acpi.sys* submits relevant command information to the TPM device by writing this information at a location within a memory region starting at a specific address. This address is stored in the field *Control area* ([Tru17], Table 7) of the ACPI hardware interface description table of the TPM device, named *TPM2* (TPM hardware interface and TPM2 in Figure 1).

The layout of the memory starting at the *Control area* address consists of several fields, among which are *Command size*, *Command*, *Response size*, and *Response*.[8] Relevant TPM command information is written in a memory region starting at the address stored in the field *Command*, with a size stored in the field *Command size*. Once the TPM device has finished processing the command, it returns information by storing it in a memory region starting at the address stored in the field *Response*, with a size stored in the field *Response size*. This information is then read by the drivers that are part of the TPM driver stack and passed to the issuer of the TPM command.

We now demonstrate a direct communication with the TPM through an example scenario. Through this scenario we obtained an accurate insight into how the TPM device is communicated with in a direct manner. We developed a simple application that uses the *Tbsip_Submit_Command* function of the TBS library to execute a TPM command represented by the byte sequence *0 0xc0 0 0 0 0x0a 0 0 0 0x50*. Figure 2 depicts a snippet of the application's program code for submitting the TPM command.

Using the *windbg* debugger operating in user-land, we set a breakpoint at *Tbsip_Submit_Command* to analyze the execution of this function. Figure 3 depicts relevant aspects of the function's execution. We observed that *Tbsip_Submit_Command* issues an IRP containing TPM command information using the *NtDeviceIoControlFile* Windows API function ([1] in Figure 3). *NtDeviceIoControlFile* submits IRPs to a device driver such that its first

---

[3] https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/i-o-request-packets [Retrieved: 22/9/2017]

[4] https://msdn.microsoft.com/en-us/library/ms648411(v=vs.85).aspx [Retrieved: 22/9/2017]

[5] https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/driver-stacks [Retrieved: 22/9/2017]

[6] https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/introduction-to-device-objects [Retrieved: 22/9/2017]

[7] https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/object-names [Retrieved: 22/9/2017]

[8] https://msdn.microsoft.com/de-de/library/windows/hardware/dn974551(v=vs.85).aspx [Retrieved: 22/9/2017]

```
BYTE data[10] = {0, 0xc0, 0, 0, 0, 0x0a, 0, 0, 0, 0x50};
BYTE buf[512];
UINT32 buf_len = 512;

rv = Tbsip_Submit_Command(hContext, 0, TBS_COMMAND_PRIORITY_NORMAL, data, 10, buf, &buf_len)
```

Figure 2: Submitting a TPM command using *Tbsip_Submit_Command*

parameter is a handle of the device object associated with the driver.[9] As per Microsoft's function calling convention, the first parameter of a function receiving a single or multiple integers as parameters is stored in the *rcx* register.[10] By printing out the contents of this register, we obtained the handle value *0xac* ([2] in Figure 3). We then obtained the address at which information about the object associated with this handle is stored. This address is *0xffffac883a4a7ba0* ([3] in Figure 3). This enabled us to obtain information about the driver stack consisting of the drivers processing the IRP, *tpm.sys* and *acpi.sys*, represented by the driver objects *\Driver\TPM* (i.e. the TPM driver *tpm.sys*) and *\Driver\ACPI* (i.e., the bus ACPI driver *acpi.sys*, [4] in Figure 3).

[1]

```
0:000> kc
#Call Site
00 ntdll!NtDeviceIoControlFile
01 tbs!Tbsip_Submit_Command_Internal
02 tbs!Tbsip_Submit_Command
03 TPM_SubmitCommand!main
04 TPM_SubmitCommand!invoke_main
[...]
```

[2]

```
0:000 > r rcx
rcx=00000000000000000ac
0:000 > !handle 00000000000000000ac 7
Handle ac
    Type            File
    Attrubutes      0
    GrantedAccess   0x12019f:

[...]
```

[3]

```
lkd> kd: Reading initial command '!handle 0xac 7 0n6652;q'

[...]

00ac: Object: ffffac883a4a7ba0 GrantedAccess: 0012019f (Protected) (Inherit) (Audit) Entry: ffffd888027062b0
[...]
```

[4]

```
lkd> kd: Reading initial command '!devstack 0xffffac88357ddc20;q'
  !DevObj           !DrvObj            !DevExt           ObjectName
  ffffac88357c5040  \Driver\TPM        ffffac88357c4b60
> ffffac88357ddc20  \Driver\ACPI       ffffac88352cbc60  00000027
!DevNode ffffac88357df770 :
  DeviceInst is "ACPI\SMO1200\1"
  ServiceName is "TPM"
[...]
```

Figure 3: Execution of *Tbsip_Submit_Command*

The TPM driver manages the scheduling of TPM resources and submits commands to the TPM device in a procedural manner. Figure 4 depicts some of the functions implemented in *tpm.sys*, which are involved in command processing. Figure 4 depicts a function callstack when a breakpoint we set at the *SubmitCommand* function was triggered. This function is implemented as part of the *TpmTransportMembase* data structure (*TpmTransport-*

[9] https://msdn.microsoft.com/de-de/library/windows/desktop/ms724457(v=vs.85).aspx [Retrieved: 22/9/2017]

[10] https://technet.microsoft.com/en-us/library/security/zthk2dkh(v=vs.90).aspx [Retrieved: 22/9/2017]

```
Breakpoint 10 hit
tpm!TpmTransportMemBase::SubmitCommand

[...]

#Call Site
00 tpm!TpmTransportMemBase::SubmitCommand
01 tpm!TpmTransport::DispatchCommand
02 tpm!Tpm20ResourceMgr::SubmitRequest

[...]

05 tpm!Tpm20Scheduler::SchedulerThreadWrapper
06 nt!PspSystemThreadStartup

[...]
```

Figure 4: Submission of TPM commands

MemBase::SubmitCommand in Figure 4).[11, 12]

We observed that *TpmTransportMemBase::SubmitCommand* submits commands to the TPM and it is invoked as follows. When an IRP containing a TPM command is received by the TPM driver, it schedules the creation of a thread handling the IRP in the *Tpm20Scheduler::SchedulerThreadWrapper* function. When the TPM is available for command processing, the driver triggers the submission of the TPM command in the *Tpm20Scheduler::SubmitRequest* function. The actual submission of the command to the TPM is done by *TpmTransport::DispatchCommand*; that is, it invokes *TpmTransportMemBase::SubmitCommand*.

As previously mentioned, the ACPI driver *acpi.sys* passes command information to the TPM by storing the information in the memory region starting at the address specified by the *Control area* field of the *TPM2* table. Figure 5 depicts the contents of this table as presented by the *RW* utility. We observed that the value stored in *Control area* is not zero. According to the TCG ACPI Specification, version 1.2, revision 8 ([Tru17], Table 7), this indicates that the memory region starting at the address stored in this field is used as previously described.

```
TPM 2.0 Hardware Interface Table: 0x000000009CDD2000

54 50 4D 32 34 00 00 00 03 DA 4C 45 4E 4F 56 4F TPM24.....LENOVO
54 50 2D 4A 42 20 20 20 90 11 00 00 50 54 45 43 TP-JB   ....PTEC
02 00 00 00 00 00 00 00 00 F0 DF 9C 00 00 00 00 ................
02 00 00 00                                     ....

Signature        "TPM2"
Length           0x00000034 (52)
Revision         0x03 (3)
Checksum         0xDA (218)
OEM ID           "LENOVO"
OEM Table ID     "TP-JB   "
OEM Revision     0x00001190 (4496)
Creator ID       "PTEC"
Creator Revision 0x00000002 (2)
Flags            0x00000000
Control Area     0x000000009CDFF000
Start Method     0x00000002 (2) - Uses an ACPI Start method
```

Figure 5: The ACPI *TPM2* table

After command information is passed to the TPM, it starts processing the command. The procedure of command procession is described in ([Tru16b], Section 5). As part of this procedure, the processed command is authorized by evaluating the provided authorization value (see [Tru16a]; Section 19 on authorization of TPM commands).

[11] In this work, we use the scope operator *::* when referring to functions declared as part of data structures.

[12] There are several different implementations of the *SubmitCommand* function, which are implemented as part of data structures different than *TpmTransportMembase*. We set breakpoints to these functions using the *windbg* debugger, observing that they are not invoked during regular system operation.

In addition, the procedure defines the behavior of the TPM in different authorization scenarios. This involves, for example, increasing the count of failed TPM authorization attempts if the authorization fails.

2.2 Abstracted TPM communication

Windows 10 provides the Cryptography API: Next Generation (CNG) library, first introduced in Windows Vista, for abstracting the functionalities of the TBS library. The functions implemented as part of the CNG library act as wrappers of functions of the TBS library, adding functionalities and making their use easier.[13]

CNG uses the concept of cryptographic providers, where providers are entities performing cryptographic operations (e.g., hashing, digital signature verification).[14] These entities may be implemented in software, hardware, or both. There are two main types of CNG providers: algorithm and key storage providers. The former are primarily used for performing basic cryptographic operations, such as hashing and signing,[15] whereas the latter are primarily used for performing key operations, such as creating and storing keys.[16]

CNG abstracts the TPM device in the form of a hardware-implemented cryptographic key storage and algorithm provider, referred to as the *Platform Cryptographic Provider*.[17] Microsoft's basic software-implemented cryptographic provider is referred to as the *Microsoft Primitive Provider*.[18]

The majory of the functions implemented as part of CNG are implemented in the *%SystemRoot%\System32\bcrypt.dll* and *%SystemRoot%\System32\ncrypt.dll* library files. The library files *%SystemRoot%\System32\PCPks.dll* and *%SystemRoot%\System32\PCPTPM12.dll* implement CNG functionalities related to the TPM (CNG TPM Implementations in Figure 1). These may invoke functions implemented as part of the TBS library.

The access and use of cryptographic provider functionalities, including those of the *Platform Cryptographic Provider*, is managed by CNG routers. For example, access to the key storage functionalities of the *Platform Cryptographic Provider* is managed by the CNG key storage router implemented in *ncrypt.dll*.[19]

In order to verify the use of the TPM when the CNG library is utilized, we developed a simple application creating an array of random data using the *Platform Cryptographic Provider*. Figure 6 depicts a snippet of the application's program code, where the *BcryptOpenAlgorithmProvider* function is used for loading and initializing this provider.[20]

We set a breakpoint at *BcryptOpenAlgorithmProvider*. We observed that it dynamically loads the CNG TPM implementations (i.e., the library files *PCPks.dll* and *PPCPTPM12.dll*) and the TBS library (i.e., the library file *tbs.dll*); see Figure 7. We also observed that TPM command execution is performed by invoking the *Tbsip_Submit_Command* function of the TBS library (see paragraph 'Direct TPM communication').

3 TPM Communication Interfaces: Kernel-land

The components of the Windows 10 system deployed in kernel-land can communicate with the TPM by invoking functions implemented in the TPM export driver (see Figure 1). Export drivers are kernel-mode library files exporting routines to the kernel or other drivers.[21] The TPM export driver is implemented in *%System-*

[13] https://msdn.microsoft.com/de-de/library/windows/desktop/aa376210%28v=vs.85%29.aspx [Retrieved: 22/9/2017]

[14] https://msdn.microsoft.com/en-us/library/windows/desktop/bb931380(v=vs.85).aspx [Retrieved: 22/9/2017]

[15] https://msdn.microsoft.com/en-us/library/windows/desktop/bb931354(v=vs.85).aspx [Retrieved: 22/9/2017]

[16] https://msdn.microsoft.com/en-us/library/windows/desktop/bb931355(v=vs.85).aspx [Retrieved: 22/9/2017]

[17] https://msdn.microsoft.com/en-us/library/windows/hardware/hh998513(v=vs.85).aspx [Retrieved: 22/9/2017]

[18] https://msdn.microsoft.com/en-us/library/windows/desktop/aa375479(v=vs.85).aspx [Retrieved: 22/9/2017]

[19] https://msdn.microsoft.com/en-us/library/windows/desktop/bb204778(v=vs.85).aspx [Retrieved: 22/9/2017]

[20] https://msdn.microsoft.com/de-de/library/windows/desktop/aa375479(v=vs.85).aspx [Retrieved: 22/9/2017]; the function's third parameter with a value of *MS_PLATFORM_CRYPTO_PROVIDER* specifies the *Platform Cryptographic Provider*.

[21] https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/creating-export-drivers [Retrieved: 22/9/2017]

```
[...]

if (!NT_SUCCESS(status = BCryptOpenAlgorithmProvider(
    &hAlgorithm,
    BCRYPT_RNG_ALGORITHM,
    MS_PLATFORM_CRYPTO_PROVIDER,
    0
)))

[...]
```

Figure 6: Loading and initializing the Platform Cryptographic Provider

```
bcrypt!BCryptOpenAlgorithmProvider+0x1a7:
00007ff9`78ee4407 e814d0ffff      call    bcrypt!LoadProviderEx (00007ff9`78ee1420)
0:000> p
ModLoad: 00007ff9`78740000 00007ff9`78768000   C:\Windows\system32\PCPKsp.dll
ModLoad: 00007ff9`7b210000 00007ff9`7b2ae000   C:\Windows\System32\msvcrt.dll
[...]
ModLoad: 00007ff9`78660000 00007ff9`786fd000   C:\Windows\system32\PCPTPM12.dll
ModLoad: 00007ff9`7cc20000 00007ff9`7ccc2000   C:\Windows\System32\advapi32.dll
ModLoad: 00007ff9`78650000 00007ff9`7865d000   C:\Windows\SYSTEM32\tbs.dll
```

Figure 7: Dynamic loading of TPM-related library files

Root%\System32\drivers\tbs.sys. It represents the kernel-mode implementation of the TBS library; that is, it implements the same functions as this library, modified for operation in kernel-mode. For example, instead of issuing a system call using the *NtDeviceIoControlFile* function (see Figure 3), which can be invoked only from user-mode, the *Tbsip_Submit_Command* function implemented in the TPM export driver issues IRPs by invoking the *ZwDeviceIoControlFile* function. *ZwDeviceIoControlFile* is the kernel-mode counterpart of *NtDeviceIoControlFile*.[22]

3.1 TPM Usage Profiles

In Section 2 and Section 3, we observed that TPM commands are sent in the form of IRPs to the TPM driver *tpm.sys* using the *NtDeviceIoControlFile* or the *ZwDeviceIoControlFile* function. We aim at automating the collection of information identifying user processes or the kernel communicating with the TPM. We also aim at collecting relevant related information, such as communication patterns and frequencies. We refer to this information as TPM usage profile and developed a script to gather it (see Appendix, section 'TPM Usage Profiler').

Once a breakpoint at *NtDeviceIoControlFile* or *ZwDeviceIoControlFile* is triggered, the script identifies the target driver of the IRP. This is based on the handle value passed as the first parameter of *NtDeviceIoControlFile* or *ZwDeviceIoControlFile*. In addition, the script displays relevant information, such as:

- timestamp information on the invocation of *NtDeviceIoControlFile* or *ZwDeviceIoControlFile*;
- the driver stack of the driver to which an IRP is being sent;
- the process ID (PID), name, and command parameters on the user process (if any) sending an IRP to the driver;
- the name of the driver object associated with the driver to which an IRP is being sent.

Since the script provides relevant information at the ingress points to the TPM driver *tpm.sys* (i.e., the functions *NtDeviceIoControlFile* and *ZwDeviceIoControlFile*), it enables the construction of comprehensive TPM usage pro-

[22] https://msdn.microsoft.com/en-us/library/windows/hardware/ff566441(v=vs.85).aspx [Retrieved: 22/9/2017]

files. The output of the script can be stored into a file using the *.logopen* and *.logclose windbg* commands for subsequent parsing and constructing TPM usage profiles. For example, in the Appendix, section 'TPM Usage', we provide a table listing user-land executables (column 'Executable') that submit commands to the TPM until a user is presented with the login screen at system booting. This table also presents relevant parameters (column 'Parameters') and the name of the entity implemented in the executable (column 'Entity'). We emphasize that the executable list presented in the Appendix is specific for the platform where the Windows 10 system was installed on. This list may differ for other platforms, depending on their configurations, for example, configurations enabling BitLocker.[23]

[23] https://docs.microsoft.com/en-us/windows/device-security/bitlocker/bitlocker-overview [Retrieved: 22/9/2017]

Appendix

TPM Usage Profiler

This script can also be found in the folder *files* of the *Windows Insight* file repository, under the name *windbg_-handle_drivername.wds*.

```
$$ ** Script usage: [function breakpoint] "$$>a<[path_to_script_file] [System address]"
$$ [function breakpoint]: a breakpoint to a single or multiple *DeviceIoControlFile functions
$$ (e.g., 'bu ntdll!NtDeviceIoControlFile', bm /a nt!*DeviceIoControlFile)
$$ [path_to_script_file]: path to this script file
$$ [System address]: the address of the EPROCESS structure of the kernel System thread. It can be obtained by issuing '!process 4 0'


.echotimestamps 1

r? $t18 = @rcx & 0xfffffffffffffffc

.if ( (@$t18 & 0x80000000) == 0x80000000)
{
        r? $t0 = ((_EPROCESS*)${$arg1})->ObjectTable
}
.else
{
        r? $t0 = @$proc->ObjectTable
}

r? $t1 = @$t0->TableCode

r? $t19 = @$t1 & 0x3

r? $t1 = @$t1 & (~0x3)

.if (@$t19 == 0)
{
        r? $t3 = @$t1 + (4*(@$t18&0x3fc))
}

.if (@$t19 == 1)
{

        r? $t3 = ((unsigned int64*)(@$t1 + @$ptrsize*(((@$t18&0x3fc00))>>10)))[0] + 4*((@$t18&0x3fc))
}


.if (@$t19 == 2)
{

        r? $t17 = ((unsigned int64*)(@$t1 + @$ptrsize*(((@$t18&0x3fc0000))>>18)))[0]
        r? $t3 = ((unsigned int64*)(@$t17 - 0x1 + @$ptrsize*(((@$t18&0x3fc00))>>10)))[0] + 4*((@$t18&0x3fc))
}

r? $t4 = (((_HANDLE_TABLE_ENTRY*)@$t3) -> ObjectPointerBits << 4) | 0xffff000000000000

r? $t4 = @$t4 + 0x30

r? $t5 = ((_FILE_OBJECT*)@$t4) -> DeviceObject

r? $t6 = ((_DEVICE_OBJECT*)@$t5) -> DriverObject

r? $t7 = (unsigned int64)@$t6 + 0x38

r $t8 = poi(@$t7 + 0x008)

.if ( (@$t18 & 0x80000000) == 0x80000000)
{
        .printf "****************\n";
        .printf "Image/Command: Kernel\n"
        .printf "Driver associated to IRP-ed device: %mu\n", @$t8
        !devstack @$t5
        .printf "****************\n";
}

.else
{
        .printf "****************\n";
        r? $t15 =((unsigned int64*) ((unsigned int64)(&((@$proc->Peb)->ProcessParameters)->CommandLine ) + 0x008 ) )[0]
        .printf "Image/Command: %mu\n", @$t15
        r? $t15 =  (unsigned int64)(@$proc->UniqueProcessId)
        .printf "PID: %d\n", @$t15
        .printf "Driver associated to IRP-ed device: %mu\n", @$t8
        !devstack @$t5
        .printf "****************\n";
}
g
```

TPM Usage

Executable	Parameters	Entity
smss.exe		Session manager
lsass.exe		Local security authority
svchost.exe	*-k netsvcs*	BitLocker Drive Encryption Service
taskhostw.exe		Host process for Windows tasks
svchost.exe	*-k netsvcs*	Microsoft Account Sign-in Assistant

References

[Tru16a] Trusted Computing Group (TCG). Trusted Platform Module Library Part 1: Architecture. 2016. https://trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-2.0-Part-1-Architecture-01.38.pdf.

[Tru16b] Trusted Computing Group (TCG). Trusted Platform Module Library Part 3: Commands. 2016. Family 2.0, Level 00, Revision 01.38; https://trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-2.0-Part-3-Commands-01.38.pdf.

[Tru17] Trusted Computing Group (TCG). TCG ACPI Specification. 2017. Family 1.2 and 2.0, Version 1.2, Revision 8; https://trustedcomputinggroup.org/wp-content/uploads/TCG_ACPIGeneralSpecification_v1.20_r8.pdf.