Federal Office
for Information Security

Work Package 5: Trusted Platform Module and Unified Extensible Firmware Interface “Secure Boot”

Version: 1.1

Federal Office for Information Security
Post Box 20 03 63
D-53133 Bonn
Phone: +49 22899 9582-0
E-Mail: bsi@bsi.bund.de
Internet: https://www.bsi.bund.de

Table of Contents

1 Introduction 7

1.1 Zusammenfassung 7

1.2 Executive Summary 11

1.3 General Concepts and Terminology 13

2 Technical Analysis of Functionalities 16

2.1 TPM Communication Interfaces 16

2.1.1 TPM Communication: User-land 17

2.1.2 TPM Communication: Kernel-land 22

2.1.3 TPM Usage Profiles 22

2.2 Windows Boot 23

2.2.1 Boot Manager 24

2.2.2 Windows Loader 27

2.2.3 Windows Kernel 29

2.3 The Windows Defender ELAM Driver 36

2.4 Integrity Measurement 40

2.5 TPM Provisioning 46

2.5.1 Manual Provisioning 47

2.5.2 Auto-provisioning 51

2.6 Security Aspects 53

3 Configuration and Logging Capabilities 55

3.1 Configuration Capabilities 55

3.1.1 Programmatic Configuration Capabilities 56

3.1.2 Non-programmatic Configuration Capabilities 57

3.1.3 Recommended Configuration Settings 62

3.2 Logging Capabilities 63

Appendix 68

Tools 68

TPM Usage Profiler 69

TPM Usage 70

ELAM Database Parser 70

WBCL Parser 73

Measured Executables 73

Reference Documentation 76

Keywords and Abbreviations 77

Figures

Figure 1: A TPM key hierarchy 14
Figure 2: Interfaces for communicating with the TPM 16
Figure 3: Submitting a TPM command using Tbsip_Submit_Command 18
Figure 4: Execution of Tbsip_Submit_Command 19
Figure 5: Submission of TPM commands 20
Figure 6: The ACPI TPM2 table 20
Figure 7: Loading and initializing the Platform Cryptographic Provider 21
Figure 8: Dynamic loading of TPM-related library files 21
Figure 9: The booting process of a Windows-based platform 23
Figure 10: Public keys stored in the boot manager and the UEFI firmware 25
Figure 11: Function stack: Image loading and integrity verification by the boot manager 26
Figure 12: A public key of a root certificate stored in the boot manager 27
Figure 13: Function stack: Image loading and integrity verification by the Windows loader 28
Figure 14: Function stack: Image loading and integrity verification by the Windows kernel 29
Figure 15: Snippet of a Windows compatibility database file 31
Figure 16: Pseudo-code of CipValidateFileHash and functions it invokes 32
Figure 17: An Authenticode signature and a PE hash value 33
Figure 18: A hash code and a list of signers 34
Figure 19: crashdmp.sys: The ContentInfo section and the PE hash value 35
Figure 20: A public key of a root certificate stored in ci.dll 36
Figure 21: Information on the Windows Defender ELAM driver 36
Figure 22: Pseudo-code of MpEbBootDriverCallback of the Windows Defender Implementation and functions that it invokes 37
Figure 23: The ELAM signature database in the registry 38
Figure 24: The ELAM signature database 39
Figure 25: Pseudo-code of EbAuthenticateSignatureData 39
Figure 26: The public key used for verification of the Windows Defender ELAM signature database 40
Figure 27: The architecture of the integrity measurement mechanism of Windows 10 41
Figure 28: An excerpt of a WBCL file 43
Figure 29: Integrity measurement in the Windows loader 46
Figure 30: The TPM provisioning process 48
Figure 31: A random value generated by TpmApiGetRandom (Base64-encoded) 49
Figure 32: The third parameter of CtpmSettingsReaderWriter::WriteStringSetting 49
Figure 33: The first and fifth parameter of CtpmSettingsReaderWriter::WriteStringSetting 49
Figure 34: BCryptOpenAlgorithmProvider in TpmApiGetRandom 50
Figure 35: A TPM command sequence for generating an SRK 50
Figure 36: taskhostw.exe changing the OwnerAuthFull registry key 52
Figure 37: The TPM-Maintenance task 52
Figure 38: Generation of an AIK 53
Figure 39: The main window of the TPM management utility 60
Figure 40: TPM command management using the TPM management utility 60
Figure 41: The registry hive HKEY_LOCAL_MACHINE\Software\Microsoft\Tpm 62
Figure 42: A GUID of an ETW provider declared in tbs.sys 64
Figure 43: Registration and deregistration of the provider with GUID 61d3c72e-6b1b-454c-a34d-b39eb95b8d99 67
Figure 44: A snippet of a log file containing TPM command information 67

Tables

Table 1: Roots of trust and TPM usage 24
Table 2: A mapping between PCR indexes and types of measurement events 45
Table 3: Attack scenarios and mitigations 54
Table 4: Activities supported by TPM configuration capabilities 55
Table 5: Relevant methods of the Win32_Tpm class 57
Table 6: Get-Tpm cmdlet: Properties 58
Table 7: Get-TpmEndorsementKeyInfo: Properties 59
Table 8: Group policies for configuring the TPM 61
Table 9: Recommended settings 63
Table 10: ETW providers for logging TPM events 64
Table 11: Event IDs generated by the ETW provider TPM 65
Table 12: Event IDs generated by the ETW provider Microsoft-Windows-TPM-WMI 66

1 Introduction

1.1 Zusammenfassung

Dieses Kapitel stellt das Ergebnis von Arbeitspaket 5 des Projekts „SiSyPHuS Win10: Studie zu Systemaufbau, Protokollierung, Härtung und Sicherheitsfunktionen in Windows 10“ dar. Das Projekt wird durch die Firma ERNW GmbH im Auftrag des Bundesamt für Sicherheit in der Informationstechnik (BSI) durchgeführt.

Ziel dieses Arbeitspakets ist die Analyse (i) der Interaktionen zwischen dem Microsoft Windows 10 Betriebssystem und einem Trusted Platform Module (TPM); (ii) der Rolle des TPM im Windows Bootprozess (im weiteren Verlauf als *Windows Boot* bezeichnet); und (iii) die Konfigurations- und Protokollierungsmöglichkeiten (auch als *logging* bezeichnet) in Bezug auf das TPM. Wie durch das BSI vorgegeben wird der TPM-Standard 2.0 und Windows 10 Build 1607, 64-bit, *long-term servicing branch* (LTSB), Deutsch betrachtet.

Die wesentlichen Inhalte dieser Arbeit sind:

- Die Analyse der Schnittstelle zwischen Windows 10 und dem TPM: Wir diskutieren das Zusammenspiel der unterschiedlichen Komponenten von Windows 10 in Userland (*System Support* Prozesse, Anwendungen, Dienste, das *Windows subsystem* und `ntdll.dll`) und Kernelland (der Betriebssystemkern und Treiber) mit dem TPM. Neben der Beschreibung der relevanten Schnittstellen wurde ein Skript entwickelt, das die TPM-Nutzung einzelner Komponenten aufzeichnet. Das Skript und exemplarische Aufzeichnungen (die TPM-Nutzungsprofile genannt werden) sind in diesem Dokument enthalten.
- Eine Analyse des Bootprozesses von Windows 10: Wir untersuchen den Windows 10 Bootprozess und beschreiben, wann und wie das TPM während des Bootprozesses genutzt wird. Das TPM stellt eine in Hardware implementierte *Root of Trust* dar, die für sensitive Operationen genutzt wird, wie etwa die sichere Speicherung von Schlüsselmaterial zur Verschlüsselung. Aufgrund dessen wird die Umsetzung der Integritätsverifikation als Teil des Bootprozesses analysiert, um die Rolle des TPMs darin bewerten zu können.
  Die logische Root of Trust wurde in der *Unified Extensible Firmware Interface* (UEFI) Firmware identifiziert. Während Firmware vor UEFI (wie bspw. das *Basic Input Output System* (BIOS) keine Verifikation im Boot-Prozess vornahm, enthält die UEFI Firmware Mechanismen zur Verifikation der folgenden Boot-Komponente. Die sichere Validierung der Boot-Komponenten des Betriebssystems stellt damit eine notwendige Voraussetzung für einen validierten Betriebssystem-Bootprozess dar; eine genaue Untersuchung der UEFI Firmware mit Hinblick auf sichere Validierung ist nicht im Fokus dieses Arbeitspakets. Die UEFI Firmware wurde dennoch als initiale Root of Trust des Boot-Prozesses identifiziert; die Verifikation der weiteren Boot-Komponenten basiert auf Roots of Trust die direkt in die beteiligten Komponenten integriert sind. Das TPM spielt demnach keine Rolle im Aufbau eines authentifizierten Boot-Prozesses (abgesehen von Funktionalität für die Treiber-Verifikation, die im nächsten Abschnitt beschrieben ist).
- Die Untersuchung der Anti-Schadsoftware-Mechanismen von Windows 10: Wir betrachten die e*arly launch anti-malware* (ELAM)-Technologie, welche Teil des Bootprozesses ist und Treiber auf Schadsoftware untersucht. ELAM ist hierbei selbst als Treiber (der sogenannte ELAM-Treiber) implementiert und ist für die Sicherheit des Systems von großer Bedeutung. Wir haben die Entscheidungskriterien für die Kategorisierung eines Treibers (als gutartig oder schadhaft) sowie die Struktur der hierzu genutzten Schadsoftware-Datenbank und die Überprüfung der Integrität dieser Datenbank analysiert und dokumentiert.
- Eine Untersuchung des in Windows 10 implementierten Provisionierungsprozesses des TPMs: Wir identifizieren und analysieren die relevanten Schritte des Provisionierungsprozesses, der für die Initialisierung des TPMs zuständig ist, so dass dieses genutzt werden kann. Wir betrachten sowohl den

Fall, in dem die Provisionierung automatisch durch Windows 10 angestoßen wird, also auch den Fall, in dem der Benutzer dies selbst veranlasst.

Die Arbeit lässt sich wie folgt zusammenfassen:

- Abschnitt 1.3 führt relevante Konzepte und Begriffe ein, welche zum besseren Verständnis dieses Kapitels beitragen. Das sind zum Beispiel root of trust for measurement (RTM), root of trust for storage (RTS) und root of trust for reporting (RTR).
- Abschnitt 2 liefert technische Informationen über die in diesem Arbeitspaket analysierten Funktionalitäten von Windows 10:
  - Abschnitt 2.1 liefert eine Übersicht über die Schnittstelle zwischen dem Betriebssystem und dem TPM, wobei Abschnitt 2.1.1 und Abschnitt 2.1.2 jeweils die Kommunikationsschnittstellen zwischen dem TPM und den Userland-/Kernelland-Komponenten von Windows 10 beschreiben. Weiterhin werden die entwickelten Methoden für die Enumeration von Komponenten, die das TPM nutzen, vorgestellt. Diese Methoden können automatisiert auf neue Windows-Versionen angewendet werden; die identifizierten Komponenten sind im Anhang dargestellt. Zusammengefasst können die Userland-Komponenten auf zwei Arten mit dem TPM kommunizieren:
    - Direkter Zugriff: Die direkte Kommunikation mit dem TPM beinhaltet die Ausführung von Funktionen, die als Teil des TPM Basis Dienstes (*TPM Base Services*, TBS) in der Datei `tbs.dll` deklariert werden. Die meisten dieser Funktionen führen Operationen dadurch aus, dass sie TPM Kommandos in Form einer Bytesequenz in einem Puffer konstruieren und an das TPM übergeben. Die Übergabe von Kommandos von der TBS Bibliothek an das TPM beinhaltet die Ausführung eines System-Aufrufs (*system call*) an den TPM-Treiber, welcher in der Datei `tpm.sys` implementiert ist.
    - Abstrahierter Zugriff: Windows 10 stellt die *Next Generation Cryptography API* (*application programming interface*, CNG API) bereit, um Funktionalität der TBS-Bibliothek zu abstrahieren und deren Benutzung zu vereinfachen. Die CNG abstrahiert das TPM-Gerät in Form eines in Hardware implementieren Kryptografie-Providers, der als `Platform Cryptographic Provider` bezeichnet wird.

    Windows 10 Kernelland Komponenten (der Betriebssystemkern selbst und Treiber) können mit dem TPM kommunizieren, indem sie Funktionen des TPM Treibers `tbs.sys` aufrufen.

    Abschnitt 2.1.3 enthält Quellcode für die Erstellung von TPM-Nutzungsprofilen (die unter anderem TPM-Zugriffszeiten sowie die zugreifenden ausführbaren Dateien auflisten).
  - Abschnitt 2.2 beschreibt die Analyse des Bootprozesses von Windows 10: Abschnitt 2.2.1, Abschnitt 2.2.2 und Abschnitt 2.2.3 erörtern die Teilaktivitäten dieses Prozesses, wie beispielsweise die Integritätsprüfung, im Kontext der drei in den Bootprozess involvierten Komponenten: Bootmanager, Windows Loader und Betriebssystemkern.

    Diese drei Komponenten überprüfen die Integrität von auszuführendem Binärcode indem sie dessen digitale Signatur überprüfen. Diese Überprüfung findet anhand eines öffentlichen kryptografischen Schlüssels statt, der Teil eines von Microsoft ausgestellten Root Zertifikats ist. Diese Zertifikate sind fest im Bootmanager, dem *Windows loader* und dem Betriebssystemkern in der sogenannten `root table` hinterlegt.

    Das TPM wird hierbei für Integritätsmessungen und BitLocker-Operationen, wie beispielsweise das Auslesen von im TPM gespeicherten kryptografischen Schlüsseln, genutzt. Die zur Integritätsprüfung durch die drei zuvor genannten Komponenten benötigten Hashwertberechnungen sind hingegen in Software implementiert.
  - Abschnitt 2.3 fokussiert sich auf die in Windows 10 eingesetzte ELAM-Technologie, d.h. den ELAM Treiber, den durch diesen implementierte Klassifizierungsprozess in gutartige und schadhafte Treiber sowie seine Schadsoftwaredatenbank.

Der ELAM-Treiber ist Teil des sogenannten Windows Defenders in der Datei `WdBoot.sys`. Dieser Treiber überprüft jeden Boot-Treiber und kategorisiert ihn als bekanntermaßen gutartig (*known good*), bekanntermaßen schadhaft (*known bad*), bekanntermaßen schadhaft und für den Bootprozess notwendig (*known bad image of a boot-critical driver*, d.h. der Treiber wird als schadhaft eingestuft, wird aber dennoch für den Bootprozess benötigt), oder unbekannt (*unknown*). Basierend auf dieser Kategorisierung entscheidet der Betriebssystemkern, ob der einzelne Boot-Treiber initialisiert wird oder nicht. Der Mechanismus wirkt wie vorgesehen; eine Evaluation auf quantitative Malware-Erkennung wurde nicht durchgeführt. Die Validierung der Treiber-Signatur wird dadurch um einen Mechanismus ergänzt, der die kurzfristige Reaktion auf bösartige Treiber mit gültiger Signatur ermöglicht.

Die Schadsoftware-Signaturdatenbank des ELAM-Treibers ist digital signiert, um sie vor unautorisierter Veränderung zu schützen. Der öffentliche Schlüssel, der für diese Verifikation herangezogen wird, ist in der Treiber-Datei `WdBoot.sys` fest hinterlegt. Insofern ist der genannte Treiber selbst die Vertrauensbasis für die Überprüfung der Schadsoftware-Signaturdatenbank.

- Abschnitt 2.4 liefert eine Übersicht über die Mechanismen zur Integritätsmessung in Windows 10. Konkret werden Informationen über die im TPM gespeicherten Messdaten, die Implementierung der Integritätsmessung an sich und die Nutzung der Messdaten zur Verifikation der Integrität des gesamten Systems gegeben.

  Die Integritätsmessung berechnet bei jedem Systemstart kryptografische Hashwerte relevanter Objekte (beispielsweise ausführbarer Dateien oder Bytesequenzen) als Messdaten. Auch die Speicherung dieser und weiterer relevanter Daten im TPM und Protokolldateien für spätere Analysen wird durch die Komponente zur Integritätsmessung durchgeführt. Die Analyse der Messdaten wird normalerweise durch eine vertrauenswürdige dritte Entität (*trusted remote platform*) durchgeführt.

  Die Daten werden in einem Kontext gespeichert, der als *Windows Boot Configuration Log* (WBCL) bezeichnet wird. Ein neues WBCL wird bei jedem Systemstart generiert wenn neue Messdaten berechnet werden. Die Plattform archiviert jedes WBCL in einer Protokolldatei (die WBCL-Datei), welche im Verzeichnis `%SystemRoot%\Logs\MeasuredBoot` gespeichert wird.

- Abschnitt 2.5 beschreibt die wichtigsten Schritte bei der Provisionierung des TPMs in Windows 10: Abschnitt 2.5.1 beschreibt den durch Windows ausgelösten automatisierten Provisionierungsprozess und Abschnitt 2.5.2 beschreibt analog die Schritte bei der manuellen, durch den Benutzer initiierten Provisionierung.

  Unter Provisionierung wird hier die Ablage von Daten im TPM verstanden, welche zur Nutzung des TPMs benötigt werden. Dies beinhaltet beispielsweise die Erzeugung und Speicherung von *authorization values*, *endorsement key* (EK) und storage root key (SRK). Weiterhin wird unter Windows ein *authorization value*, *owner authorization value2*angelegt, der das zentrale Authentifizierungsmerkmal für die Verwaltung des TPMs darstellt (d.h. dieser Wert wird wird als Schlüssel vor der Ausführung von TPM Verwaltungsoperationen abgefragt). Der EK ist ein Schlüssel, der für jedes TPM einmalig ist und das daher TPM und das ganze System eindeutig identifiziert. Der SRK wird genutzt, um Objekte dadurch zu schützen, dass sensitive Bereiche (wie beispielsweise Teile eines privaten Schlüssels) mit einer symmetrischen Verschlüsselungsmethode unter Verwendung eines Schlüssels verschlüsselt werden, der aus dem privaten Teil des SRK abgeleitet wird.

  Es wurde festgestellt, dass bei der Provisionierung der neue *owner authorization value* im Userland generiert wird und dann an den TPM-Treiber in Firm eines input/output (I/O) request packets (IRPs) weitergereicht wird. Dieser Treiber veranlasst dann die Speicherung des Wertes im TPM. Die Generierung des owner authorization value im Userland ist dabei nicht als Sicherheitsproblem zu werten, da der entsprechende Wert auch als Nutzer-Passwort für das TPM dient und damit notwendigerweise im Userland verarbeitet werden muss.

    - Abschnitt 2.6 gibt einen Überblick über relevante Sicherheitsaspekte; Dies beinhaltet einen Überblick über Angriffsszenarien, die während der Analyse berücksichtigt wurden, sowie implementierte Maßnahmen.

- Abschnitt 3 enthält eine Übersicht über die Möglichkeiten zur Konfiguration und Protokollierung in Bezug auf die Verwaltung des TPMs durch Windows 10 und die Protokollierung von TPM-Ereignissen. Abschnitt 3.1 diskutiert die Schnittstellen, die Windows 10 zur Konfiguration des TPMs bietet. Die sind zum einen programmatische Konfigurationsmöglichkeiten und zum anderen nicht-programmatische. In diesem Abschnitt werden als programmatische Optionen die TPM Windows Management Instrumentation (WMI) Schnittstellen und die TBS API erläutert. Als nicht-programmatische Konfigurationsmöglichkeiten werden die PowerShell, das sogenannte TPM Management Utility (`tpm.msc`), Gruppenrichtlinien und die Registry betrachtet. Abschnitt 3.2 erläutert die Benutzung der Basis-Protokollierungsfunktionen von Windows 10 (nämlich die in [ERNW WP2] betrachteten EventLog und Event Tracing for Windows, ETW) zur Protokollierung von TPM-Ereignissen. Es wurden die in den Software-Bibliotheken und Anwendungen, welche in Bezug auf das TPM relevant sind, enthaltenen Logging-Funktionalitäten untersucht. Hierbei wurden 5 ETW Provider identifiziert, welchen in diesem Abschnitt mit ihrer *global unique identifier* (GUID) und den jeweiligen Dateinamen tabellarisch aufgelistet werden.

- Der Appendix gliedert sich wie folgt: Der Abschnitt *Tools* im Appendix gibt eine Übersicht über relevante Software-Werkzeuge und wo diese bezogen werden können. Der Abschnitt *TPM Usage Profiler* dokumentiert ein während der Bearbeitung dieses Arbeitspakets erstelltes Script, welches zur Erstellung von TPM-Nutzungsprofilen eingesetzt werden kann. Der Abschnitt *TPM Usage* enthält ein zugehöriges Ergebnis des *TPM Usage Profilers* von einem exemplarischen Windows System. Dieser Abschnitt enthält eine Liste ausführbarer Dateien (Systemkomponenten), die mit dem TPM kommunizieren. Im Verlauf dieser Arbeit haben wir BitLocker als einzige Windows-Komponente identifiziert, die das TPM während des Systemstarts für kryptografische Operationen aktiv verwendet. BitLocker kann auch ohne TPM funktionieren. Windows selbst verwendet das TPM während des Systemstarts zum Speichern von Integritätsmessungen. Im Rahmen dieser Arbeit haben wir keine anderen Windows-Komponenten und -Verfahren beobachtet (z. B. den ELAM-Treiber), die das TPM für kryptografische Operationen oder sicheren Speicher verwenden. Der Abschnitt ELAM *Database Parser* liefert ein Script, das zum Parsen der Schadsoftware-Datenbank von ELAM genutzt werden kann. Der Abschnitt *Measured Executables* gibt eine Übersicht über die Namen der ausführbaren Dateien, deren Integrität von Windows 10 gemessen wird.

1.2 Executive Summary

This chapter implements the work plan outlined in Work Package 5 of the project "SiSyPHuS Win10: Studie zu Systemaufbau, Protokollierung, Härtung und Sicherheitsfunktionen in Windows 10" (orig., ger.), contracted by the German Federal Office for Information Security (orig., ger., Bundesamt für Sicherheit in der Informationstechnik (BSI)). The work planned as part of Work Package 5 has been conducted by ERNW GmbH in the time period between July and September 2017, in accordance with the time plan agreed upon by ERNW GmbH and the German Federal Office for Information Security.

The objective of this work package is the analysis of the: (i) interactions between the Windows 10 operating system that is subject of analysis and the Trusted Platform Module (TPM); (ii) the role that the TPM plays in activities of the operating system, with a focus on the booting process (which we refer to as Windows boot); and (iii) the configuration and logging capabilities of the TPM. As required by the German Federal Office for Information Security, the TPM standard in focus is that of version 2.0. The exact release of the Windows 10 system in focus is build 1607, 64-bit, long-term servicing branch (LTSB), German language. The core contributions of this work are:

- An analysis of the interfaces between Windows 10 and the TPM: We discuss how the different parts of Windows 10 deployed in user-land and in kernel-land use the TPM. We focus on the communication interfaces between these parts of Windows 10 and the TPM. We also provide the means for identifying Windows 10 components that use the TPM as well as a list of such components we identified;
- An analysis of the booting process of Windows 10: We analyze the booting process of the Windows operating system and we provide an overview of the use of the TPM during this process. Given that the TPM is a component representing a hardware-implemented root of trust for sensitive operations, we analyze the implemented integrity verification procedures conducted as part of the booting process of Windows 10. When analyzing these procedures, we identify the UEFI firmware as the root of trust of the overall booting process. Firmware is low-level software enabling the functioning of hardware devices. Legacy, non-UEFI firmware is not designed to play any role in securing the booting process of systems. However, the UEFI firmware extends the trust chain securing the booting process of Windows serving as the first root of trust in this chain. Detailed analysis on the role of UEFI in the booting process of Windows is out of the scope of this work package. The roots of trust for the further booting process were identified to be included in the relevant binaries. The TPM does not play a role in establishing an authenticated boot chain except for driver verification as described in the next paragraph;
- An analysis of the anti-malware mechanism of Windows 10: We discuss the early launch anti-malware (ELAM) technology implemented in Windows 10. This technology, implemented in a driver referred to as the ELAM driver, is used for checking drivers for malware and is part of the booting process of Windows 10. Therefore, it is critical for the security of the boot process of the system. We provide an overview of the decision-making process of the ELAM driver for categorizing an executable as benign or malicious. We also provide an overview of the structure of the malware database that this driver uses, and of the verification of the integrity of this database;
- An analysis of the workflow and implementation of the integrity measurement mechanism of Windows 10: We provide an overview of the integrity measurement mechanism of Windows 10. This includes discussions on the contents and structure of integrity measurement data stored in the TPM, how integrity measurement is implemented in Windows 10, and how measurement data is used for verifying the integrity of a given platform. We also discuss on how actual measurements stored in the TPM can be viewed. In addition, we provide information on the different types of measurement data and what concrete executables are measured by Windows 10 for integrity verification purposes;
- An analysis of the TPM provisioning process implemented in Windows 10: We identify and analyze relevant activities that are part of the TPM provisioning process. This is a process for initializing the TPM and making it ready for use. We consider both scenarios when the TPM provisioning process is triggered automatically by Windows 10 or manually by users. We provide a technical overview of the workflow of

the TPM provisioning process and we identify the system components and resources taking part in it, such as registry values and library DLL files;

- All analysis steps have been performed with particular regard to potential attack vectors on the overall TPM and integrity verification/measurement mechanisms.

This work is structured as follows:

- Section 1.3 introduces concepts and terms relevant for better understanding the contents we present;
- Section 2 provides technical information on functionalities whose analysis is in the scope of this work package:
  - Section 2.1 provides an overview of the interfaces between Windows 10 and the TPM: Section 2.1.1 and Section 2.1.2 discuss the communication interfaces between the TPM and parts of the Windows 10 system deployed in user- and kernel-land; Section 2.1.3 introduces code useful for constructing TPM usage profiles. Under TPM usage profiles, we understand information on system entities communicating with the TPM, such as specific user processes or system services. TPM usage profiles also include relevant related information, such as communication patterns and frequencies;
  - Section 2.2 provides an analysis of the booting process of Windows 10: Section 2.2.1, Section 2.2.2, and Section 2.2.3, discuss the activities performed as part of this process in the context of the three entities booting the Windows system to its full extent: the boot manager, the Windows loader, and the Windows kernel;
  - Section 2.3 focuses on the ELAM technology implemented in Windows 10: the ELAM driver, its decision-making process, and its malware database;
  - Section 2.4 provides an overview of the integrity measurement mechanism of Windows 10; that is, it provides information on the measurement data stored in the TPM, the implementation of integrity measurement in Windows 10, and the use of measurement data for platform integrity verification;
  - Section 2.5 discusses relevant activities that are part of the TPM provisioning process implemented in Windows 10: Section 2.5.1 and Section 2.5.2 discuss these activities, conducted when the TPM provisioning process is triggered automatically by Windows 10 and manually by users, respectively;
  - Section 2.6 provides an overview of relevant security aspects; this includes an overview of attack scenarios considered during analysis as well as implemented mitigations.

  Throughout the technical discussions, we depict function call stacks and pseudo-code. We emphasize that these call stacks, for the sake of brevity, present only functions that we consider relevant to the discussions. We do not necessarily depict all functions that are part of the call stacks as implemented in Windows 10. In addition, the depicted pseudo-code is a high abstraction of real code and does not consistently follow the syntax of a particular programming language. We depict pseudo-code in a form considered optimal for better understanding the discussed matter. The pseudo-code depicted in this work loosely follows a C and C++-like programming language syntax.
- Section 3 provides an overview of the configuration and logging capabilities of Windows 10 for managing the TPM and logging TPM events: Section 3.1 discusses the interfaces that Windows 10 provides for users to configure the TPM; Section 3.2 discusses the use of the core logging facilities of Windows 10 for logging TPM events;
- In the Appendix: the 'Tools' section lists used tools and where they can be found; the 'TPM Usage Profiler' section provides a script for constructing TPM usage profiles, relevant to the discussions in Section 2.1.3; the 'TPM Usage' section provides a table listing executables that communicate with the TPM, relevant to the discussions in Section 2.1.3; the 'ELAM Database Parser' section provides a script for parsing the malware database that the ELAM driver uses for anti-malware verification, relevant to the discussions in 2.3; the 'Measured Executables' section provides a list of names of executables whose integrity is measured by Windows 10, relevant to the discussions in Section 2.4.

1.3 General Concepts and Terminology

The TPM is a standard for a secure cryptoprocessor developed by the Trusted Computing Group (TCG). The TPM implements in hardware three roots of trust ([TCGLP1 2016], Section 9.4): root of trust for measurement (RTM), root of trust for storage (RTS), and root of trust for reporting (RTR).

The RTM is typically the CPU executing an implicitly trusted, immutable code that initiates the integrity measurement process in firmware context. Integrity measurement typically consists of calculating hash values of relevant data. These values are stored for later comparison with previously measured hash values of the same data, such that a mismatch indicates data corruption.

Due to its immutability, the trusted code executed by the CPU is also referred to as the static root of trust for measurement (SRTM), or the core root of trust for measurement (CRTM). The SRTM is a set of instructions measuring itself and other firmware content, and storing these measurements in the TPM ([Butterworth 2013], Section 1). Although the SRTM itself can be stored in the TPM, it is typically stored in the platform's Boot Block. The Boot Block is part of the platform's firmware ([TCGF 2016], Section 2.3.3.1).

The RTS is the TPM's memory, which is shielded from external access. The TPM has volatile and non-volatile memory, structured into registers ([ERNW WP2], Section3.5). Examples are the platform configuration registers (PCRs), which are used for storing integrity measurement data (see Section 2.4). A typical TPM has 24 PCRs, such that each PCR is uniquely identified by an integer number with a value between 0 and 23, known as the PCR's index.

The RTS provides secure storage of data and protection of objects stored outside the TPM, where the root of this protection is the RTS ([TCGLP1 2016], Section 23). Example objects are keys and arbitrary files. The protection of external objects is structured as a hierarchy of protected objects. The root of this hierarchy is a TPM key named the storage root key (SRK). TPM keys are encryption keys stored in a format understandable by the TPM. The SRK is created by the TPM, stored in the TPM's non-volatile memory, and its private part never leaves the TPM.

The SRK protects a given object by encrypting the object's sensitive area (e.g., a key's private part) with a symmetric encryption key derived from the private part of the SRK ([TCGLP1 2016], Section 22.3). This is known as wrapping. Among other objects, the SRK may wrap TPM keys of any type, for example, signing keys (keys used for digitally signing data), or storage keys. Storage keys are TPM keys that themselves can wrap other keys or any object, thus constructing a hierarchy of protected objects with multiple parent-child relationships ([TCGLP1 2016], Section 23.1).

The RTR is the TPM and is implemented as the TPM's functionality of reporting contents stored in the RTS (e.g., values of PCRs or the TPM's audit logs, [TCGLP1 2016], Section 9.4.3) to external entities, such as remote attestation servers (see Section 2.4). The identity of the RTR is determined by the TPM's EK. The EK is a TPM key generated in TPM context such that its private part never leaves the TPM. The EK is typically, however, not necessarily, installed on the TPM at platform manufacturing time. The EK is stored in the TPM's non-volatile memory. The EK is unique for each TPM. This makes the EK a key uniquely identifying the TPM it is stored on, and therefore, the platform the TPM is installed on.

Due to privacy concerns, EKs are not directly used for platform identification. TPM keys known as attestation identity keys (AIKs) are used for this purpose. AIKs are generated by the TPM and bound to EKs during a certification process; that is, they act as aliases of EKs. There can be multiple AIKs bound to a single EK. We refer to the AIK Certificate Enrollment Specification[TCGAIK 2011] for more information on the certification process for associating AIKs with EKs.

AIKs are TPM signing keys, exclusively used for signing data originating from the TPM (e.g., values of PCRs). For example, a certificate authority (CA) can verify that a key originates from the TPM and then certify it, only after it has verified the signature of the signing AIK. This AIK itself has to be certified. This is known as key attestation.[1]

1 https://msdn.microsoft.com/en-us/library/dn410314.aspx [Retrieved: 22/9/2017]

Figure 1 depicts a sample TPM key hierarchy including the previously mentioned EK, SRK, and AIKs. In Figure 1, *StorK* is a TPM storage key, *data* is arbitrary data, and *external storage* is any storage medium that is not the TPM's memory, for example, the hard disk of a given platform.

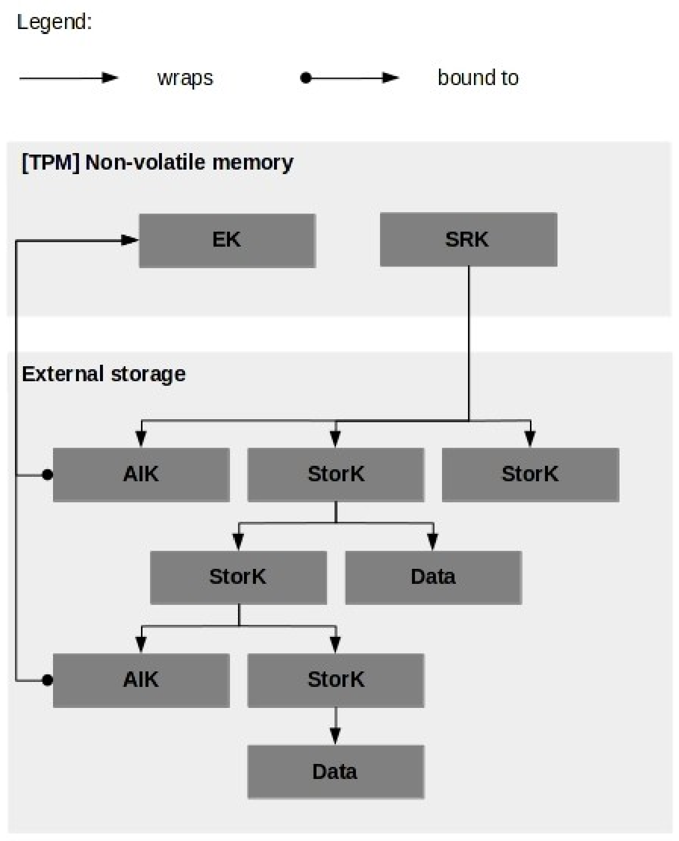


Figure 1: A TPM key hierarchy

For each EK, an endorsement and a platform certificate is created. The endorsement certificate contains the public part of the EK and is used for attesting that the platform has a unique TPM installed on it. The platform certificate contains a reference to the EK and provides a proof of the binding between the EK and the platform where the TPM is installed on, acknowledged by a trusted CA. The platform and endorsement certificates are typically, however, not necessarily, created at manufacture time by the platform's manufacturer. For more information on the creation of the endorsement and platform certificates, we refer to the Certificate Management Messages Over CMS (CMC) Profile for EK/Platform Certificate Enrollment for TPMv1.2 Specification, version 1.0, revision 5. This is the latest such specification at the time of writing [TCGCMC 2013].[2]

The TPM is a passive device executing commands submitted to it, and returning relevant data, such as status codes. We refer to these commands as TPM commands. In their raw form, a TPM command is a sequence of bytes stored in a TPM command buffer. This buffer has a command-specific layout defined in the Trusted Platform Module Library, Part 3: Commands [TCGLP3 2016].

In the context of the TPM, each TPM command is uniquely identified by an integer value, known as the TPM command code (see [TCGLP2 2016], Section 6.5.2). In the context of the Windows operating system, each TPM command is uniquely identified by a command ID. Based on these IDs, Windows implements access control over TPM commands to restrict the execution of the commands to users (see Section 3.1.2).[3]

2 At the time of writing, there is no EK/platform certificate enrollment specification for TPM 2.0.
3 https://technet.microsoft.com/en-us/library/dn466537(v=ws.11).aspx [Retrieved: 22/9/2017]

Some TPM commands and functionalities are protected such that they can be executed only if a proof of ownership of the TPM is provided. This proof is in the form of an authorization value. There are three types of TPM authorization values: platform authorization, endorsement authorization, and owner authorization value. Among other things, the first value is used for operations performed by the platform's firmware (e.g., allocation of non-volatile memory), the second for performing EK-related operations (e.g., creation of an EK), and the third for managing the TPM and its storage hierarchy ([TCGLP1 2016], Section 13.3).

The TPM allows for a holder of an authorization value to delegate its privileges to others. The TPM of version 1.2 stores delegation information in a data structure known as a delegation binary large object (blob) ([TCGPro 2014], Section 1.3.4). The TPM of version 2.0 may also implement delegation through policies.[4]

In order to protect itself from dictionary attacks, where an attacker tries different authorization values until one succeeds, the TPM implements a lockout mechanism. The TPM counts the number of TPM authorization failures over a time period, and when a given threshold is reached, it locks. TPM lockout can be reset by providing an authorization value, known as lockout authorization ([TCGLP1 2016], Section 13.7).

The owner, endorsement, and lockout authorization values are set during a process of taking the ownership of the TPM ([TCGLP1 2016], Section 13.8.1). This process is the core activity of the TPM provisioning process, which initializes and prepares the TPM for use. TPM provisioning may be triggered manually by a user, or automatically by the operating system installed on the platform where the TPM being provisioned is deployed. In Section 2.5, we discuss the scope and implementation of the TPM provisioning process in Windows 10 in particular.

For a detailed information on the terms and concepts mentioned in this section, we refer to the TPM 2.0 Library Specification. This specification is available-online at: https://trustedcomputinggroup.org/tpm-library-specification/ [Retrieved: 22/9/2017]. The documents that are part of this specification are the latest official source of information on the TPM 2.0 standard, which is in the focus of this work (see Section 'Executive Summary').

4 For an overview of the difference between the implementation of the delegation mechanism in the TPM 1.2 and 2.0 standards, we refer to ([Proudler 2014], Table 6.21).

2 Technical Analysis of Functionalities

2.1 TPM Communication Interfaces

In this section, we discuss how the different parts of the Windows 10 operating system (see [ERNW WP2]) deployed in user-land (Section 2.1.1) and in kernel-land (Section 2.1.2), use the TPM. We focus on the communication interfaces between Windows 10 and the TPM, which we depict in Figure 2. 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 2.1.3).

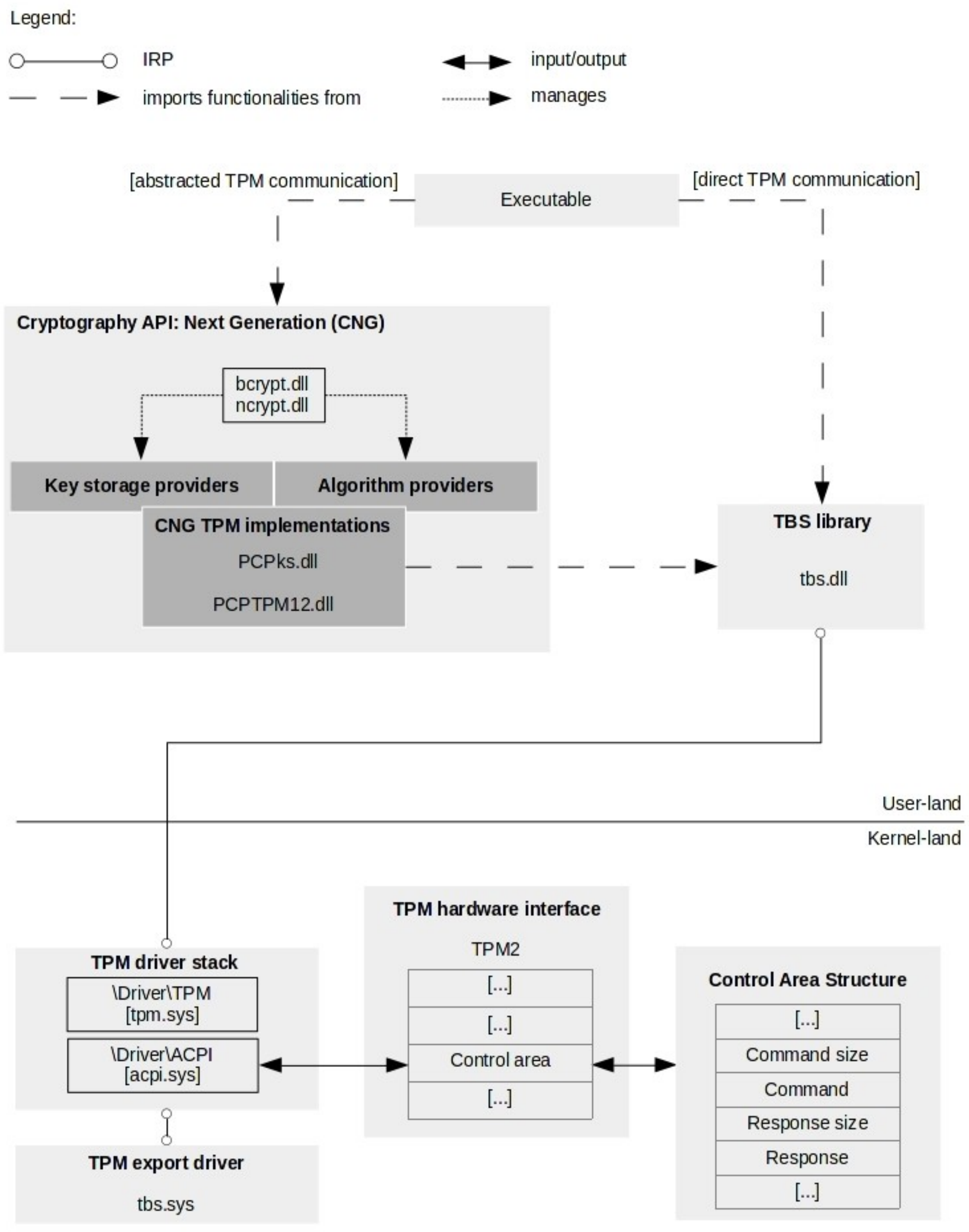


Figure 2: Interfaces for communicating with the TPM

2.1.1 TPM Communication: User-land

The parts of the Windows 10 system deployed in user-land (referred to as *Executable* in Figure 2) can communicate with the TPM in two ways: direct (*direct TPM communication* in Figure 2) or abstracted (*abstracted TPM communication* in Figure 2).

Direct TPM communication: The direct TPM communication involves executing functions declared as part of the TBS library file named `tbs.dll` (*TBS library* in Figure 2). This library file implements a number of functions, structures, and data types for communicating with the TPM.[5] 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.[6] 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, passing the TPM command byte sequence in the form of IRPs.[7] The system call may be issued, for example, using the `NtDeviceIoControlFile` Windows API function ([ERNW WP2], Section 2.1).[8]

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.[9] 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, a driver object and a device object of the physical device is associated.[10] 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.[11]

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`).

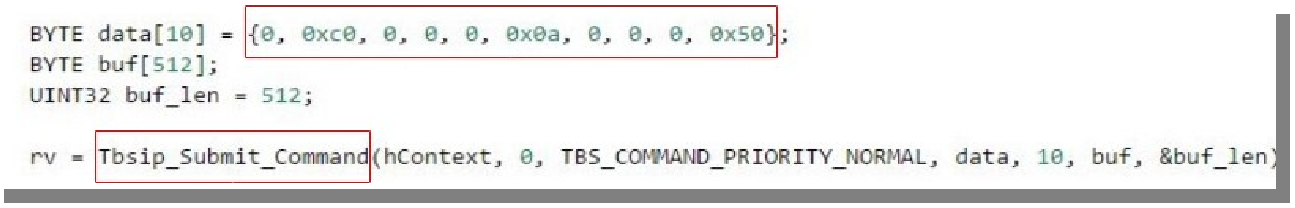
Following the hierarchy of the TPM driver stack, when a command in the form of an IRP is submitted to the TPM driver `tpm.sys`, it passes the procession of the IRP to the ACPI driver `acpi.sys`. According to the TCG ACPI Specification, version 1.2, revision 8 (this is the latest TCG ACPI specification at the time of writing, [TCGACPI 2017]), `acpi.sys` submits relevant command information to the TPM device by writing

5 https://msdn.microsoft.com/de-de/library/windows/desktop/aa446794(v=vs.85).aspx [Retrieved: 22/9/2017]
6 https://msdn.microsoft.com/de-de/library/windows/desktop/aa446799(v=vs.85).aspx [Retrieved: 22/9/2017]
7 https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/i-o-request-packets [Retrieved: 22/9/2017]
8 https://msdn.microsoft.com/en-us/library/ms648411(v=vs.85).aspx [Retrieved: 22/9/2017]
9 https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/driver-stacks [Retrieved: 22/9/2017]
10 https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/introduction-to-device-objects [Retrieved: 22/9/2017]
11 https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/object-names [Retrieved: 22/9/2017]

this information at a location within a memory region starting at a specific address. This address is stored in the field `Control area` ([TCGACPI 2017], Table 7) of the ACPI hardware interface description table of the TPM device, named `TPM2` (*TPM hardware interface* and *TPM2* in Figure 2).

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`.[12] 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 3 depicts a snippet of the application's program code for submitting the TPM command.



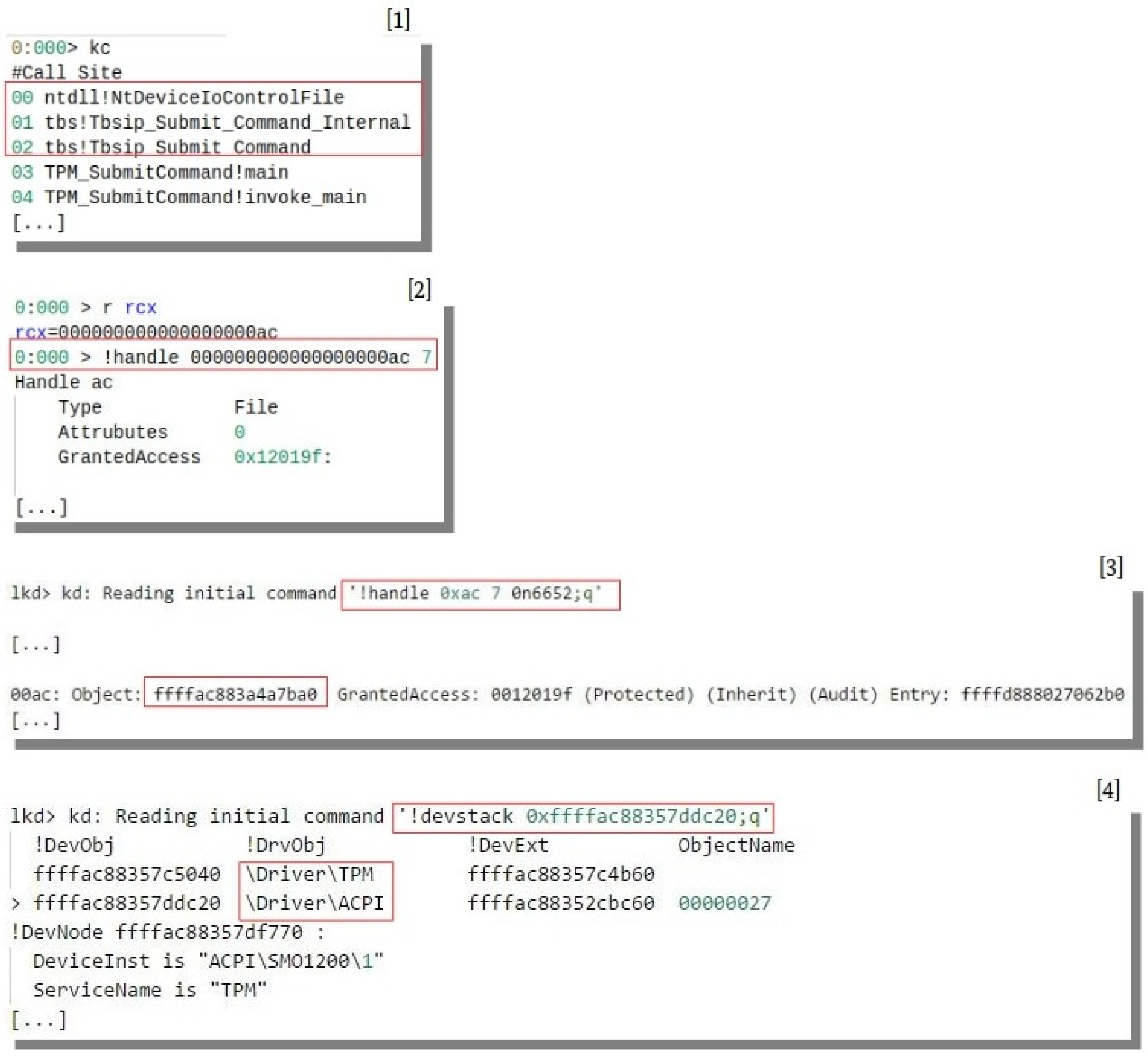
```
BYTE data[10] = {0, 0xc0, 0, 0, 0, 0x0a, 0, 0, 0, 0x50};
BYTE buf[512];
UINT32 buf_len = 512;

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

Figure 3: Submitting a TPM command using Tbsip_Submit_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 4 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 4). `NtDeviceIoControlFile` submits IRPs to a device driver such that its first parameter is a handle of the device object associated with the driver.[13] 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.[14] By printing out the contents of this register, we obtained the handle value `0xac` (*[2]* in Figure 4). We then obtained the address at which information about the object associated with this handle is stored. This address is `0xffffac883a4a7ba0` (*[3]* in Figure 4). 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 4).

The TPM driver manages the scheduling of TPM resources and submits commands to the TPM device in a procedural manner. Figure 5 depicts some of the functions implemented in `tpm.sys`, which are involved in command processing. Figure 5 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 (`TpmTransportMemBase::SubmitCommand` in Figure 5).[15,16]

12 https://msdn.microsoft.com/de-de/library/windows/hardware/dn974551(v=vs.85).aspx [Retrieved: 22/9/2017]
13 https://msdn.microsoft.com/de-de/library/windows/desktop/ms724457(v=vs.85).aspx [Retrieved: 22/9/2017]
14 https://technet.microsoft.com/en-us/library/security/zthk2dkh(v=vs.90).aspx [Retrieved: 22/9/2017]
15 In this work, we use the scope operator :: when referring to functions declared as part of data structures.
16 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.


[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
[...]
```

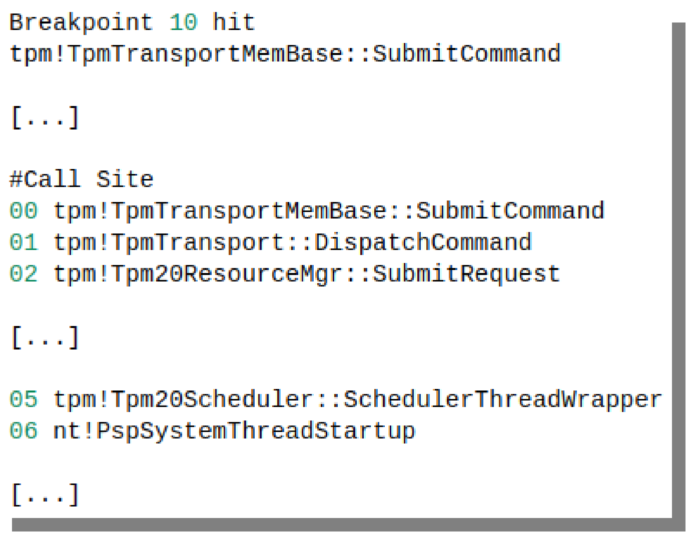
[4]

```
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 4: Execution of Tbsip_Submit_Command

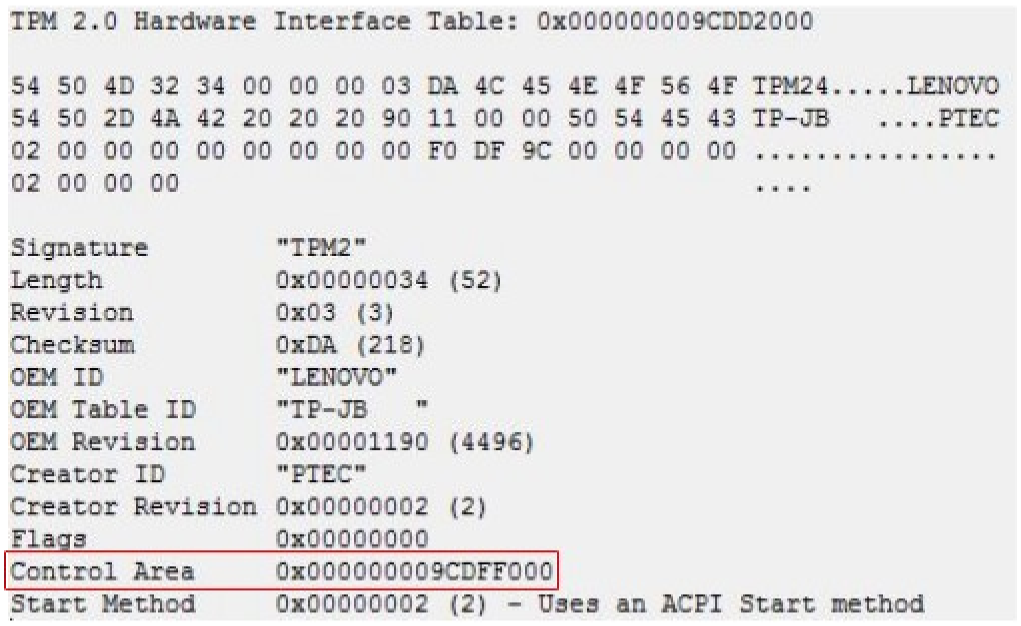
We observed that `TpmTransportMemBase::SubmitCommand` submits commands to the TPM and it is invoked as follows. When an IRP containing a TPM command is received by the TPM driver, it schedules the creation of a thread handling the IRP in the `Tpm20Scheduler::SchedulerThreadWrapper` function. When the 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 6 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 ([TCGACPI 2017], Table 7), this indicates that the memory region starting at the address stored in this field is used as previously described.



```
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 5: Submission of TPM commands



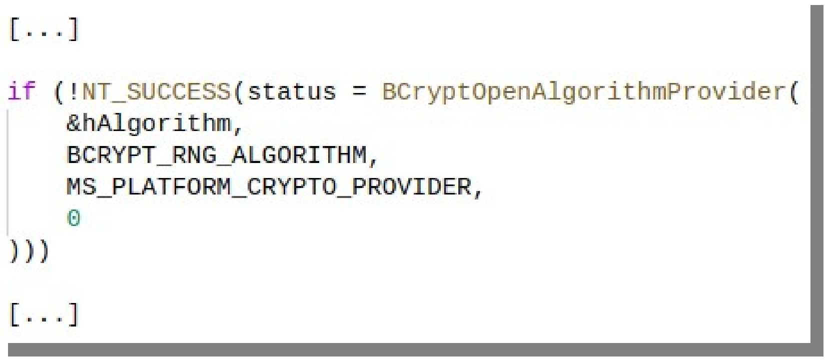
```
TPM 2.0 Hardware Interface Table: 0x000000009CDD2000

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 6: 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 ([TCGLP3 2016], Section 5). As part of this procedure, the processed command is authorized by evaluating the provided authorization value (see [TCGLP1 2016]; Section 19 on authorization of TPM commands). 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.

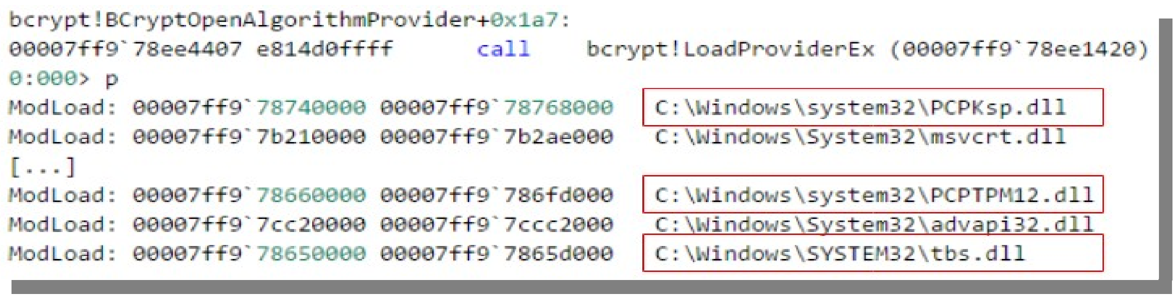
Abstracted TPM communication: Windows 10 provides the CNG library, first introduced in Windows Vista, for abstracting the functionalities of the TBS library. The functions implemented as part of the CNG library act as wrappers of functions of the TBS library, adding functionalities and making their use easier.[17]

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



```
[...]

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

[...]
```

Figure 7: 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 8: Dynamic loading of TPM-related library files

CNG uses the concept of cryptographic providers, where providers are entities performing cryptographic operations (e.g., hashing, digital signature verification).[18] 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,[19] whereas the latter are primarily used for performing key operations, such as creating and storing keys.[20]

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`.[21] Microsoft's basic software-implemented cryptographic provider is referred to as the `Microsoft Primitive Provider`.[22]

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.[23] The library files `%SystemRoot\System32\PCPks.dll` and `%SystemRoot\System32\PCPTPM12.dll` implement CNG functionalities related to the TPM (*CNG TPM* Implementations in Figure 2). 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`.[24]

18 https://msdn.microsoft.com/en-us/library/windows/desktop/bb931380(v=vs.85).aspx [Retrieved: 22/9/2017]
19 https://msdn.microsoft.com/en-us/library/windows/desktop/bb931354(v=vs.85).aspx [Retrieved: 22/9/2017]
20 https://msdn.microsoft.com/en-us/library/windows/desktop/bb931355(v=vs.85).aspx [Retrieved: 22/9/2017]
21 https://msdn.microsoft.com/en-us/library/windows/hardware/hh998513(v=vs.85).aspx [Retrieved: 22/9/2017]
22 https://msdn.microsoft.com/en-us/library/windows/desktop/aa375479(v=vs.85).aspx [Retrieved: 22/9/2017]
23 https://msdn.microsoft.com/de-de/library/windows/desktop/aa376214(v=vs.85).aspx [Retrieved: 22/9/2017]
24 https://msdn.microsoft.com/en-us/library/windows/desktop/bb204778(v=vs.85).aspx [Retrieved: 22/9/2017]

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 7 depicts a snippet of the application's program code, where the `BcryptOpenAlgorithmProvider` function is used for loading and initializing this provider.[25]

We set a breakpoint at `BcryptOpenAlgorithmProvider`. We observed that it dynamically loads the CNG TPM implementations (i.e., the library files `PCPks.dll` and `PCPTPM12.dll`) and the TBS library (i.e., the library file `tbs.dll`); see Figure 8. 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').

2.1.2 TPM Communication: Kernel-land

The parts 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 2). Basically, export drivers are kernel-mode library files exporting routines to the kernel or other drivers.[26]

The TPM export driver is implemented in `%SystemRoot\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 4), 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`.[27]

2.1.3 TPM Usage Profiles

In Section 2.1.1 and Section 2.1.2, 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. Taking into account the scope of this work package, 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.

25 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`.
26 https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/creating-export-drivers [Retrieved: 22/9/2017]
27 https://msdn.microsoft.com/en-us/library/windows/hardware/ff566441(v=vs.85).aspx [Retrieved: 22/9/2017]

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 profiles. 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 or disabling BitLocker or Microsoft Account ([ERNW WP2], Table 2).[28]

For observing the usage of the TPM by processes that use the CNG TPM interface (see Section 2.1.1, paragraph 'Abstracted TPM communication'), the output of the script can be analyzed in a combination with detailed log data produced by Windows. Windows has the capability to log detailed information on issued TPM commands, such as command codes and byte sequences. Such an analysis provides a comprehensive, in-depth insight into the activities of the `Platform Cryptographic Provider`.

2.2 Windows Boot

This section describes the booting process of Windows 10. We focus on activities performed as part of this process by Windows itself, which we refer to as the Windows boot process. Activities performed by the UEFI firmware, an environment predecessing the operating system in the booting process are out of scope.

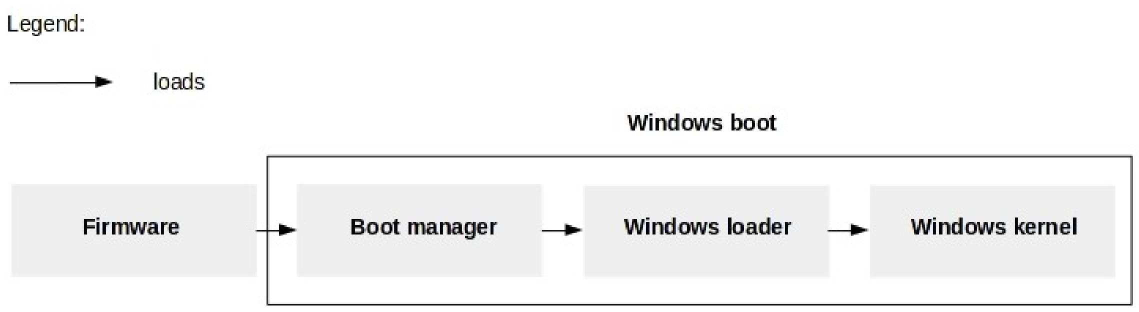


Figure 9: The booting process of a Windows-based platform

Figure 9 depicts the booting process of a Windows-based platform, where each entity taking part in it loads the next entity in the booting chain ([Russinovich 2012], Chapter 13).[29] The core activities of the Windows boot process are performed by three entities booting the Windows system to its full extent, making it ready for use: the boot manager, the Windows loader, and the Windows kernel (see Figure 9). In summary:

- the boot manager reads the Boot Configuration Database (BCD, a database of configuration parameters for starting Windows), potentially presents a boot menu, and loads the Windows loader;
- the Windows loader loads boot drivers and the Windows kernel; boot drivers are drivers that are required for starting the Windows kernel and the Windows operating system in general, such as file-system drivers;[30]

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

29 Although [Russinovich 2012] provides technical facts about versions of Windows older than Windows 10, the ones we reference apply to Windows 10 too. We verified and observed this over the course of this work.

- the Windows kernel loads system drivers, initializes the Windows subsystem, and runs the session manager `smss.exe` ([ERNW WP2], Section 2.1); system drivers are drivers required for regular system operation.

A structured and more detailed overview of the booting process of a Windows-based platform is presented in ([Russinovich 2012], Table 13-1).

In Section 2.2.1, Section 2.2.2, and Section 2.2.3, we discuss the boot manager, the Windows loader, and the Windows kernel, respectively. Given that the TPM is in the focus of this work package, a component representing a hardware-implemented root of trust for sensitive operations, we focus our discussions in these sections on:

- the loading of executables (also referred to as images) and the verification of their integrity: we identify the root of trust for integrity verification in the context of the boot manager, the Windows loader, and the Windows kernel (paragraph 'Image loading and integrity verification' in Section 2.2.1, Section 2.2.2, and Section 2.2.3);
- the verification of the integrity of the boot manager, the Windows loader, and the Windows kernel: we identify the root of trust for integrity verification of the core participants in the Windows boot process (paragraph 'Root of trust' in Section 2.2.1, Section 2.2.2, and Section 2.2.3);
- the use of the TPM in the context of the boot manager, the Windows loader, and the Windows kernel (paragraph 'TPM usage' in Section 2.2.1, Section 2.2.2, and Section 2.2.3).

In Table 1 below, we summarize information on the roots of trust for integrity verification (column 'Root of trust') and the use of the TPM (column 'TPM usage') in the context of the boot manager, the Windows loader, and the kernel. We discuss the contents of Table 1 in more detail in the following sections.

	Root of trust	**TPM usage**
Boot manager	UEFI	• integrity measurement • BitLocker operations
Windows loader	Boot manager	• integrity measurement • BitLocker operations
Windows kernel	Windows loader	

Table 1: Roots of trust and TPM usage

2.2.1 Boot Manager

The boot manager is implemented in the `bootmgr.exe` executable. This executable is stored in a hidden partition of the system disk, known as the boot partition. On UEFI-based platforms, boot is the first partition, followed by other partitions, such as the Windows and the Recovery partitions.[31] The boot manager executable is compressed with the Microsoft-proprietary Xpress Huffmann compression algorithm [MicXCA 2017].

We mounted the boot partition using the `diskpart` utility (executable: `diskpart.exe`) and decompressed the boot manager executable using the `bmzip` tool. The observations presented in this section are based on a static analysis of the decompressed executable of the boot manager, primarily using

30 https://docs.microsoft.com/en-us/windows-hardware/drivers/install/installing-a-boot-start-driver [Retrieved: 22/9/2017]

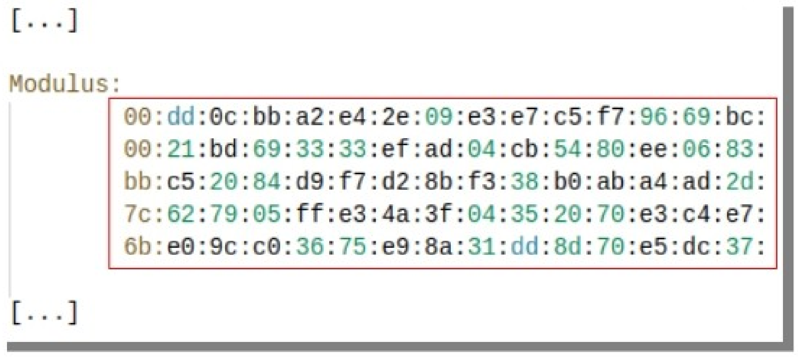
31 For detailed information on the partition layout on UEFI-based platforms, we refer to https://docs.microsoft.com/en-us/windows-hardware/manufacture/desktop/configure-uefigpt-based-hard-drive-partitions [Retrieved: 22/9/2017]

the `IDA` disassembler. In addition, we performed dynamic analysis using the `windbg` debugger by establishing a serial debugging session.

Root of trust The boot manager is digitally signed from Microsoft using the Authenticode digital signing technology.[32] We present more details on Authenticode when discussing integrity verification in the context of the Windows kernel in Section 2.2.3. The boot manager is loaded by the platform's firmware. Therefore, if Secure Boot is enabled, the UEFI firmware is responsible for verifying the integrity of the boot manager by verifying its signature as specified in the UEFI Specification, version 2.7, errata A ([UEFIFE], Section 31.2.4), the latest such specification at the time of writing. The analysis of the UEFI firmware (and thus the analysis of the image verification by UEFI) is out of scope of this work package. However, the Windows Secure Boot mechanism depends on the initial proper image verification by the UEFI. The rest of the image verification chain is described in the following paragraphs.

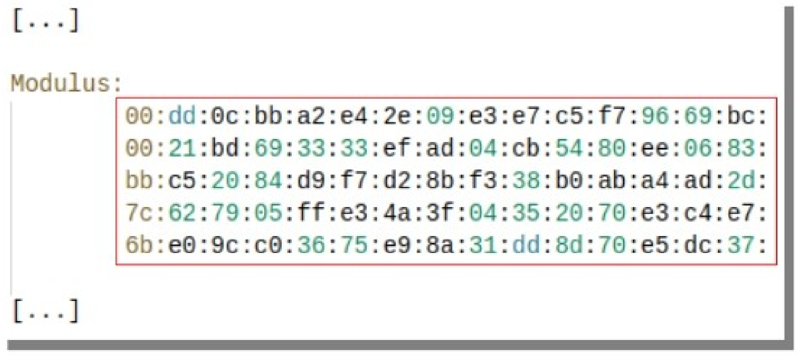
In general, signature verification is a process consisting of: (i) calculation of a hash of the entity whose integrity is to be verified; (ii) decryption of an encrypted digest of a pre-calculated hash of the entity (i.e., the signature) using the public key of the signer; and (iii) comparing the hash values. A mismatch indicates corruption of the executable. In the context of Secure Boot, the UEFI secure database stores certificates issued by signers of system loading components, such as the boot manager. Signers are typically operating system vendors, for example, Microsoft. These certificates encapsulate public keys needed for signature verification.

[1]



```
[...]

Modulus:
        00:dd:0c:bb:a2:e4:2e:09:e3:e7:c5:f7:96:69:bc:
        00:21:bd:69:33:33:ef:ad:04:cb:54:80:ee:06:83:
        bb:c5:20:84:d9:f7:d2:8b:f3:38:b0:ab:a4:ad:2d:
        7c:62:79:05:ff:e3:4a:3f:04:35:20:70:e3:c4:e7:
        6b:e0:9c:c0:36:75:e9:8a:31:dd:8d:70:e5:dc:37:

[...]
```

[2]



```
[...]

Modulus:
        00:dd:0c:bb:a2:e4:2e:09:e3:e7:c5:f7:96:69:bc:
        00:21:bd:69:33:33:ef:ad:04:cb:54:80:ee:06:83:
        bb:c5:20:84:d9:f7:d2:8b:f3:38:b0:ab:a4:ad:2d:
        7c:62:79:05:ff:e3:4a:3f:04:35:20:70:e3:c4:e7:
        6b:e0:9c:c0:36:75:e9:8a:31:dd:8d:70:e5:dc:37:

[...]
```

Figure 10: Public keys stored in the boot manager and the UEFI firmware

In order to identify the root of trust for verifying the integrity of the boot manager, we first extracted the certificate issued by the signer of the boot manager from the boot manager executable. We did this by right clicking on the executable and clicking on `Properties` → `Digital Signatures` → `Details` → `View Certificate` → `Details` → `Copy to File...`[33] We then printed out the contents of the certificate using the `openssl` utility. We executed `openssl x509 -inform DER -text -in certificate.cer`,

32 https://docs.microsoft.com/en-us/windows-hardware/drivers/install/authenticode [Retrieved: 22/9/2017]

where `certificate.cer` is a certificate file in the X.509 Distinguished Encoding Rules (DER) format.[34] Figure 10 depicts a portion of the public key (i.e., the modulus) stored as part of the extracted certificate (*[1]* in Figure 10). The issuer of this certificate is 'Microsoft Windows Production PCA 2011'.

We continued by extracting the certificates stored in the UEFI secure database using the `chipsec` toolset. We then printed out the contents of these certificates using the `openssl` utility. Figure 10 depicts the modulus of the public key stored as part of a certificate extracted from the allowed UEFI database (*[2]* in Figure 10), with an issuer of 'Microsoft Windows Production PCA 2011'. This modulus is same as the modulus stored as part of the certificate extracted from the boot manager executable. Taking into account ([UEFIFE], Section 31.2.4), this shows that the certificate stored in the UEFI secure database is used for verifying the integrity of the boot manager executable; that is, the root of trust for verifying the integrity of the boot manager is the UEFI firmware (if present).

Image loading and integrity verification Figure 11 is a graphical representation of the stack of functions involved in the image loading and integrity verification performed by the boot manager. The latter are marked in grey in Figure 11. The `BmMain` function is the entry point of the boot manager executable. In addition to required images (e.g., library DLL files), we observed that the boot manager loads only two executables: `bootmgr.exe.mui` (a language resource file for the boot manager)[35] and `winload.exe` (the Windows loader). The executable `bootmgr.exe.mui` is loaded by executing the function stack `BlInitializeLibrary` -> `BlpResourceInitialize` -> `ImgpLoadPEImage`, and `winload.exe` by executing the function stack `BmpLaunchBootentry` -> `BmpTransferExecution` -> `BlImgLoadBootApplication` -> `ImgpLoadPEImage`. `ImgpLoadPEImage` performs image loading and integrity verification.

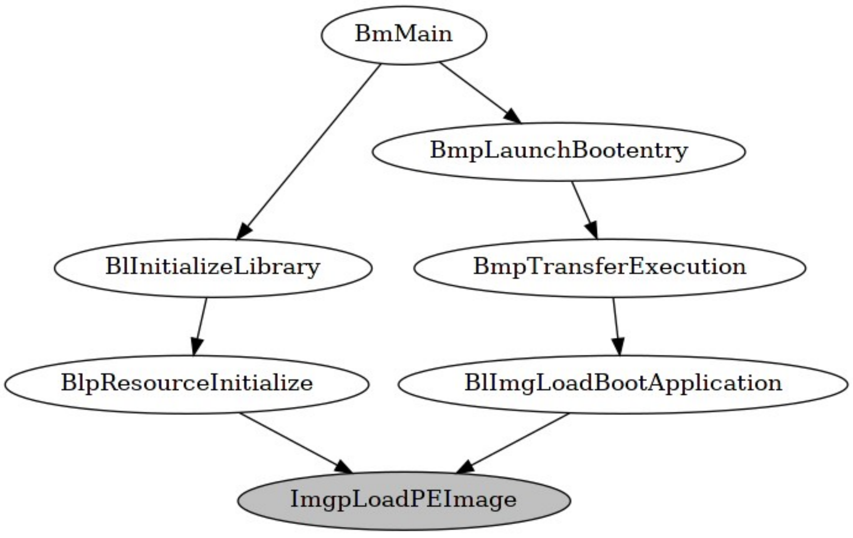


Figure 11: Function stack: Image loading and integrity verification by the boot manager

The image integrity verification implemented in `ImgpLoadPEImage` is conceptually identical to the one implemented in the Windows kernel. We describe the image integrity verification process in detail in Section 2.2.3, where we discuss the implementation of this process in the Windows kernel.

33 The certificate issued by the signer of the boot manager is embedded in `bootmgr.exe` and therefore can be extracted from it [Mic 2008]. We emphasize that this is not always the case. We discuss more on this topic in Section 2.2.3.

34 https://technet.microsoft.com/en-us/library/cc770735(v=ws.11).aspx [Retrieved: 22/9/2017]

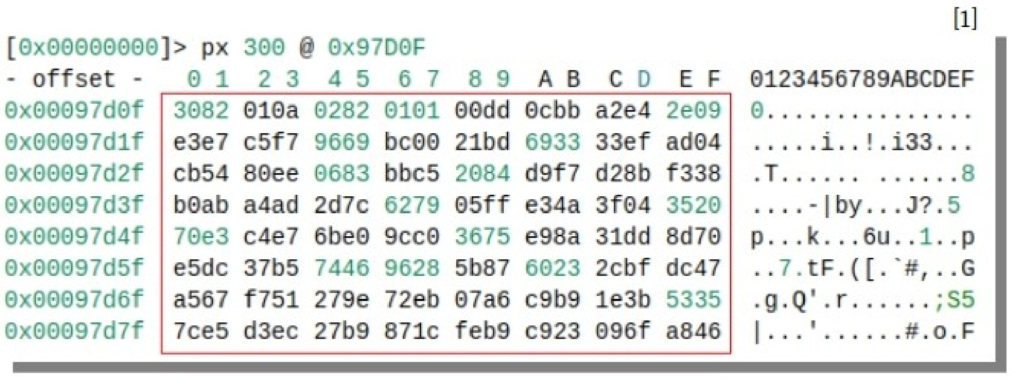
35 https://msdn.microsoft.com/de-de/library/windows/desktop/dd319073(v=vs.85).aspx [Retrieved: 22/9/2017]

When analyzing the operation of `ImgpLoadPEImage`, we observed that certificates of image signers are verified using a public key stored as part of a root certificate issued by Microsoft. This certificate is hardcoded in the boot manager executable, in a data structure informally referred to as the `root table`. Figure 12 depicts the public key stored in the `root table` of the boot manager (*[1]* in Figure 12). We extracted this public key using the `radare2` framework. The public key is encoded in Abstract Syntax Notation One (ASN.1) format. Figure 12 depicts the same public key, however, stored in a variable used for verification in `ImgpLoadPEImage` (*[2]* in Figure 12). The fact that hardcoded contents of the `root table` structure are used for verification of signers' certificates shows that the root of trust for verifying the integrity of images loaded by the boot manager is the boot manager itself.

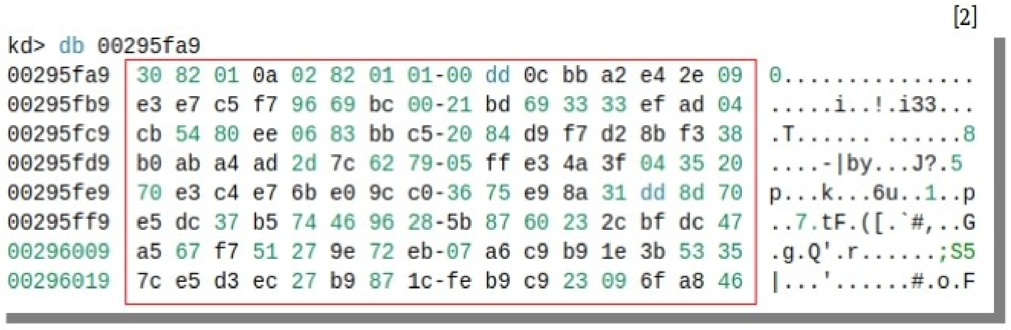
When the boot manager is finished with image loading, it transfers execution control to the Windows loader. To this end, it executes the `Archx86TransferTo64BitApplicationAsm` function. Depending on the architecture of the platform (e.g., 32- or 64-bit), an alternative function may be executed, for example, `Archx86TransferTo32BitApplicationAsm`. This function passes boot parameters to the Windows loader, primarily in the form of byte sequences.

TPM usage In the context of the boot manager, the TPM is used for integrity measurement and BitLocker operations (e.g., reading keys stored in the TPM). We observed this by setting breakpoints at the functions that are part of the API implemented in the boot manager, which we refer to as the boot manager's TpmApi. This API consists of functions performing TPM-related operations, with names beginning with `TpmApi`. These functions construct TPM command buffers and communicate with the TPM.

We present more details on the use of the TPM for integrity measurement in Section 2.4, paragraph 'Implementation of integrity measurement'.


[1]

```
[0x00000000]> px 300 @ 0x97D0F
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x00097d0f  3082 010a 0282 0101 00dd 0cbb a2e4 2e09  0...............
0x00097d1f  e3e7 c5f7 9669 bc00 21bd 6933 33ef ad04  .....i..!.i33...
0x00097d2f  cb54 80ee 0683 bbc5 2084 d9f7 d28b f338  .T...... ......8
0x00097d3f  b0ab a4ad 2d7c 6279 05ff e34a 3f04 3520  ....-|by...J?.5
0x00097d4f  70e3 c4e7 6be0 9cc0 3675 e98a 31dd 8d70  p...k...6u..1..p
0x00097d5f  e5dc 37b5 7446 9628 5b87 6023 2cbf dc47  ..7.tF.([.`#,..G
0x00097d6f  a567 f751 279e 72eb 07a6 c9b9 1e3b 5335  .g.Q'.r......;S5
0x00097d7f  7ce5 d3ec 27b9 871c feb9 c923 096f a846  |...'......#.o.F
```


[2]

```
kd> db 00295fa9
00295fa9  30 82 01 0a 02 82 01 01-00 dd 0c bb a2 e4 2e 09  0...............
00295fb9  e3 e7 c5 f7 96 69 bc 00-21 bd 69 33 33 ef ad 04  .....i..!.i33...
00295fc9  cb 54 80 ee 06 83 bb c5-20 84 d9 f7 d2 8b f3 38  .T...... ......8
00295fd9  b0 ab a4 ad 2d 7c 62 79-05 ff e3 4a 3f 04 35 20  ....-|by...J?.5
00295fe9  70 e3 c4 e7 6b e0 9c c0-36 75 e9 8a 31 dd 8d 70  p...k...6u..1..p
00295ff9  e5 dc 37 b5 74 46 96 28-5b 87 60 23 2c bf dc 47  ..7.tF.([.`#,..G
00296009  a5 67 f7 51 27 9e 72 eb-07 a6 c9 b9 1e 3b 53 35  .g.Q'.r......;S5
00296019  7c e5 d3 ec 27 b9 87 1c-fe b9 c9 23 09 6f a8 46  |...'......#.o.F
```

Figure 12: A public key of a root certificate stored in the boot manager

2.2.2 Windows Loader

The Windows loader is implemented in the `%SystemRoot\System32\winload.exe` executable. The observations presented in this section are based on a static analysis of the implementation of the Windows loader, primarily using the `IDA` disassembler. In addition, we performed dynamic analysis using the `windbg` debugger by establishing a serial debugging session.

Root of trust The Windows loader is loaded by the boot manager ([Russinovich 2012], Table 13-1). In Section 2.2.1, paragraph 'Image loading and integrity verification', we showed that the root of trust for images loaded by the boot manager is the boot manager itself. This includes the executable implementing the Windows loader.

Image loading and integrity verification Figure 13 is a graphical representation of the stack of functions involved in image loading and integrity verification performed by the Windows loader. The latter are marked in grey in Figure 13. The `OslMain` function is the entry point of `winload.exe` once the boot manager has transferred execution control to it (see Section 2.2.1). It invokes `OslpLoadAllModules`, which invokes `OslLoadDrivers` for loading of driver executables, and `OslLoadImage` for loading any other type of image. This includes the executable implementing the Windows kernel, `ntoskrnl.exe`. The Windows loader loads images of required executables (e.g., library DLL files) in the `LoadImports` function.

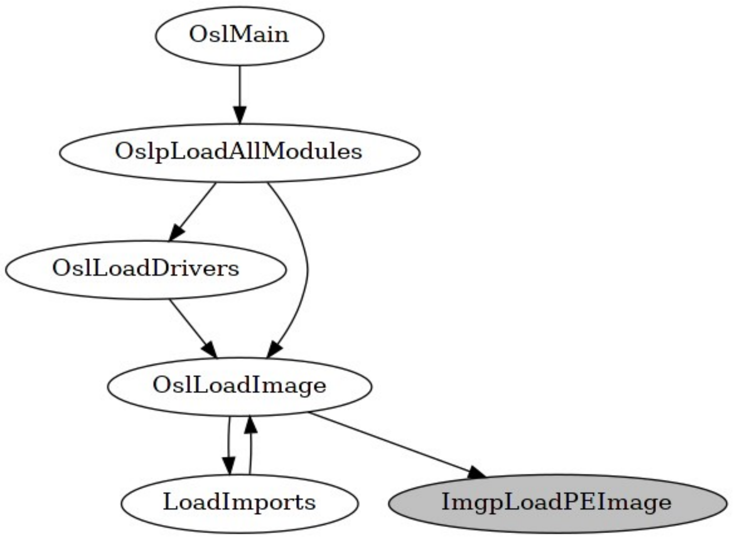


Figure 13: Function stack: Image loading and integrity verification by the Windows loader

All of the functions above ultimately invoke `ImgpLoadPEImage`, which performs image loading and integrity verification. The image integrity verification implemented in `ImgpLoadPEImage` is conceptually identical to the one implemented in the boot manager and the Windows kernel (see Section 2.2.1). We describe the image integrity verification process in detail in Section 2.2.3, where we discuss the implementation of this process in the Windows kernel.

When analyzing the operation of `ImgpLoadPEImage`, we observed that certificates of image signers are verified using a public key stored as part of a root certificate issued by Microsoft. This certificate is hardcoded in `winload.exe`, in a data structure informally referred to as the `root table`. This implementation is identical to that in the boot manager (see Figure 12 and Section 2.2.1). The fact that hardcoded contents of the `root table` structure are used for verification of signers' certificates shows that the root of trust shows that the root of trust for verifying the integrity of images loaded by the Windows loader is the Windows loader itself.

When the Windows loader is finished with image loading, it transfers execution control to the Windows kernel. To this end, it executes the `OslArchTransferToKernel` function. This function passes an instance of the `LOADER_PARAMETER_BLOCK` structure, which contains relevant information, such as system and boot partition paths ([Russinovich 2012], Chapter 13).

TPM usage In the context of the Windows loader, the TPM is used for integrity measurement and BitLocker operations (e.g., reading keys stored in the TPM). We observed this by setting breakpoints at the functions that are part of the TpmApi API implemented in the Windows loader. This API is identical to the TpmApi implemented in the boot manager (see Section 2.2.1).

We present more details on the use of the TPM for integrity measurement in Section 2.4, paragraph 'Implementation of integrity measurement'.

2.2.3 Windows Kernel

The Windows kernel is implemented in the `%SystemRoot\System32\ntoskrnl.exe` executable ([ERNW WP2], Section 2.1). The observations presented in this section are based on a static analysis of the implementation of the Windows kernel, primarily using the `IDA` disassembler. In addition, we performed dynamic analysis using the `windbg` debugger by establishing a network debugging session.

Root of trust The Windows kernel is loaded by the Windows loader ([Russinovich 2012], Table 13-1). In Section 2.2.2, paragraph 'Image loading and integrity verification', we showed that the root of trust for images loaded by the Windows loader is the Windows loader itself. This includes the executable implementing the Windows kernel.

Image loading and integrity verification Figure 14 is a graphical representation of the stack of functions involved in image loading and integrity verification performed by the Windows kernel. The latter are marked in gray in Figure 14. For the sake of brevity, we focus on loading and verification of driver images. The verification of the integrity of other types of images is done same as that of driver images.

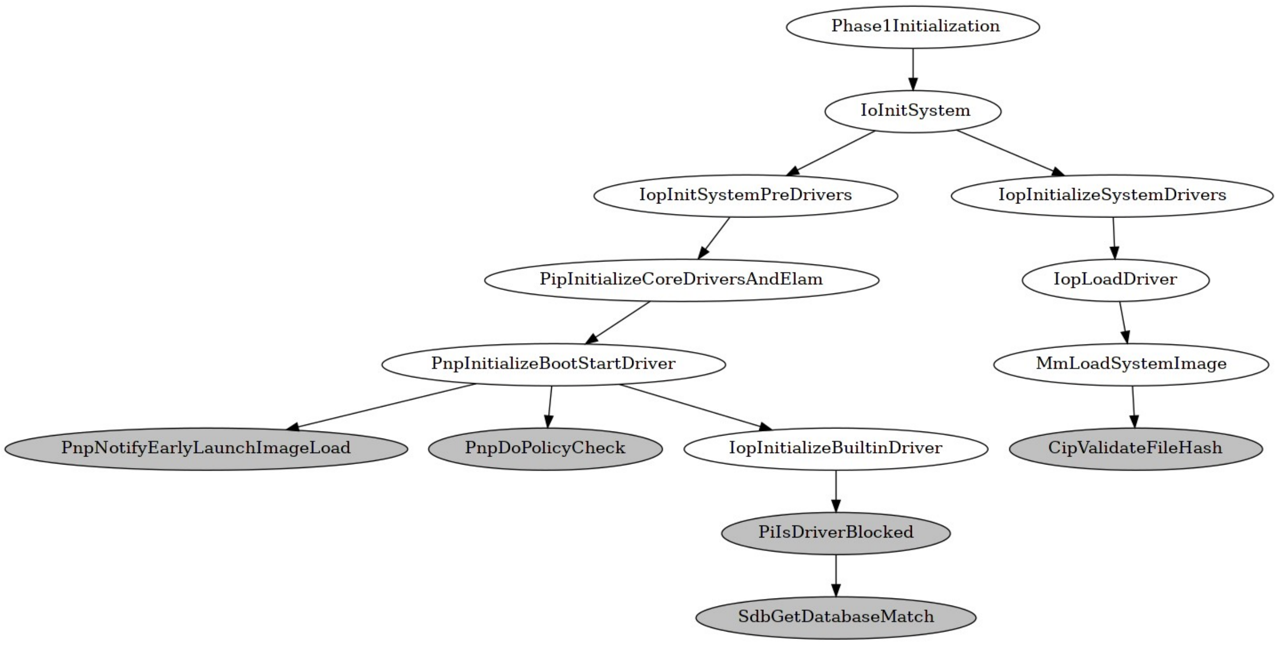


Figure 14: Function stack: Image loading and integrity verification by the Windows kernel

Once the Windows loader has transferred execution control to the kernel (see Section 2.2.2), the kernel is initialized in two phases: Phase 0 and Phase 1 ([Russinovich 2012], Chapter 13). The activities performed in Phase 0 prepare the kernel for image loading and system startup, performed as part of Phase 1. This involves, for example, constructing page tables and allocating internal data structure for memory operations.

Once Phase 0 is finished, the kernel starts Phase 1 (*Phase1Initialization* in Figure 14). In this phase, among other things, the kernel initializes the core I/O infrastructure of the Windows system (*IoInitSystem* in Figure 14). This involves initializing ETW and boot drivers (*IopInitSystemPreDrivers →*

PipInitializeCoreDriversAndElam → *PnpInitializeBootStartDriver* in Figure 14) ([Russinovich 2012], Chapter 13). Boot drivers are loaded and integrity-verified by the Windows loader. In addition, the kernel loads system drivers (*IopInitializeSystemDrivers* → *IopLoadDriver* → *MmLoadSystemImage* in Figure 14).

We observed that the kernel verifies images of boot drivers and system drivers in two distinct manners. In paragraph 'Verification of boot drivers', we describe the verification of boot drivers, and in paragraph 'Verification of system drivers', we describe the verification of system drivers.

Verification of boot drivers

To remind, images of boot drivers are loaded by the Windows loader. The Windows loader verifies the signatures of these drivers such that the root of trust for this verification is the Windows loader itself (see Section 2.2.2). The kernel performs two additional verifications: anti-malware and compatibility verification.

Anti-malware verification of boot drivers is done by a driver specifically developed for that purpose, known as ELAM driver. This driver is initialized before the majority of the other drivers and is used for checking for malware the drivers loaded after it. An ELAM driver implements anti-malware technology, for example, detection of known malicious driver images based on searching a database of properties of such images (e.g., image hashes). This is the ELAM database of malware signatures.

Any vendor may develop an ELAM driver following the development guidelines provided by Microsoft.[36] The Windows 10 system is distributed with an ELAM driver that is part of the Microsoft's Windows Defender anti-malware technology.[37] We discuss the signature database and the image verification process performed by this ELAM driver in detail in Section 2.3. In summary, the ELAM driver verifies the image of each boot driver being initialized by the kernel and categorizes it as: a known good image, a known bad image, a known bad image of a boot-critical driver, or an unknown image. The kernel notifies the ELAM driver that it initializes a boot driver image that needs to be verified. The notification is implemented in the `PnpNotifyEarlyLaunchImageLoad` function (see Figure 14).

Based on the image categorization done by the ELAM driver, the kernel decides whether the image of a given boot driver will be initialized in the `PnpDoPolicyCheck` function (see Figure 14). The kernel brings a decision based on the configuration of the ELAM group policy located at the policy path: `Computer Configuration` → `Administrative Templates` → `System` → `Early Launch Antimalware`. For example, users may configure this policy such that only known good images are initialized.[38]If the ELAM group policy allows for it, the boot driver is initialized (*IopInitializeBuiltinDriver* in Figure 14). This is when verification of the compatibility of the driver takes place.

`IopInitializeBuiltinDriver` invokes `PiIsDriverBlocked`, which searches the Windows compatibility database. This is a database of information identifying known incompatible drivers, stored in multiple database files.[39] `PiIsDriverBlocked` invokes `SdbGetDatabaseMatch` to search the database file located at `%SystemRoot%\AppPatch\drvmain.sdb`. A match indicates that the driver being initialized is not compatible with Windows. In that case, `PiIsDriverBlocked` returns a positive number and the boot driver will not be initialized by the kernel.

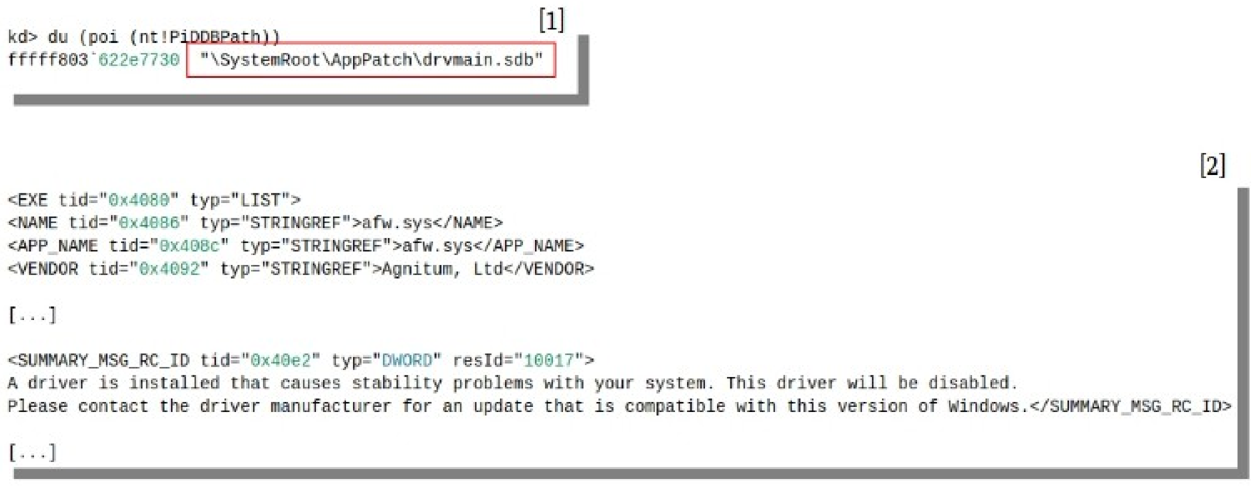
Figure 15 depicts the path to the database file searched in `SdbGetDatabaseMatch` (*[1]* in Figure 15). It also depicts a snippet of the database's content, which we decompiled into Extensible Markup Language (XML) format using the `sdb` tool (*[2]* in Figure 15). The database files comprising the Windows compatibility database are binary files compiled from files in XML format.

36 https://docs.microsoft.com/en-us/windows-hardware/drivers/install/elam-driver-requirements [Retrieved: 22/9/2017]

37 https://docs.microsoft.com/en-us/windows/threat-protection/windows-defender-antivirus/windows-defender-antivirus-in-windows-10; https://blogs.technet.microsoft.com/dubaisec/2016/05/09/elam-driver/ [Retrieved: 22/9/2017]

38 https://blogs.technet.microsoft.com/dubaisec/2016/05/09/elam-driver/ [Retrieved: 22/9/2017]

39 https://msdn.microsoft.com/en-us/library/bb432182(v=vs.85).aspx [Retrieved: 22/9/2017]



```
kd> du (poi (nt!PiDDBPath))
fffff803`622e7730  "\SystemRoot\AppPatch\drvmain.sdb"
```

```
<EXE tid="0x4080" typ="LIST">
<NAME tid="0x4086" typ="STRINGREF">afw.sys</NAME>
<APP_NAME tid="0x408c" typ="STRINGREF">afw.sys</APP_NAME>
<VENDOR tid="0x4092" typ="STRINGREF">Agnitum, Ltd</VENDOR>

[...]

<SUMMARY_MSG_RC_ID tid="0x40e2" typ="DWORD" resId="10017">
A driver is installed that causes stability problems with your system. This driver will be disabled.
Please contact the driver manufacturer for an update that is compatible with this version of Windows.</SUMMARY_MSG_RC_ID>

[...]
```

Figure 15: Snippet of a Windows compatibility database file

Verification of system drivers

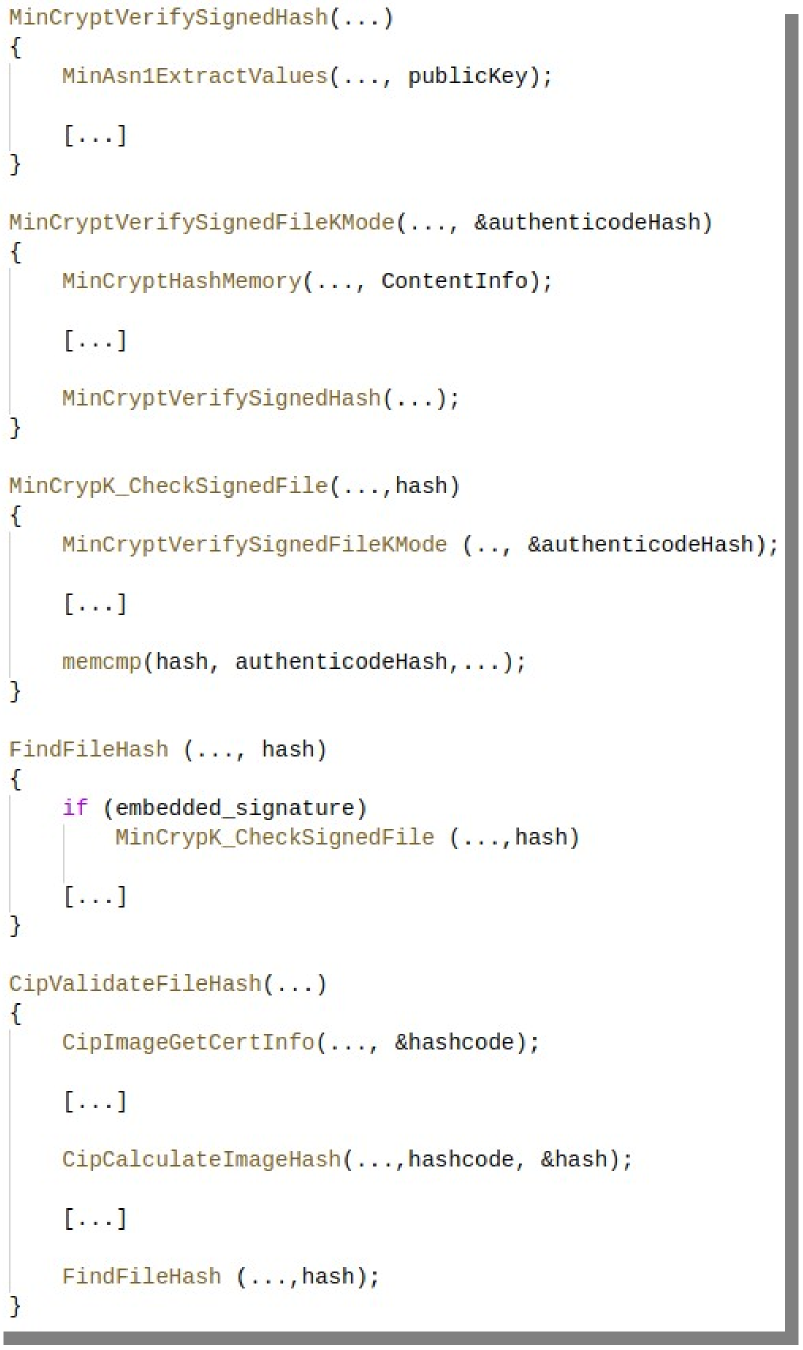
Same as the boot manager and the Windows loader (see Section 2.2.1 and Section 2.2.2), the kernel verifies the integrity of images it loads by verifying their digital signatures. Images of system drivers are signed using the Authenticode technology. In this section, we describe the verification of Authenticode signatures as performed in the context of the kernel. As we previously mentioned, this is conceptually identical to the same process performed in the context of the boot manager and the Windows loader. The image integrity verification process implemented in the boot manager and the Windows loader are simpler than the one described in this section in several aspects. We emphasize this in our discussions.

The kernel verifies the integrity of images using functions implemented in the kernel's code integrity module `%SystemRoot%\System32\ci.dll` ([ERNW WP2], Section 3.4). In this section, we focus only on some aspects of the image integrity verification process implemented in `ci.dll`. We discuss this process in greater detail as part of Work Package 7 [ERNW WP7].

The kernel starts the image integrity verification process by invoking `CipValidateFileHash` (see Figure 14). To identify and analyze the operation of `CipValidateFileHash`, we configured the Windows kernel to perform integrity verification when the `windbg` debugger has established a debugging session with it and not to load images whose integrity cannot be verified. By default, the kernel allows any driver image to load if a kernel debugging session is established. We configured the kernel to stop its execution if the integrity of a driver image cannot be verified by setting the registry key `HKLM\SYSTEM\CurrentControlSet\Control\CI\DebugFlags` to `0x1`.[40] We then installed an unsigned driver, the driver of the `PeerGuardian` firewall[41], and rebooted the system. Since the integrity of this driver could not be verified, the kernel stopped its execution revealing `CipvalidateFileHash` in the function call stack. We emphasize that `CipvalidateFileHash` is invoked deep in the call stack of `MmLoadSystemImage` (see Figure 14). This makes its discovery through static or dynamic analysis of `MmLoadSystemImage` challenging.

Figure 16 depicts pseudo-code of the implementation of `CipValidateFileHash` and of relevant functions it invokes. We describe the implementation of these functions through a running example, where the integrity of the image `%\System32\drivers\crashdmp.sys` is verified.

40 https://docs.microsoft.com/en-us/windows-hardware/drivers/install/appendix-1--enforcing-kernel-mode-signature-verification-in-kernel-debugging-mode [Retrieved: 22/9/2017]
41 http://www.phoenixlabs.org/pg2/ [Retrieved: 22/9/2017]



```
MinCryptVerifySignedHash(...)
{
    MinAsn1ExtractValues(..., publicKey);

    [...]
}

MinCryptVerifySignedFileKMode(..., &authenticodeHash)
{
    MinCryptHashMemory(..., ContentInfo);

    [...]

    MinCryptVerifySignedHash(...);
}

MinCrypK_CheckSignedFile(...,hash)
{
    MinCryptVerifySignedFileKMode (.., &authenticodeHash);

    [...]

    memcmp(hash, authenticodeHash,...);
}

FindFileHash (..., hash)
{
    if (embedded_signature)
        MinCrypK_CheckSignedFile (...,hash)

    [...]
}

CipValidateFileHash(...)
{
    CipImageGetCertInfo(..., &hashcode);

    [...]

    CipCalculateImageHash(...,hashcode, &hash);

    [...]

    FindFileHash (...,hash);
}
```

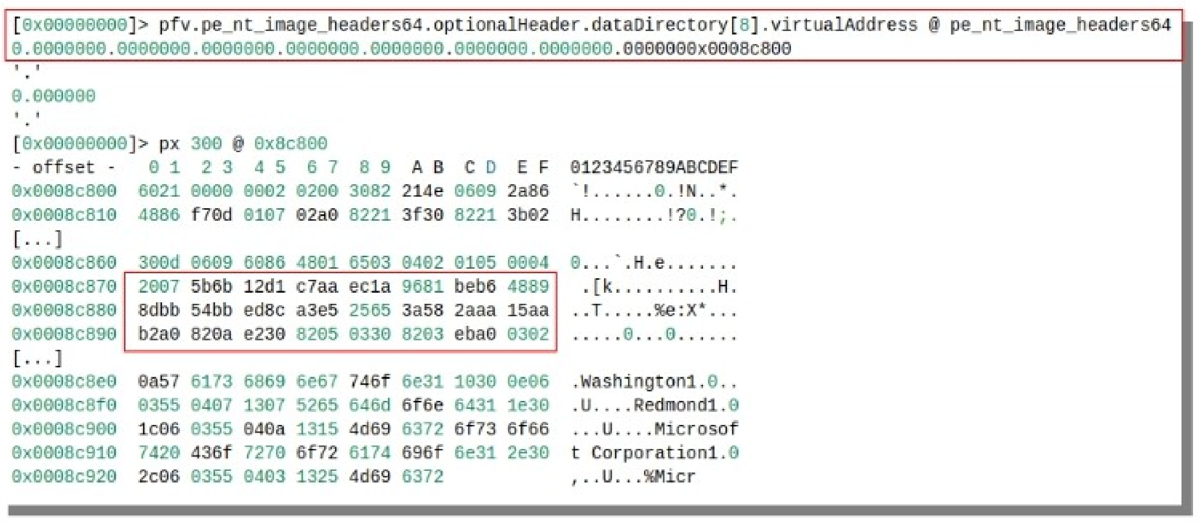
Figure 16: Pseudo-code of CipValidateFileHash and functions it invokes

`CipValidateFileHash` first invokes `CipImageGetCertInfo`. This function extracts Authenticode signature information embedded in `crashdmp.sys`. We emphasize that an Authenticode signature is not merely an encrypted hash value, but a structure containing data. This includes certificates issued by the signers of the image, a calculated hash value of the image, the encrypted digest of the hash value (i.e., the actual signature), the hashing algorithm used for calculation of the hash value, and so on. For more details on the contents of this structure, we refer to the Microsoft Windows Authenticode Portable Executable Signature Format Specification, revision 1.0.

This specification is available on-line at: http://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/Authenticode_PE.docx [Retrieved: 22/9/2017]. This is the latest Authenticode specification at the time of writing.

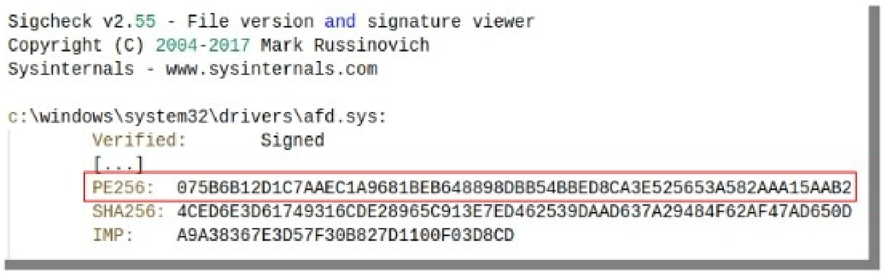
The Authenticode signature may be embedded in the header of an image, in the `Security Directory` section, of the `Data Directories` section, of the `Optional Headers` section [Mic 2008]. As an example, Figure 17 depicts the first 300 bytes of the Authenticode signature embedded in the `%SystemRoot%\System32\drivers\afd.sys` driver image (*[1]* in Figure 17). We extracted the contents of the `Security Directory` section of the image's header, located at address `0x8c800`, using the `radare2` framework. According to ([Mic 2008], Section 'Overview'), the contents displayed in Figure 17 contain a calculated hash value of the image (*075b [...] aab2* in Figure 17), relevant data structures, and a segment of the list of certificates issued by the signers of the image.

[1]



```
[0x00000000]> pfv.pe_nt_image_headers64.optionalHeader.dataDirectory[8].virtualAddress @ pe_nt_image_headers64
0.0000000.0000000.0000000.0000000.0000000.0000000.0000000.0000000x0008c800
'.'
0.000000
'.'
[0x00000000]> px 300 @ 0x8c800
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x0008c800  6021 0000 0002 0200 3082 214e 0609 2a86  `!......0.!N..*.
0x0008c810  4886 f70d 0107 02a0 8221 3f30 8221 3b02  H........!?0.!;.
[...]
0x0008c860  300d 0609 6086 4801 6503 0402 0105 0004  0...`.H.e.......
0x0008c870  2007 5b6b 12d1 c7aa ec1a 9681 beb6 4889   .[k..........H.
0x0008c880  8dbb 54bb ed8c a3e5 2565 3a58 2aaa 15aa  ..T.....%e:X*...
0x0008c890  b2a0 820a e230 8205 0330 8203 eba0 0302  .....0...0......
[...]
0x0008c8e0  0a57 6173 6869 6e67 746f 6e31 1030 0e06  .Washington1.0..
0x0008c8f0  0355 0407 1307 5265 646d 6f6e 6431 1e30  .U....Redmond1.0
0x0008c900  1c06 0355 040a 1315 4d69 6372 6f73 6f66  ...U....Microsof
0x0008c910  7420 436f 7270 6f72 6174 696f 6e31 2e30  t Corporation1.0
0x0008c920  2c06 0355 0403 1325 4d69 6372            ,..U...%Micr
```

[2]



```
Sigcheck v2.55 - File version and signature viewer
Copyright (C) 2004-2017 Mark Russinovich
Sysinternals - www.sysinternals.com

c:\windows\system32\drivers\afd.sys:
        Verified:       Signed
        [...]
        PE256:   075B6B12D1C7AAEC1A9681BEB648898DBB54BBED8CA3E525653A582AAA15AAB2
        SHA256:  4CED6E3D61749316CDE28965C913E7ED462539DAAD637A29484F62AF47AD650D
        IMP:     A9A38367E3D57F30B827D1100F03D8CD
```

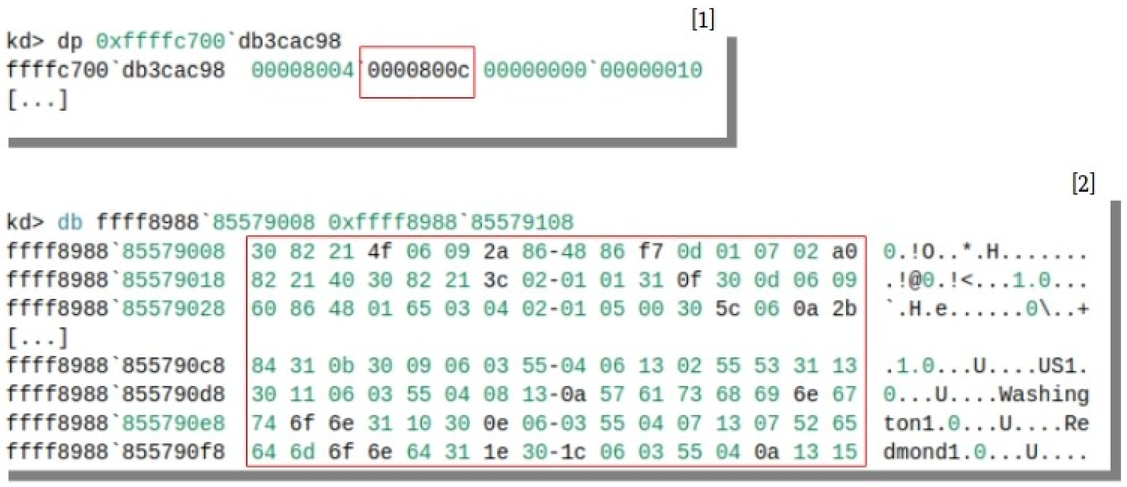
Figure 17: An Authenticode signature and a PE hash value

The hash value is calculated only on specific sections of the signed image, as defined in ([Mic 2008], Section 'Calculating the PE Image Hash'). This value is referred to as the portable executable (PE) hash value. The PE hash value depicted in Figure 17 can also be extracted using the `Sigcheck` utility, part of the `Sysinternals` suite (*[2]* in Figure 17). The section of the Authenticode signature that contains the PE hash value is referred to as `ContentInfo` (see [Mic 2008], Figure 1).

An Authenticode signature may not be embedded in an image file, but stored in a separate digitally signed file, known as catalog file.[42] We focus in this section on integrity verification in scenarios where the Authenticode signature is embedded in an image file. We discuss other image verification scenarios as part of Work Package 7 [ERNW WP7].

42 https://docs.microsoft.com/en-us/windows-hardware/drivers/install/authenticode [Retrieved: 22/9/2017]

`CipImageGetCertInfo` extracts a code for the hashing algorithm used for calculation of the PE hash value (i.e., a hash code, *hashcode* in Figure 16), the `ContentInfo` section (*ContentInfo* in Figure 16), and the list of certificates issued by the signers of the image. Figure 18 depicts the extracted code (*[1]* in Figure 18), where a code of *800C* denotes the Secure Hash Algorithm (SHA)-256 algorithm. Figure 18 also depicts a portion of the list of signers of the image (*[2]* in Figure 18).



```
kd> dp 0xffffc700`db3cac98
ffffc700`db3cac98  00008004 0000800c 00000000`00000010
[...]
```

```
kd> db ffff8988`85579008 0xffff8988`85579108
ffff8988`85579008  30 82 21 4f 06 09 2a 86-48 86 f7 0d 01 07 02 a0  0.!O..*.H.......
ffff8988`85579018  82 21 40 30 82 21 3c 02-01 01 31 0f 30 0d 06 09  .!@0.!<...1.0...
ffff8988`85579028  60 86 48 01 65 03 04 02-01 05 00 30 5c 06 0a 2b  `.H.e......0\..+
[...]
ffff8988`855790c8  84 31 0b 30 09 06 03 55-04 06 13 02 55 53 31 13  .1.0...U....US1.
ffff8988`855790d8  30 11 06 03 55 04 08 13-0a 57 61 73 68 69 6e 67  0...U....Washing
ffff8988`855790e8  74 6f 6e 31 10 30 0e 06-03 55 04 07 13 07 52 65  ton1.0...U....Re
ffff8988`855790f8  64 6d 6f 6e 64 31 1e 30-1c 06 03 55 04 0a 13 15  dmond1.0...U....
```

Figure 18: A hash code and a list of signers

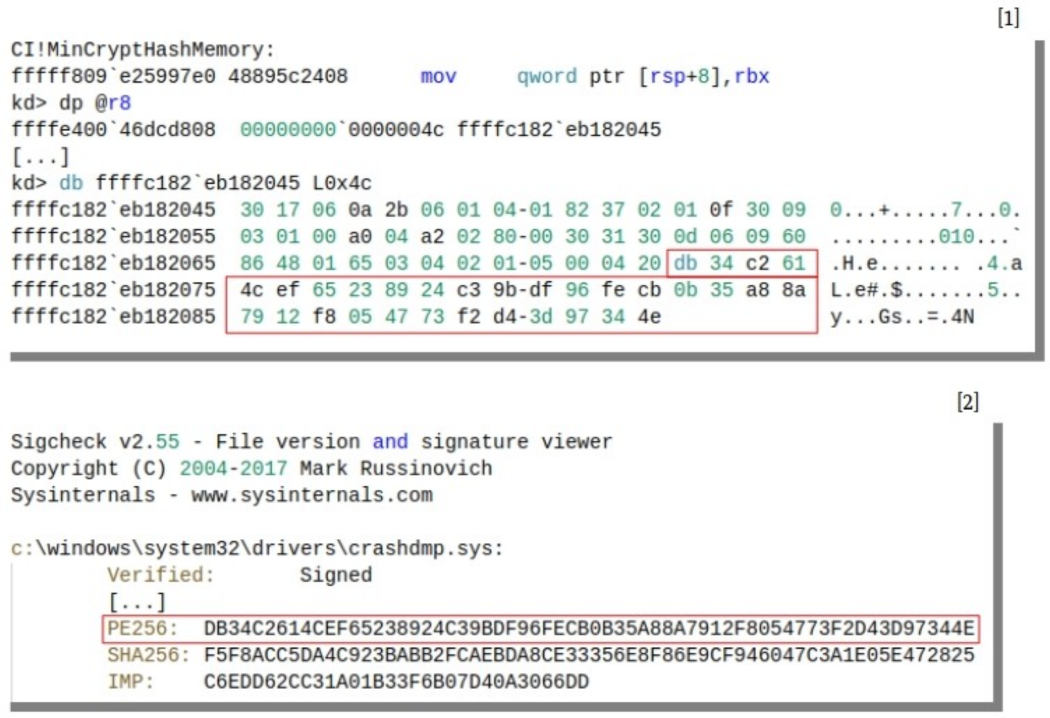
Once the hash code has been extracted from the embedded Authenticode signature, the `CipCalculateImageHash` function calculates the PE hash value (*hash* in Figure 16) using the hash algorithm referenced by the extracted code (i.e., SHA-256 in our scenario). Then, the function `FindFileHash` is invoked. If an Authenticode signature is embedded in the image being verified (*embedded_signature* in Figure 16), it invokes `MinCryptK_CheckSignedFile`. Alternatively, `FindFileHash` queries catalog files by issuing remote procedure calls (RPCs). Authenticode signatures are embedded in all of the images of boot drivers and other images loaded by the boot manager and the Windows loader for performance reasons.[43] Therefore, the image verification procedures implemented in the boot manager and the Windows loader typically do not query catalog files.

`MinCryptK_CheckSignedFile` verifies the integrity of the Authenticode signature, that is, it verifies the integrity of the `ContentInfo` section. A signature of this section is stored in the Authenticode signature, in a section referred to as the `SignerInfo` ([Mic 2008], Figure 1). If the `ContentInfo` section of the Authenticode signature has not been tampered with, the calculated PE hash value in `CipCalculateImageHash` (*hash* in Figure 16) is compared with the PE hash value stored in the Authenticode signature (*authenticodeHash* and *memcmp* in Figure 16). The latter is extracted from the Authenticode signature in the `MinCryptVerifySignedFileKMode` function (see Figure 16). If the compared hash values are identical, the image is considered authentic.

The verification of the integrity of the Authenticode signature is done in `MinCryptVerifySignedFileKMode`. It first calculates a hash of the `ContentInfo` section (*MinCryptHashMemory* in Figure 16). Figure 19 depicts the `ContentInfo` section of the Authenticode signature embedded in `crashdmp.sys` when passed to `MinCryptHashMemory` for hashing (*[1]* in Figure 19). That the hashed content is that of `ContentInfo`, can be verified by identifying the PE hash value in it. We extracted the PE hash value from `crashdmp.sys` using the `sigcheck` utility (*[2]* in Figure 19), observing the same value as that stored in the hashed content.

43 https://docs.microsoft.com/en-us/windows-hardware/drivers/install/embedded-signatures-in-a-driver-file [Retrieved: 22/9/2017]

After hashing `ContentInfo`, `MinCryptVerifySignedFileKMode` verifies its signature (*MinCryptVerifySignedHash* in Figure 16). The decryption of this signature is done using a public key that is stored as part of the signer's certificate, placed in the Authenticode signature and processed in the `MinAsn1ExtractValues` function (*publicKey* in Figure 16). The signer's certificate is verified in the `MinCryptVerifyCertificateWithPolicy2` function, ultimately against its root certificate. This verification involves verifying the certificate's signature using the root certificate public key by invoking `MinCryptVerifySignedHash` – this public key is stored as part of a root certificate issued by Microsoft. The root certificate is hardcoded in the `ci.dll` library file, in a data structure informally referred to as the `root table`. Figure 20 depicts the public key stored in the `root table` of `ci.dll` (*[1]* in Figure 20). We extracted the public key from `ci.dll` using the `radare2` framework. In addition, Figure 20 depicts the same public key, however, stored in a variable referenced in `MinAsn1ExtractValues` (*[2]* in Figure 20). The fact that hardcoded contents of the `root table` structure are used for verification of signers' certificates shows that the root of trust for verifying the integrity of images loaded by the kernel is `ci.dll`.



```
CI!MinCryptHashMemory:
fffff809`e25997e0 48895c2408      mov     qword ptr [rsp+8],rbx
kd> dp @r8
ffffe400`46dcd808  00000000`0000004c ffffc182`eb182045
[...]
kd> db ffffc182`eb182045 L0x4c
ffffc182`eb182045  30 17 06 0a 2b 06 01 04-01 82 37 02 01 0f 30 09  0...+.....7...0.
ffffc182`eb182055  03 01 00 a0 04 a2 02 80-00 30 31 30 0d 06 09 60  .........010...`
ffffc182`eb182065  86 48 01 65 03 04 02 01-05 00 04 20 db 34 c2 61  .H.e....... .4.a
ffffc182`eb182075  4c ef 65 23 89 24 c3 9b-df 96 fe cb 0b 35 a8 8a  L.e#.$.......5..
ffffc182`eb182085  79 12 f8 05 47 73 f2 d4-3d 97 34 4e              y...Gs..=.4N
```

```
Sigcheck v2.55 - File version and signature viewer
Copyright (C) 2004-2017 Mark Russinovich
Sysinternals - www.sysinternals.com

c:\windows\system32\drivers\crashdmp.sys:
        Verified:       Signed
        [...]
        PE256:   DB34C2614CEF65238924C39BDF96FECB0B35A88A7912F8054773F2D43D97344E
        SHA256:  F5F8ACC5DA4C923BABB2FCAEBDA8CE33356E8F86E9CF946047C3A1E05E472825
        IMP:     C6EDD62CC31A01B33F6B07D40A3066DD
```

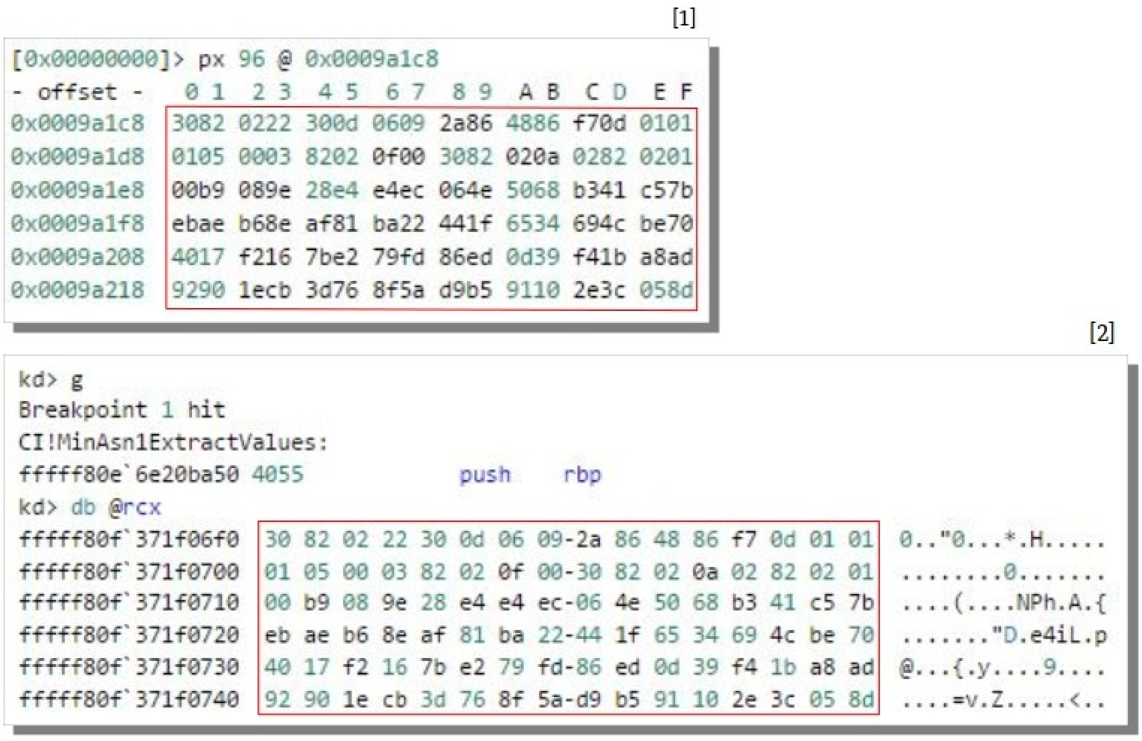
Figure 19: crashdmp.sys: The ContentInfo section and the PE hash value

We emphasize that the description of the kernel's integrity verification process we provided in this section presents only some relevant aspects of it. For example, certificates issued by the signers of an image and stored as part of an Authenticode signature, are also verified as described in ([Mic 2008], Section 'Certificate Processing'). We did not discuss this verification in detail this section.

We did not observe the TPM playing any role in the image integrity verification process. We observed that the root of trust for verifying Authenticode signatures is the `ci.dll` file (see Figure 20) and hash calculations are implemented in software.

When the kernel is finished with loading of images of system drivers, it initializes the Windows subsystem, and starts the session manager `smss.exe` ([Russinovich 2012], Table 13-1).

TPM usage We set breakpoints at functions for TPM command processing implemented in the TPM driver `tpm.sys` and the TPM export driver `tbs.sys`. We observed that none of the functions of the TPM export driver were invoked. Those of `tpm.sys` were invoked for device initialization. Once loaded, we observed TPM commands being sent from the Windows subsystem and various system services, such as the session manager `smss.exe` and the local security authority `lsa.exe`. The entities communicating with the TPM can be identified with the script we presented in Section 2.1.3. The purpose of the TPM commands we observed after the kernel was loaded is not within the scope of this work package.



[1]

```
[0x00000000]> px 96 @ 0x0009a1c8
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F
0x0009a1c8  3082 0222 300d 0609 2a86 4886 f70d 0101
0x0009a1d8  0105 0003 8202 0f00 3082 020a 0282 0201
0x0009a1e8  00b9 089e 28e4 e4ec 064e 5068 b341 c57b
0x0009a1f8  ebae b68e af81 ba22 441f 6534 694c be70
0x0009a208  4017 f216 7be2 79fd 86ed 0d39 f41b a8ad
0x0009a218  9290 1ecb 3d76 8f5a d9b5 9110 2e3c 058d
```

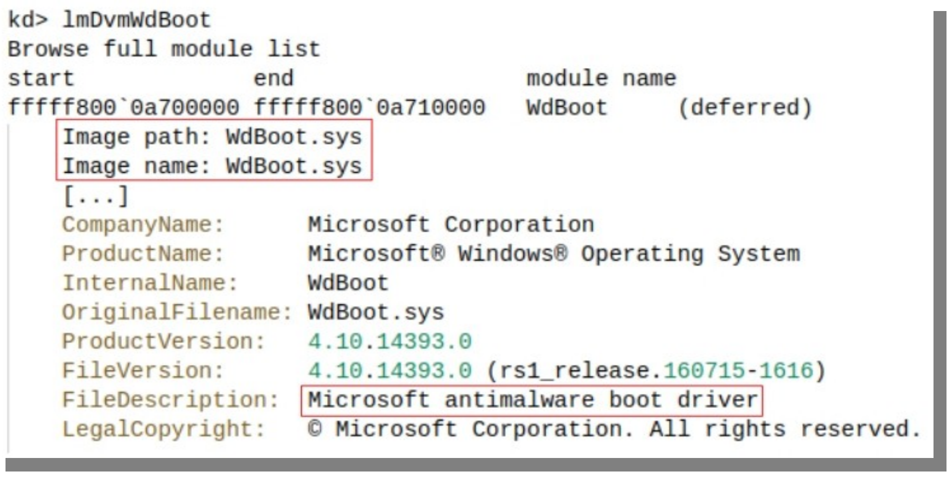
[2]

```
kd> g
Breakpoint 1 hit
CI!MinAsn1ExtractValues:
fffff80e`6e20ba50 4055            push    rbp
kd> db @rcx
fffff80f`371f06f0  30 82 02 22 30 0d 06 09-2a 86 48 86 f7 0d 01 01  0.."0...*.H.....
fffff80f`371f0700  01 05 00 03 82 02 0f 00-30 82 02 0a 02 82 02 01  ........0.......
fffff80f`371f0710  00 b9 08 9e 28 e4 e4 ec-06 4e 50 68 b3 41 c5 7b  ....(....NPh.A.{
fffff80f`371f0720  eb ae b6 8e af 81 ba 22-44 1f 65 34 69 4c be 70  ......."D.e4iL.p
fffff80f`371f0730  40 17 f2 16 7b e2 79 fd-86 ed 0d 39 f4 1b a8 ad  @...{.y....9....
fffff80f`371f0740  92 90 1e cb 3d 76 8f 5a-d9 b5 91 10 2e 3c 05 8d  ....=v.Z.....<..
```

Figure 20: A public key of a root certificate stored in ci.dll

2.3 The Windows Defender ELAM Driver

The Windows Defender ELAM driver is implemented in the `%SystemRoot\System32\drivers\WdBoot.sys` executable. It is loaded by the Windows loader and initialized in the kernel (*PipInitializeCoreDriversAndElam* in Figure 14). It is unloaded by the kernel when it has checked for malware all boot drivers. Figure 21 depicts the information on the Windows Defender ELAM Driver that the kernel maintains after loading it.



```
kd> lmDvmWdBoot
Browse full module list
start             end                 module name
fffff800`0a700000 fffff800`0a710000   WdBoot     (deferred)
    Image path: WdBoot.sys
    Image name: WdBoot.sys
    [...]
    CompanyName:      Microsoft Corporation
    ProductName:      Microsoft® Windows® Operating System
    InternalName:     WdBoot
    OriginalFilename: WdBoot.sys
    ProductVersion:   4.10.14393.0
    FileVersion:      4.10.14393.0 (rs1_release.160715-1616)
    FileDescription:  Microsoft antimalware boot driver
    LegalCopyright:   © Microsoft Corporation. All rights reserved.
```

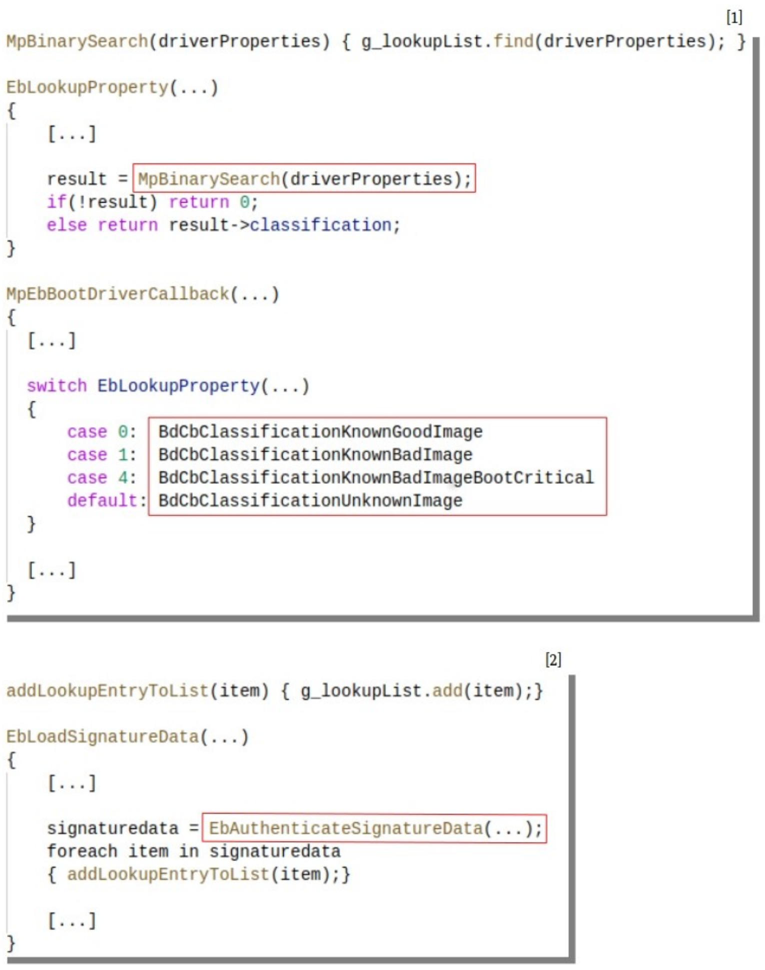
Figure 21: Information on the Windows Defender ELAM driver

The ELAM driver performs anti-malware verification in its function `MpEbBootDriverCallback`. This is a callback function invoked when the kernel notifies the ELAM driver that a boot driver image is to be initialized (see Section 2.2.3).

Figure 22 depicts a pseudo-code of the implementation of `MpEbBootDriverCallback` and relevant functions that it invokes (*[1]* in Figure 22). `MpEbBootDriverCallback` invokes `EbLookupProperty`, which sets an integer code indicating the category of the verified image. The analysis as well as the pseudo-code is based on the Windows Defender ELAM driver. This driver works in a blacklist-fashion (besides the mandatory verification of the driver signature); Non-Microsoft implementations may work in different ways.

Adopting the Microsoft terminology on this topic, an image may be categorized as:

- a known good image (code value *BdCbClassificationKnownGoodImage* in Figure 22);
- a known bad image (code value *BdCbClassificationKnownBadImage* in Figure 22);
- a known bad image of a boot-critical driver (code value *BdCbClassificationKnownBadImageBootCritical* in Figure 22); or
- an unknown image (code value *BdCbClassificationUnknownImage* in Figure 22).



[1]

```
MpBinarySearch(driverProperties) { g_lookupList.find(driverProperties); }

EbLookupProperty(...)
{
    [...]

    result = MpBinarySearch(driverProperties);
    if(!result) return 0;
    else return result->classification;
}

MpEbBootDriverCallback(...)
{
  [...]

  switch EbLookupProperty(...)
  {
      case 0:  BdCbClassificationKnownGoodImage
      case 1:  BdCbClassificationKnownBadImage
      case 4:  BdCbClassificationKnownBadImageBootCritical
      default: BdCbClassificationUnknownImage
  }

  [...]
}
```

[2]

```
addLookupEntryToList(item) { g_lookupList.add(item);}

EbLoadSignatureData(...)
{
    [...]

    signaturedata = EbAuthenticateSignatureData(...);
    foreach item in signaturedata
    { addLookupEntryToList(item);}

    [...]
}
```

Figure 22: Pseudo-code of MpEbBootDriverCallback of the Windows Defender Implementation and functions that it invokes

We refer to these categories as ELAM image categories. `MpEbBootDriverCallback` passes the integer set by `EbLookupProperty` back to the kernel. Based on its value, the kernel decides whether a given boot driver will be initialized (see Section 2.2.3).

`EbLookupProperty` performs a binary search for specific properties of the verified driver image, for example, file hash values, in the ELAM database of malware signatures (*MpBinarySearch*, *driverProperties* in Figure 22). Each entry of this database is stored in an array (*g_lookupList* in Figure 22). This array is filled when the signature database is loaded into the driver's memory space.

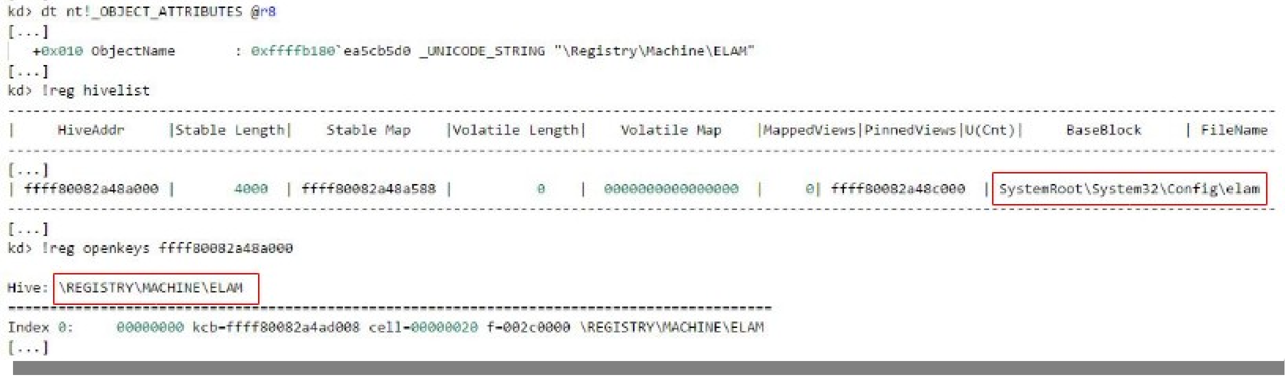
The loading of the ELAM signature database is done when the driver is initialized and is performed in two steps (*[2]* in Figure 22): First, the function `MpEbLoadSignatures` (not depicted in 22) using the `ZwOpenKey` routine[44] loads the signature database. This database is stored in the form of a registry hive in the system's registry. Then, the function `EbLoadSignatureData` verifies the integrity of the signature database by invoking `EbAuthenticateSignatureData`. It also stores each entry of the database (*item* in Figure 22) in the previously mentioned array of database entries (*g_lookupList, addLookupEntryToList* in Figure 22). The stored entries are later used in `EbLookupProperty`.

Figure 23 depicts this database loading operation. The ELAM signature database is loaded in the registry as the registry hive `HKEY_LOCAL_MACHINE\ELAM` (*\REGISTRY\MACHINE\ELAM* in Figure 23). We observed that this registry hive is unloaded when the ELAM driver is unloaded. The registry hive of the ELAM signature database is stored on the file-system at `%SystemRoot%\System32\Config\elam` (see Figure 23). This enabled us to analyze the contents of the registry hive after the ELAM driver and its signature database have been unloaded.

The ELAM signature database is stored as binary data. This data has a specific format structuring it into multiple entries. Each entry, among other things, consists of a malware signature, an entry type, and a trust level. A malware signature is often, but not necessarily, a hash value. An entry type is a code value indicating the type of data representing a malware signature. A trust level is a code value corresponding to the ELAM image categories known good image, known bad image, known bad image of a boot-critical driver, and unknown image. The trust level specifies the ELAM category of image associated with the malware signature.

We observed the above by parsing the signature database of the Windows Defender ELAM driver using a parser we developed. The code of this parser, implemented in the Python programming language, is placed in the Appendix, section 'ELAM Database Parser'. Figure 24 depicts the output of the parser, where `EntryType`, `TrustLevel`, and `data` mark an entry type, a trust level, and a malware signature, respectively. An example entry in the ELAM signature database is the hash value `F4 27 86 [...] 9F 8E`, which is a hash value of a known bad image.

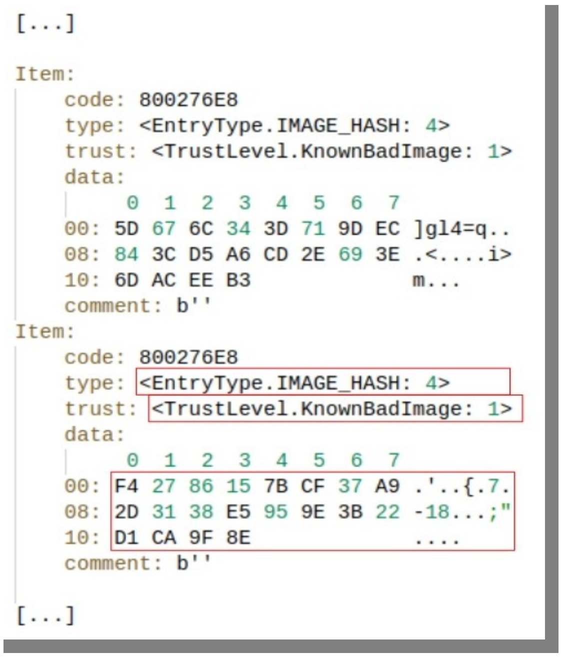
As we previously mentioned, the integrity of the ELAM signature database is verified in `EbAuthenticateSignatureData`. The signature database of the Windows Defender ELAM driver is digitally signed for preventing unauthorized modifications. Figure 25 depicts pseudo-code of the



```
WdBoot!MpEbLoadSignatures+0x72:
fffff800`defeb73e ff154ca9ffff    call    qword ptr [WdBoot!_imp_ZwOpenKey (fffff800`defe6090)]
kd> t
[...]
kd> dt nt!_OBJECT_ATTRIBUTES @r8
[...]
|   +0x010 ObjectName       : 0xffffb180`ea5cb5d0 _UNICODE_STRING "\Registry\Machine\ELAM"
[...]
kd> !reg hivelist
-------------------------------------------------------------------------------------------------------------------------------------------------
|     HiveAddr     |Stable Length|    Stable Map    |Volatile Length|    Volatile Map    |MappedViews|PinnedViews|U(Cnt)|     BaseBlock     | FileName
-------------------------------------------------------------------------------------------------------------------------------------------------
[...]
| ffff80082a48a000 |       4000  | ffff80082a48a588 |            0  | 0000000000000000 |          0|  ffff80082a48c000  | SystemRoot\System32\Config\elam
-------------------------------------------------------------------------------------------------------------------------------------------------
[...]
kd> !reg openkeys ffff80082a48a000

Hive: \REGISTRY\MACHINE\ELAM
===============================================================================================
Index 0:     00000000 kcb-ffff80082a4ad008 cell-00000020 f-002c0000 \REGISTRY\MACHINE\ELAM
[...]
```

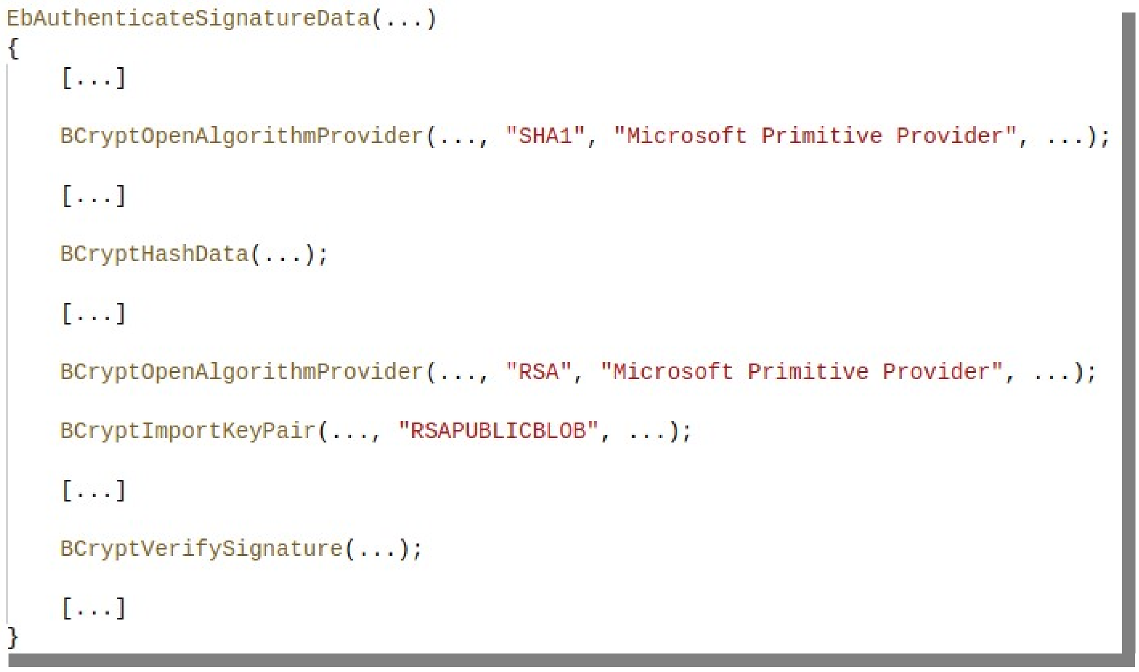
Figure 23: The ELAM signature database in the registry

implementation of `EbAuthenticateSignatureData`. This function verifies the signature of the ELAM signature database using the kernel implementation of the CNG library (see Section 2.1.1). The ELAM signature database may be updated by system services, such as Windows Update. These services mount and modify the registry hive `HKEY_LOCAL_MACHINE\ELAM`. However, the updated signature database has to be properly signed since its integrity is verified by the ELAM driver as discussed in this section.[45]



```
[...]

Item:
    code: 800276E8
    type: <EntryType.IMAGE_HASH: 4>
    trust: <TrustLevel.KnownBadImage: 1>
    data:
         0  1  2  3  4  5  6  7
    00: 5D 67 6C 34 3D 71 9D EC ]gl4=q..
    08: 84 3C D5 A6 CD 2E 69 3E .<....i>
    10: 6D AC EE B3             m...
    comment: b''
Item:
    code: 800276E8
    type: <EntryType.IMAGE_HASH: 4>
    trust: <TrustLevel.KnownBadImage: 1>
    data:
         0  1  2  3  4  5  6  7
    00: F4 27 86 15 7B CF 37 A9 .'..{.7.
    08: 2D 31 38 E5 95 9E 3B 22 -18...;"
    10: D1 CA 9F 8E             ....
    comment: b''

[...]
```

Figure 24: The ELAM signature database



```
EbAuthenticateSignatureData(...)
{
    [...]

    BCryptOpenAlgorithmProvider(..., "SHA1", "Microsoft Primitive Provider", ...);

    [...]

    BCryptHashData(...);

    [...]

    BCryptOpenAlgorithmProvider(..., "RSA", "Microsoft Primitive Provider", ...);

    BCryptImportKeyPair(..., "RSAPUBLICBLOB", ...);

    [...]

    BCryptVerifySignature(...);

    [...]
}
```

Figure 25: Pseudo-code of EbAuthenticateSignatureData

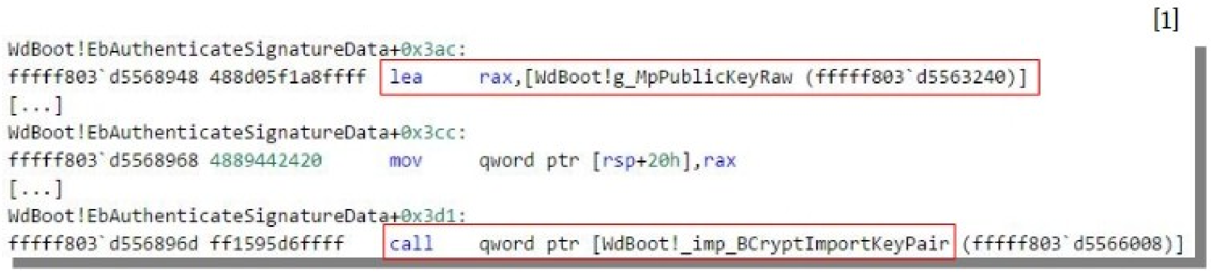
45 https://docs.microsoft.com/en-us/windows-hardware/drivers/install/elam-driver-requirements [Retrieved: 22/9/2017]

`EbAuthenticateSignatureData` first hashes the signature database (*BcryptHashData*[46] in Figure 25) using the SHA-1 algorithm and the software-implemented `Microsoft Primitive Provider` (*BcryptOpenAlgorithmProvider* in Figure 25). It then imports a public key (*BCryptImportKeyPair, RSAPUBLICBLOB*[47] in Figure 25). Finally, `EbAuthenticateSignatureData` uses the `Microsoft Primitive Provider` and the imported public key to decrypt the signature of the ELAM signature database and compare it with the previously calculated hash value of this database (*BCryptVerifySignature*[48] in Figure 25).

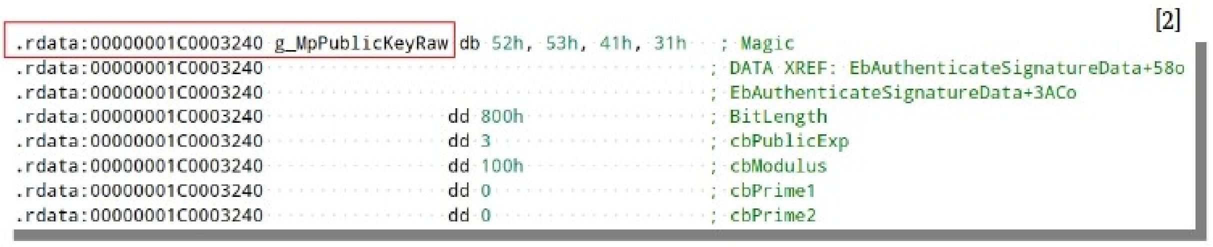
We observed that `EbAuthenticateSignatureData` imports a public key hardcoded in the `WdBoot.sys` driver executable, stored in the variable `g_MpPublicKeyRaw`. This indicates that the root of trust for verifying the malware signature database used by the Windows ELAM driver is the driver itself. Figure 26 depicts the passing of `g_MpPublicKeyRaw` to `BcryptImportKeyPair` for importing (*[1]* in Figure 26). It also depicts the declaration of `g_MpPublicKeyRaw` in the `WdBoot.sys` driver executable, which we observed with the `IDA` disassembler (*[2]* in Figure 26).

According to the Microsoft's ELAM driver development guidelines, vendors of ELAM drivers may store the ELAM signature database and related relevant data in the registry keys `Measured`, `Policy`, and/or `Config`, under the registry hive `HKEY_LOCAL_MACHINE\ELAM`. We observed that the Windows Defender ELAM driver stores its database in `Measured`.

The registry keys `Measured`, `Policy`, and `Config` are measured by the integrity measurement mechanism of Windows 10. This measurement allows for an additional verification of the integrity of the ELAM signature database by a remote entity. We discuss the integrity measurement mechanism implemented in Windows 10 in Section 2.4.


[1]

```
WdBoot!EbAuthenticateSignatureData+0x3ac:
fffff803`d5568948 488d05f1a8ffff  lea     rax,[WdBoot!g_MpPublicKeyRaw (fffff803`d5563240)]
[...]
WdBoot!EbAuthenticateSignatureData+0x3cc:
fffff803`d5568968 4889442420      mov     qword ptr [rsp+20h],rax
[...]
WdBoot!EbAuthenticateSignatureData+0x3d1:
fffff803`d556896d ff1595d6ffff    call    qword ptr [WdBoot!_imp_BCryptImportKeyPair (fffff803`d5566008)]
```


[2]

```
.rdata:00000001C0003240 g_MpPublicKeyRaw db 52h, 53h, 41h, 31h   ; Magic
.rdata:00000001C0003240                                          ; DATA XREF: EbAuthenticateSignatureData+58o
.rdata:00000001C0003240                                          ; EbAuthenticateSignatureData+3ACo
.rdata:00000001C0003240                  dd 800h                 ; BitLength
.rdata:00000001C0003240                  dd 3                    ; cbPublicExp
.rdata:00000001C0003240                  dd 100h                 ; cbModulus
.rdata:00000001C0003240                  dd 0                    ; cbPrime1
.rdata:00000001C0003240                  dd 0                    ; cbPrime2
```

Figure 26: The public key used for verification of the Windows Defender ELAM signature database

2.4 Integrity Measurement

In this section, we discuss the integrity measurement mechanism of Windows 10 and the role that the TPM plays as part of it. This mechanism, among other things, implements the production of measurement data. This involves calculation of hashes of critical executable files or of code sequences at every system startup. It

46 https://msdn.microsoft.com/en-us/library/windows/desktop/aa375468(v=vs.85).aspx [Retrieved: 22/9/2017]
47 https://msdn.microsoft.com/en-us/library/windows/desktop/aa375472(v=vs.85).aspx [Retrieved: 22/9/2017]
48 https://msdn.microsoft.com/de-de/library/windows/desktop/aa375515(v=vs.85).aspx [Retrieved: 22/9/2017]

also involves the storage of these hashes and relevant related data in the TPM device and in log files for later analysis.

The analysis of measurement data is normally performed by a trusted remote platform, a platform different than the one where hashes have been calculated. The remote platform can be reached over a secure network connection. A typical analysis of measurement data consists of, for example, comparing the most recently calculated hashes with hashes calculated at a previous time, or with hashes known as good hashes. A mismatch in the hash values indicates platform corruption. The verification of platform integrity by a remote platform is known as remote attestation.

Figure 27 depicts the architecture of the integrity measurement mechanism implemented in Windows 10. During the booting process of a given platform (*Platform* in Figure 27, see Section 2.2), the UEFI firmware, the boot manager, and the Windows loader measure relevant entities. They then store the produced measurement data in the PCRs of the TPM installed on the platform (*measured into* in Figure 27). In Figure 27, we refer to the UEFI firmware, the boot manager, and the Windows loader as the pre-operating system (OS) environment (*Pre-OS* in Figure 27). We discuss in greater detail the measurement performed in the pre-OS environment in paragraph ’Implementation of integrity measurement‘ of this section.

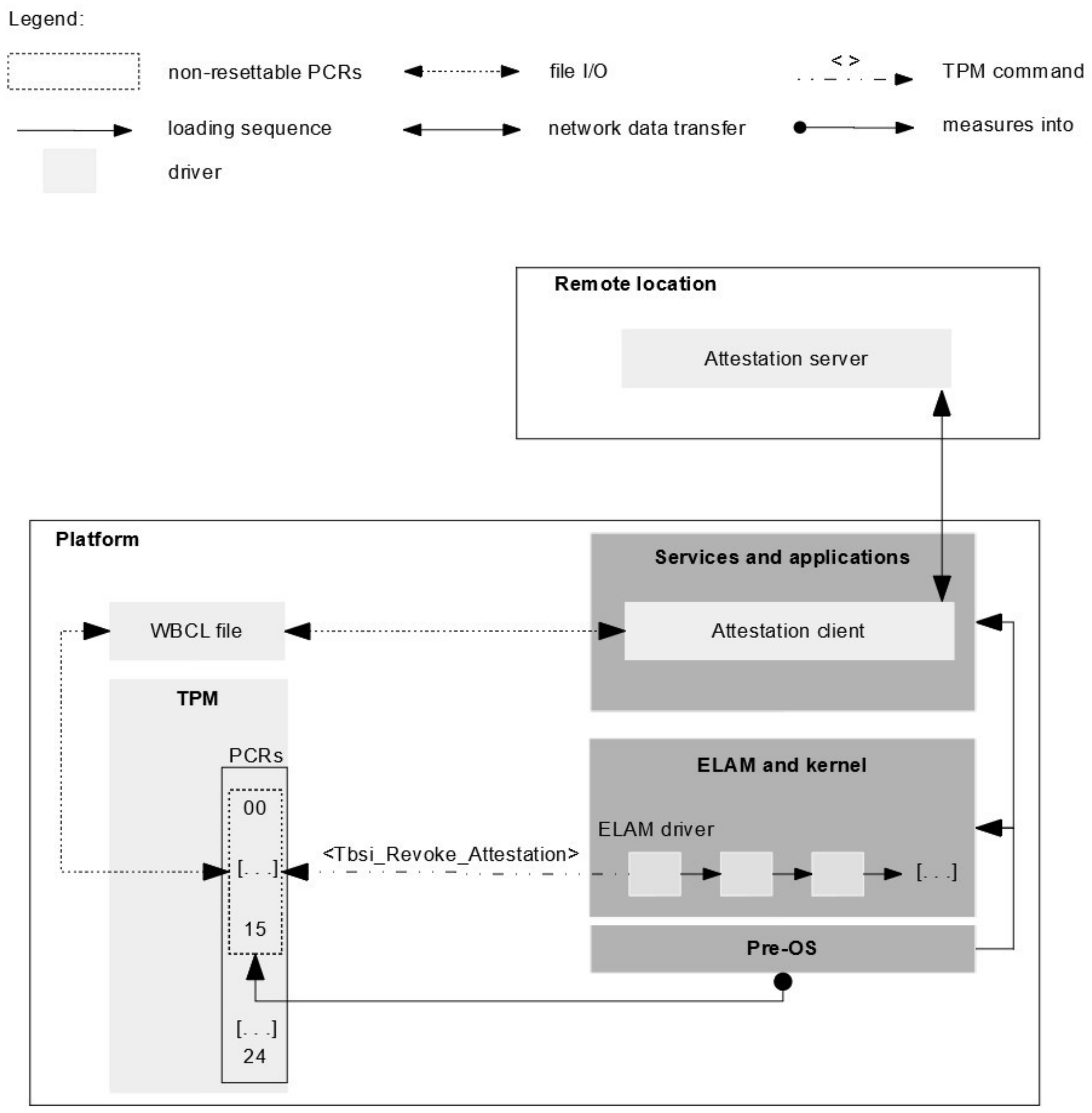


Figure 27: The architecture of the integrity measurement mechanism of Windows 10

The platform stores the hashes calculated in the pre-OS environment and relevant related data into a context known as the WBCL ([Thom 2016], Section 'Windows Boot Configuration Log'). A new WBCL is generated at every system startup since this is when new integrity measurements are made ([Thom 2016], Section 'Windows Boot Configuration Log'). Each WBCL is archived into a log file, referred to as the WBCL file. WBCL files are stored in the `%SystemRoot\Logs\MeasuredBoot` directory. We discuss the content and format of WBCL files in paragraph 'Windows Boot Configuration Log' of this section.

The Windows loader loads the kernel, which may implement ELAM technology in the form of an ELAM driver (*ELAM and kernel* in Figure 27, see Section 2.3). In case it detects the loading of a malicious driver, the ELAM driver may revoke the current WBCL using the `Tbsi_Revoke_Attestation` function of the TBS library ([Thom 2016], Section 'Invalidating the System Trust State').[49,50] Among other things, a revocation of a WBCL consists of storing an unspecified value in the PCR with index 12. This indicates system corruption to the remote entity verifying platform integrity.

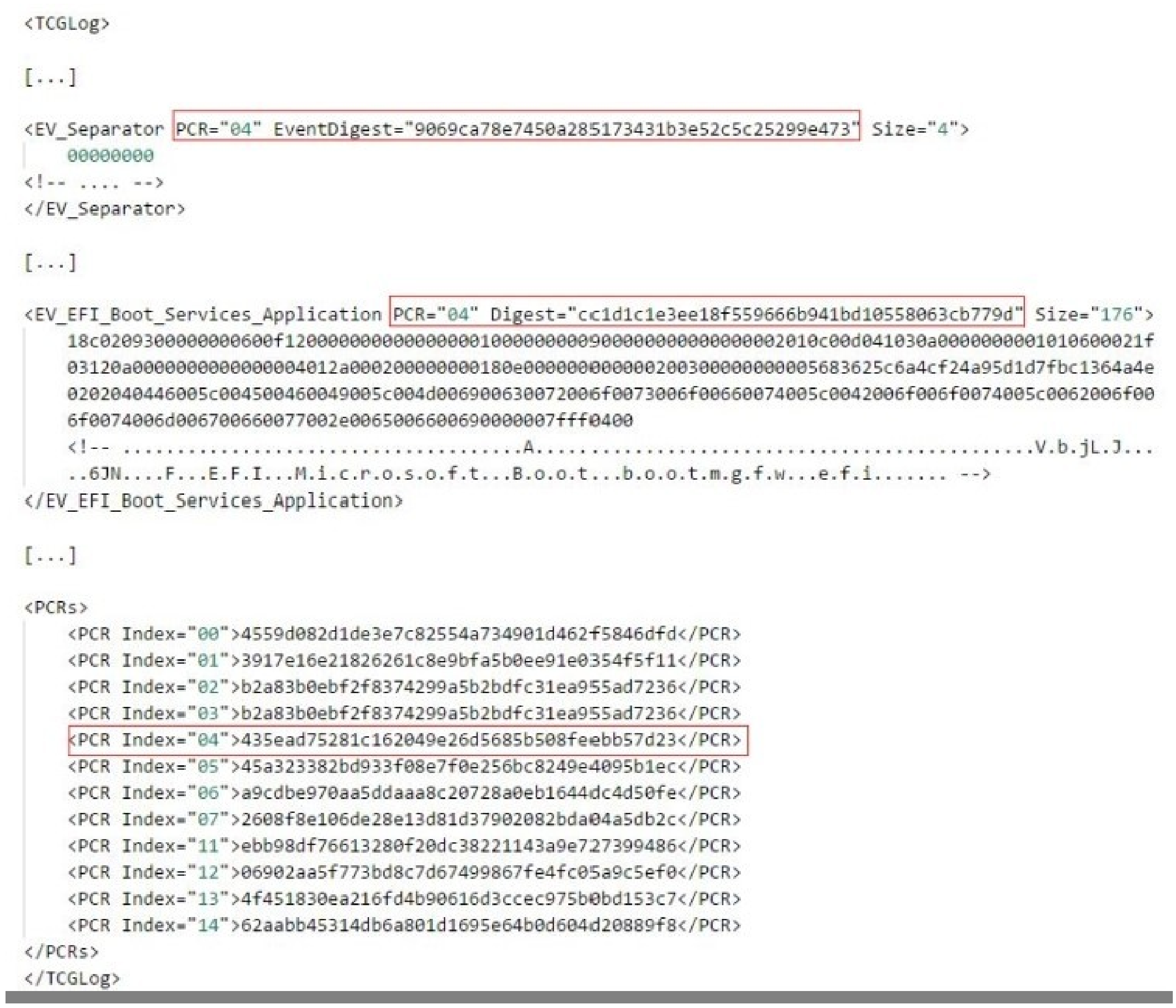
We analyzed the Windows Defender ELAM driver revoking the WBCL in a scenario where a given boot driver is considered malicious. We first configured the policy at `Computer Configuration` → `Administrative Templates` → `System` → `Early Launch Antimalware` such that the kernel initializes only known good images (see Section 2.3). We then set breakpoints at the functions for submitting and processing TPM commands implemented as part of the export TPM driver `tbs.sys` and the TPM driver `tpm.sys` (see Figure 2). Finally, we modified the return value of the `EbLookupProperty` function to `1` when the Windows Defender ELAM driver was checking a boot driver for malware. This return value indicates a known bad image. To remind, the return value of `EbLookupProperty` represents the decision of the ELAM driver on the maliciousness of a given boot driver.

We did not observe the Windows Defender ELAM driver or the kernel invoking `Tbsi_Revoke_Attestation` in order to revoke the current WBCL. They also did not invoke any other function of the TBS library or sent any TPM command to the TPM device after a decision on the maliciousness of the driver was made. It remains to be investigated whether the WBCL is revoked using means other than the ones we were focusing on, those specified in the Microsoft's development guidelines for ELAM drivers.[51] Although we did not observe the revocation of the WBCL, we observed that the kernel did not load the boot driver designated as a known bad image; that is, we observed that the Windows Defender ELAM driver effectively blocks the loading of malicious drivers.

The Windows kernel loads drivers and eventually the Windows subsystem, enabling the execution of system services and user applications (*Services and applications* in Figure 27, see Section 2.2). At this point, the content of WBCL files may be read by an application that transfers relevant content of these files to a remote entity verifying platform integrity (*Remote location* in Figure 27).[52] In Figure 27, we refer to the former as *attestation client* and to the latter as *attestation server*. The attestation client may obtain the most recent WBCL by issuing the `Tbsi_Get_TCG_Log` function of the TBS library.[53]

Windows Boot Configuration Log WBCL files contain data in binary form. This data can be translated into XML format using the `PCPTool` utility. Figure 28 depicts an excerpt of a WBCL file in XML format. This WBCL file was generated by the Windows 10 operating system after a regular system reboot.

49 https://msdn.microsoft.com/en-us/library/windows/desktop/jj553829(v=vs.85).aspx [Retrieved: 22/9/2017]
50 https://docs.microsoft.com/en-us/windows-hardware/drivers/install/elam-driver-requirements [Retrieved: 22/9/2017]
51 https://docs.microsoft.com/en-us/windows-hardware/drivers/install/elam-driver-requirements [Retrieved: 22/9/2017]
52 https://docs.microsoft.com/en-us/windows/device-security/protect-high-value-assets-by-controlling-the-health-of-windows-10-based-devices [Retrieved: 22/9/2017]
53 https://msdn.microsoft.com/de-de/library/windows/desktop/bb530712(v=vs.85).aspx [Retrieved: 22/9/2017]



```
<TCGLog>

[...]

<EV_Separator PCR="04" EventDigest="9069ca78e7450a285173431b3e52c5c25299e473" Size="4">
    00000000
<!-- .... -->
</EV_Separator>

[...]

<EV_EFI_Boot_Services_Application PCR="04" Digest="cc1d1c1e3ee18f559666b941bd10558063cb779d" Size="176">
    18c0209300000000600f120000000000000000010000000009000000000000000002010c00d041030a000000000101060002 1f
    03120a000000000000000004012a00020000000000180e000000000000002003000000000005683625c6a4cf24a95d1d7fbc1364a4e
    0202040446005c004500460049005c004d006900630072006f0073006f00660074005c0042006f006f0074005c0062006f00
    6f0074006d00670066077002e00650066006900000007fff0400
    <!-- ....................................A...........................................V.b.jL.J...
    ..6JN....F...E.F.I...M.i.c.r.o.s.o.f.t...B.o.o.t...b.o.o.t.m.g.f.w...e.f.i....... -->
</EV_EFI_Boot_Services_Application>

[...]

<PCRs>
    <PCR Index="00">4559d082d1de3e7c82554a734901d462f5846dfd</PCR>
    <PCR Index="01">3917e16e21826261c8e9bfa5b0ee91e0354f5f11</PCR>
    <PCR Index="02">b2a83b0ebf2f8374299a5b2bdfc31ea955ad7236</PCR>
    <PCR Index="03">b2a83b0ebf2f8374299a5b2bdfc31ea955ad7236</PCR>
    <PCR Index="04">435ead75281c162049e26d5685b508feebb57d23</PCR>
    <PCR Index="05">45a323382bd933f08e7f0e256bc8249e4095b1ec</PCR>
    <PCR Index="06">a9cdbe970aa5ddaaa8c20728a0eb1644dc4d50fe</PCR>
    <PCR Index="07">2608f8e106de28e13d81d37902082bda04a5db2c</PCR>
    <PCR Index="11">ebb98df76613280f20dc38221143a9e727399486</PCR>
    <PCR Index="12">06902aa5f773bd8c7d67499867fe4fc05a9c5ef0</PCR>
    <PCR Index="13">4f451830ea216fd4b90616d3ccec975b0bd153c7</PCR>
    <PCR Index="14">62aabb45314db6a801d1695e64b0d604d20889f8</PCR>
</PCRs>
</TCGLog>
```

Figure 28: An excerpt of a WBCL file

A WBCL file consists of multiple entries, where each entry contains relevant information on a given measured entity in the form of a `TCG_PCR_EVENT` structure. This structure is defined in the TCG (Extensible Firmware Interface) EFI Protocol Specification, family "2.0", level 00, revision 00.13 ([TCGEP 2016], Section 5), which is the latest TCG EFI specification at the time of writing. It represents each measurement of an entity as a single 'measurement event' in TCG terminology (see the XML tags starting with *EV_* in Figure 28). Some relevant fields of `TCG_PCR_EVENT` are `PCRIndex` and `Digest`. `PCRIndex` is the number of the PCR into which the entity has been measured, *(PCR Index* in Figure 28). `Digest` is the calculated hash of the measured entity (*Digest* in Figure 28).

Hashes of measured entities may be SHA-1 hashes or hashes of other types, referred to as 'crypto agile' hashes in the TCG EFI Protocol Specification ([TCGEP 2016], Section 5.2). They are extended into specific PCRs of the TPM (see the values of the *PCR* XML tags in Figure 28). Extension of a hash into a given PCR is done by updating the value already stored in the PCR as follows: $\mathtt{PCR}_{new} = \mathtt{H}\ (\mathtt{PCR}_{old} \,||\, \mathtt{Digest})$, where $\mathtt{PCR}_{old}$ is the old value stored in the PCR, $\mathtt{PCR}_{new}$ is the new value to be stored in the PCR, and `H` is a hash algorithm. In summary, the extension of a hash of a measured entity into a PCR consists of:

- concatenating the old value stored in the PCR with the hash of the entity;
- hashing the resulting value of the above operation; and

- storing the resulting hash into the PCR.

The extension of hashes into PCRs is described in detail in the Trusted Platform Module Library Part 3: Commands, family "2.0", level 00, revision 01.16 ([TCGLP3 2016], Section 22.2.1). Given that the values stored in the TPM's PCRs are hashes of measurement data, they serve primarily for verification of the integrity of WBCLs.

Into what PCRs hashes are extended depends on what is measured. Table 1 of the TCG PC Client Platform Firmware Profile Specification, family "2.0", level 00, revision 00.21[TCGF 2016] presents a mapping between measured entities and PCR indexes. In summary, the PCRs with indexes between 0 and 7 are used when extending hashes of firmware-related entities.Example such entities are UEFI variables and sections of the secure database ([ERNW WP2], Section 3.5). PCRs with indexes between 8 and 15 are used for measuring entities related to the installed operating system. What is stored in these PCRs is left to the discretion of the operating system's vendor. The PCRs with indexes between 0 and 15 are non-resettable PCRs, that is, the values stored in them cannot be cleared by the operating system, but only by hardware at each system reboot (*non-resettable PCRs* in Figure 27, [TCGPCI 2013], Section 5.3).[54]

Each measurement event is of a specific type, which indicates what has been measured. For example, the measurement event of type `EV_EFI_VARIABLE_BOOT` contains measurement of a UEFI variable ([TCGF 2016], Table 5). Table 2 presents a mapping between PCR indexes and types of measurement events. The events were extended into the PCRs on the Windows 10 system after a regular system reboot. Table 2 lists only events specified in the TCG PC Client Platform Firmware Profile Specification. We obtained the results presented in Table 2 using a parser of WBCL files translated into XML format, which we developed. The code of the parser is placed in the Appendix, section 'WBCL parser'.

The section 'Event Types' of Table 2 presents only brief descriptions of the event types listed in the table. Some event types have multiple sub-types storing comprehensive information on measured entities. We refer to ([TCGF 2016], Section 9.3.1) for detailed descriptions of event types.

PCR	Event Type
0	`EV_CRTM_Contents`; `EV_CRTM_Version`; `EV_Post_Code`; `EV_EFI_Handoff_Tables`; `EV_Separator`
1	`EV_Event_Tag`; `EV_Event_Tag`; `EV_EFI_Handoff_Tables`; `EV_Separator`; `EV_EFI_Variable_Boot`
2	`EV_Separator`
3	`EV_Separator`
4	`EV_Separator`; `EV_EFI_Boot_Services_Application`
5	`EV_EFI_Action`; `EV_Separator`
6	`EV_Action`; `EV_Separator`
7	`EV_EFI_Variable_Driver_Config`; `EV_Separator`
11	`EV_Compact_Hash`
12	`EV_Event_Tag`; `EV_Separator`
13	`EV_Event_Tag`; `EV_Separator`
Event Types	
`EV_CRTM_Contents`: Measurement of SRTM code (see Section 1.3)	

54 ([Thom 2016], Section 'Root of Trust Overview') presents a mapping between measured Windows 8 entities and PCR indexes. To the best of our knowledge, such a mapping for Windows 10 entities is not available at the time of writing. Based on our analysis of the content of WBCL files, we assume that the mapping between Windows 10 entities and PCR indexes is to a great extent the same as that specified in [Thom 2016].

`EV_CRTM_Version`: Measurement of a string specifying the version of the SRTM

`EV_Post_Code`: Measurement of firmware code or data (e.g., power-on self-test (POST) firmware code, or ACPI-related data)

`EV_EFI_Handoff_Tables`: Measurement of system configuration tables (UEFI variables)

`EV_Separator`: A generic event typically used for indicating the end of contents (i.e., capping) of a given PCR

`EV_Event_Tag`: Typically measurement of read-only memory (ROM) code or arbitrary application code (this is a deprecated event type according to the latest TCG PC Client Platform Firmware Profile Specification [TCGF 2016])

`EV_EFI_Variable_Boot`: Measurement of UEFI variables relevant to the platform's booting process (e.g., a variable storing the device boot order)

`EV_EFI_Boot_Services_Application`: Measurement of code loading from the boot device (typically code sequences of the boot manager, see Section 2.2.1)

`EV_EFI_Action`: Measurement of strings specifying executed UEFI actions (e.g., "Calling UEFI Application from Boot Option", see ([TCGF 2016], Table 7).

`EV_EFI_Variable_Driver_Config`: Measurements of properties of UEFI variables (e.g., GUID, see [ERNW WP4])

`EV_Compact_Hash`: Measurement of arbitrary code or data, typically used by the boot manager (e.g., measurement of the code sequences of the Windows loader)

Table 2: A mapping between PCR indexes and types of measurement events

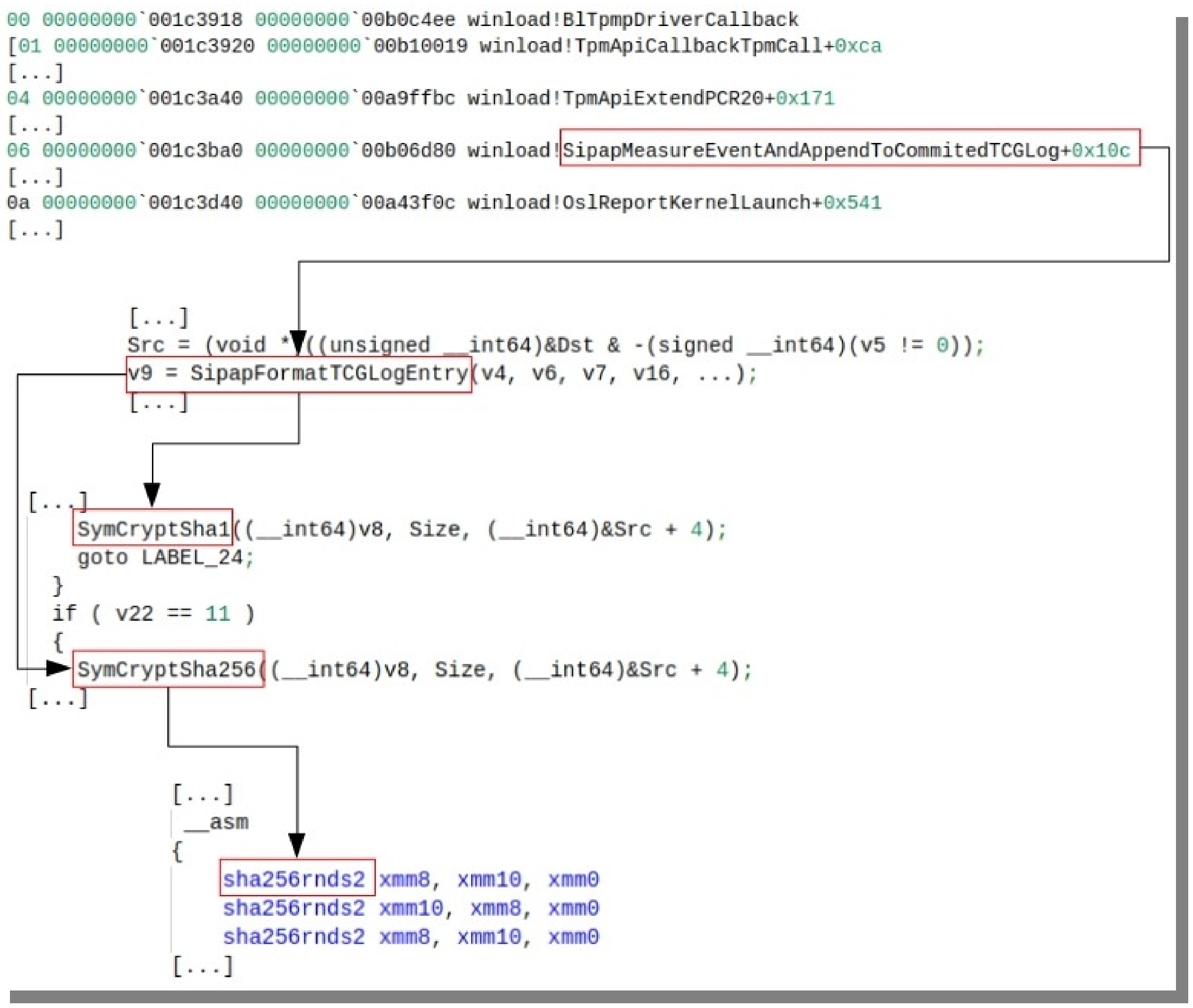
In addition to the data presented in Table 2, we extracted from the WBCL file a list of measured executables. This includes system drivers, system services, and driver executables. Measurements of executables are stored in WBCL files as events of type `EV_Event_Tag`. This event type contains information on the Authenticode hashes of the executables (see Section 2.2.3). Measurements of executables are extended into the PCRs with indexes 12 and 13 ([Thom 2016], Section 'Windows Integrity Measurements'). The list of executables we extracted is placed in the Appendix, section 'Measured Executables'. It contains paths to, and filenames of, measured executables. The hard disc volume is omitted in the paths presented in the Appendix.

Implementation of integrity measurement Measurements of Windows entities are performed by the boot manager (see Section 2.2.1) and the Windows loader (see Section 2.2.2, *Pre-OS* in Figure 27). In this paragraph, we focus on the implementation of the integrity measurement mechanism in the Windows loader. We observed that the implementation of this mechanism in the boot manager is conceptually identical to the one presented in this paragraph.

Figure 29 depicts the stack and code snippets of functions executed as part of the integrity measurement mechanism implemented in the Windows loader. The content depicted in Figure 29 is as displayed by the `windbg` debugger and by the `IDA` disassembler in the form of pseudo-code. The `OslReportKernelLaunch` function is one of the functions implemented in the Windows loader that triggers integrity measurement. This is done by queueing measurement events for processing by submitting them to the `SipapMeasureEventAndAppendToCommitedTCGLog` function. This indicates that integrity measurements are conducted in an asynchronous manner.

`SipapMeasureEventAndAppendToCommitedTCGLog` first calculates hashes and therefore conducts the actual measurements. It then extends the measurements into PCRs by invoking `TpmApiExtendPCR`. This function is part of the TpmApi of the Windows loader (see Section 2.2.2, paragraph 'TPM usage').`TpmApiExtendPCR` constructs a TPM command buffer and invokes `TpmApiCallbackTpmCall`. This function communicates with the TPM by invoking `BlTpmpDriverCallback`.

Operations for hash calculation are implemented as functions of the Windows loader, that is, they are software-implemented (see *SipapFormatTCGLogEntry*, *SymCryptSha1/256* in Figure 29). For example, if the Intel SHA extensions are present [Gulley 2013], hash calculation is performed by executing CPU instructions specifically developed for that purpose (*sha256rnds2* in Figure 29).



```
00 00000000`001c3918 00000000`00b0c4ee winload!BlTpmpDriverCallback
[01 00000000`001c3920 00000000`00b10019 winload!TpmApiCallbackTpmCall+0xca
[...]
04 00000000`001c3a40 00000000`00a9ffbc winload!TpmApiExtendPCR20+0x171
[...]
06 00000000`001c3ba0 00000000`00b06d80 winload!SipapMeasureEventAndAppendToCommitedTCGLog+0x10c
[...]
0a 00000000`001c3d40 00000000`00a43f0c winload!OslReportKernelLaunch+0x541
[...]
```

```
[...]
Src = (void *)((unsigned __int64)&Dst & -(signed __int64)(v5 != 0));
v9 = SipapFormatTCGLogEntry(v4, v6, v7, v16, ...);
[...]
```

```
[...]
    SymCryptSha1((__int64)v8, Size, (__int64)&Src + 4);
    goto LABEL_24;
  }
  if ( v22 == 11 )
  {
    SymCryptSha256((__int64)v8, Size, (__int64)&Src + 4);
[...]
```

```
[...]
__asm
{
    sha256rnds2 xmm8, xmm10, xmm0
    sha256rnds2 xmm10, xmm8, xmm0
    sha256rnds2 xmm8, xmm10, xmm0
[...]
```

Figure 29: Integrity measurement in the Windows loader

2.5 TPM Provisioning

In this section, we describe the implementation of the TPM provisioning process in Windows 10. We first define the term TPM provisioning as there is no clear definition of it in the TPM 2.0 Library Specification. Under TPM provisioning, we understand activities storing data in the TPM device, where the stored data is a requirement for the TPM device to be used. This includes: authorization values, the EK, and the SRK (see Section 1.3), which are also in the main focus of this section.

The TPM 2.0 Library Specification Part 1: Architecture, level 00, revision 01.38 ([TCGLP1 2016], Section 13.8.1) refers to the process of storing the owner, endorsement, and lockout authorization values in the TPM as 'taking ownership' of the TPM. When analyzing the implementation of this process in Windows 10, we observed that there is a discrepancy in the nomenclature used in the TPM 2.0 Library Specification and in the context of Windows. Windows uses an authorization value, named `OwnerAuth`, for managing the TPM (e.g., access control over TPM commands) and unlocking the TPM if it is in a locked state. This is the use of both the owner and lockout authorization values as specified in the TPM 2.0 Library Specification. In the context of Windows, `OwnerAuth` represents the core authorization value for managing the TPM, required for most TPM management activities.

In addition to `OwnerAuth`, Windows uses the authorization values `EndorsementAuth` and `StorageOwnerAuth`. We observed that `EndorsementAuth` is used in Windows context same as specified in the TPM 2.0 Library Specification. For example, `EndorsementAuth` is used when a new EK is generated. To the contrary, `StorageOwnerAuth` is not defined as an authorization value in this specification. In Windows context, `StorageOwnerAuth` is used for storing keys in the TPM's storage hierarchy (see Figure 1).

We observed that the values of `EndorsementAuth` and `StorageOwnerAuth` are zeroed-out by default.[55] Therefore, the generation and storage of `EndorsementAuth` and `StorageOwnerAuth` is not in the focus of this section.

Figure 30 depicts the workflow of the TPM provisioning process in scenarios where the TPM is manually and auto-provisioned (see Section 1.3). In Figure 30, numbers encapsulated in boxes with dashed lines mark activities that are part of the manual TPM provisioning process in the order they occur. Numbers encapsulated in boxes with full lines mark activities that are part of the TPM auto-provisioning process in the order they occur. In Section 2.5.1 and Section 2.5.2, we discuss the workflow of the TPM manual and auto-provisioning process, respectively. In this work, we refer to `OwnerAuth` as the 'owner authorization value'.

2.5.1 Manual Provisioning

The TPM is provisioned manually by a user triggering the provisioning process after the TPM has been cleared. Clearing the TPM is a process involving the deletion of authorization values and the SRK stored in the TPM's memory [TCGLP3 2016], Section 24.6). In Windows, the TPM may be cleared, for example, using the `Clear-Tpm` PowerShell cmdlet (see Section 3.1.2, paragraph 'PowerShell').

The TPM provisioning process can be triggered manually by using the TPM management utility (executable: `tpm.msc`), the TPM initialization wizard (executable: `tpminit.exe`), or by executing the `Initialize-Tpm` PowerShell cmdlet. Alternatively, a user may invoke the `Provision` function of the `Win32_Tpm` class. This class is part of the TPM's WMI interface. We discuss the TPM management utility, the PowerShell cmdlets for managing the TPM, and the `Win32_Tpm` class in Section 3.1. In Figure 30, we refer to the TPM management utility, the TPM initialization wizard, PowerShell, and any user application instantiating the `Win32_Tpm` class as *TPM management applications*.

We triggered the TPM provisioning process manually using all TPM management applications mentioned above. We observed that all of them invoke the `CtpmCoreClass::Provision` function implemented in the `%SystemRoot%\System32\TpmCoreProvisioning.dll` library file (*1* in Figure 30). This function first invokes `TpmApiGetRandom` (2 in Figure 30). `TpmApiGetRandom` generates a new random owner authorization value, which we verified as discussed next.

55 The values of `EndorsementAuth` and `StorageOwnerAuth` are stored in the system's registry at the keys `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\TPM\WMI\Endorsement\EndorsementAuth` and `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\TPM\WMI\Endorsement\StorageOwnerAuth`, respectively.

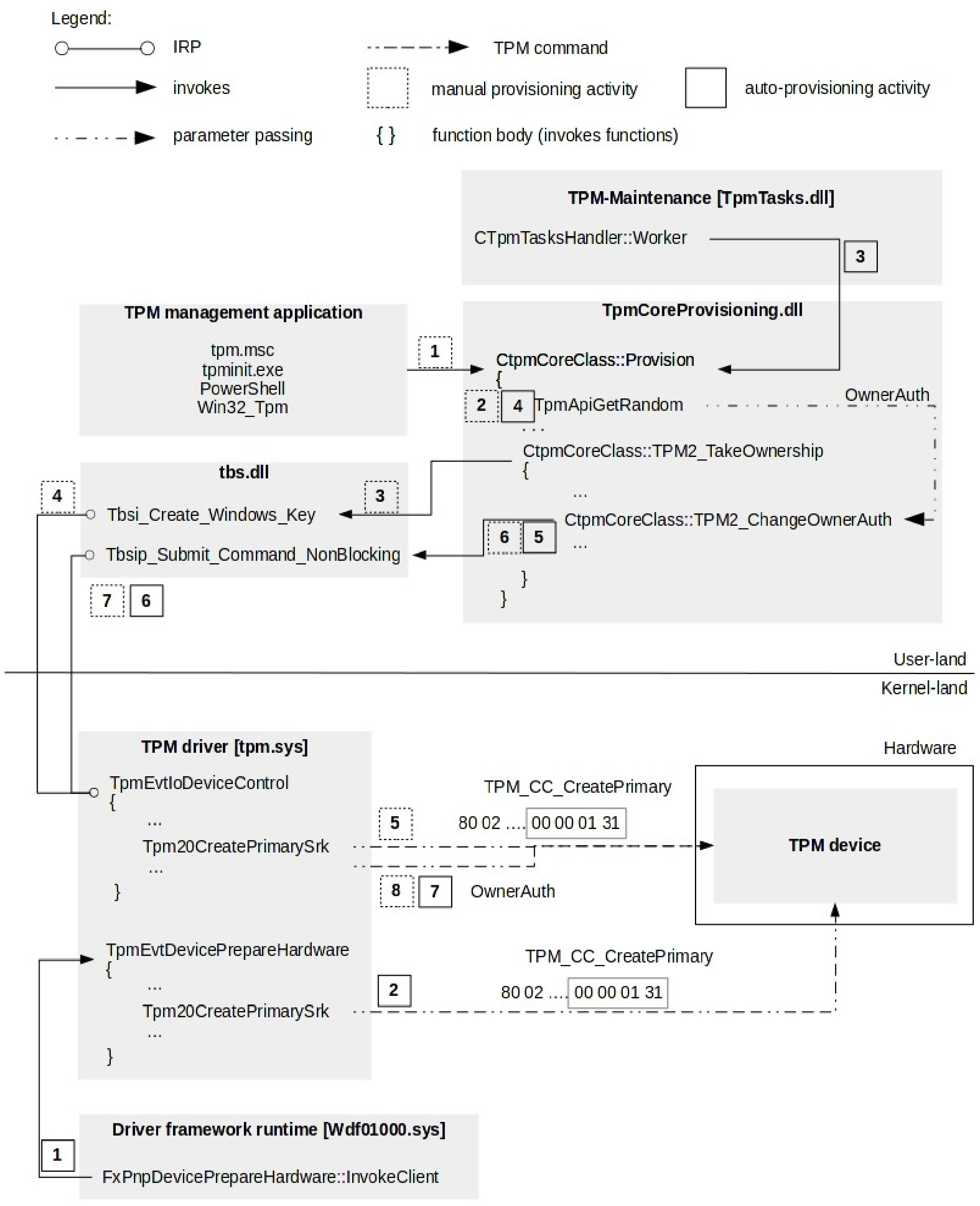
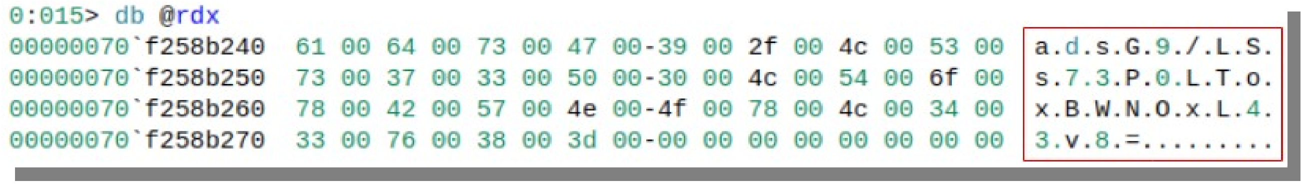


Figure 30: The TPM provisioning process

We first enabled storing of the owner authorization value at the registry key `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\TPM\WMI\Admin\OwnerAuthFull` (see Section 3.1.2, paragraph 'Group Policy').[56] The owner authorization value is generated as part of the TPM provisioning process. We then triggered the TPM provisioning process and analyzed the execution of `CtpmCoreClass::Provision`.

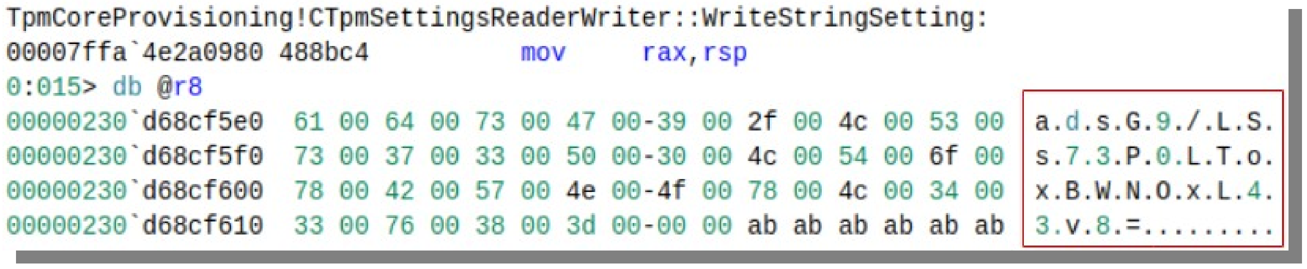
56 https://blogs.technet.microsoft.com/dubaisec/2017/02/28/tpm-owner-password/ [Retrieved: 22/9/2017]

The random value generated by `TpmApiGetRandom` is encoded using the Base64 algorithm[57] and passed as the second parameter of `CtpmCoreClass::TPM2_TakeOwnership` (see Figure 30). Figure 31 depicts the value of this parameter. The Base64-encoded random value is then passed as the third parameter of `CtpmSettingsReaderWriter::WriteStringSetting` (not depicted in Figure 30). Figure 32 depicts the value of this parameter.



```
0:015> db @rdx
00000070`f258b240  61 00 64 00 73 00 47 00-39 00 2f 00 4c 00 53 00  a.d.s.G.9./.L.S.
00000070`f258b250  73 00 37 00 33 00 50 00-30 00 4c 00 54 00 6f 00  s.7.3.P.0.L.T.o.
00000070`f258b260  78 00 42 00 57 00 4e 00-4f 00 78 00 4c 00 34 00  x.B.W.N.O.x.L.4.
00000070`f258b270  33 00 76 00 38 00 3d 00-00 00 00 00 00 00 00 00  3.v.8.=.........
```

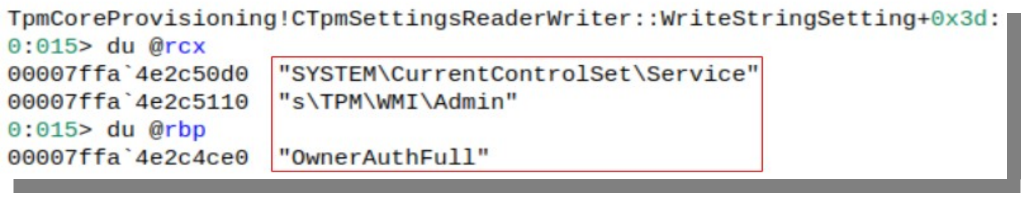
Figure 31: A random value generated by TpmApiGetRandom (Base64-encoded)



```
TpmCoreProvisioning!CTpmSettingsReaderWriter::WriteStringSetting:
00007ffa`4e2a0980 488bc4          mov     rax,rsp
0:015> db @r8
00000230`d68cf5e0  61 00 64 00 73 00 47 00-39 00 2f 00 4c 00 53 00  a.d.s.G.9./.L.S.
00000230`d68cf5f0  73 00 37 00 33 00 50 00-30 00 4c 00 54 00 6f 00  s.7.3.P.0.L.T.o.
00000230`d68cf600  78 00 42 00 57 00 4e 00-4f 00 78 00 4c 00 34 00  x.B.W.N.O.x.L.4.
00000230`d68cf610  33 00 76 00 38 00 3d 00-00 00 ab ab ab ab ab ab  3.v.8.=.........
```

Figure 32: The third parameter of CtpmSettingsReaderWriter::WriteStringSetting

`CtpmSettingsReaderWriter::WriteStringSetting` stores the value of its third parameter in the registry, at the registry key specified in its first and fifth parameter. We observed that `CtpmSettingsReaderWriter::WriteStringSetting` writes the Base64-encoded random value generated by `TpmApiGetRandom` in the registry at the key `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\TPM\WMI\Admin\OwnerAuthFull` (see Figure 33). This shows that the value generated by `TpmApiGetRandom` is the new owner authorization value.

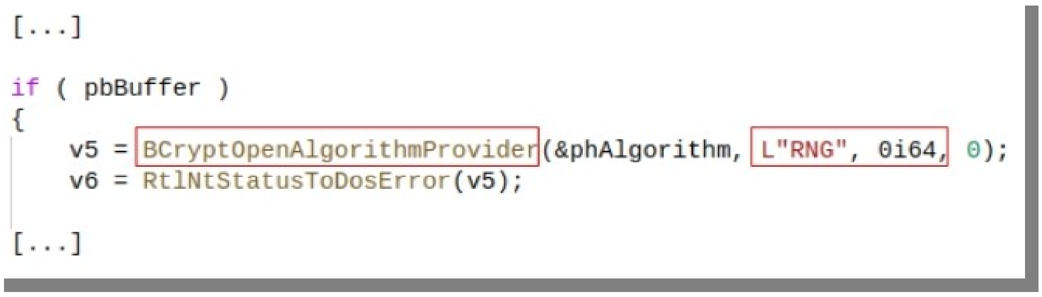


```
TpmCoreProvisioning!CTpmSettingsReaderWriter::WriteStringSetting+0x3d:
0:015> du @rcx
00007ffa`4e2c50d0  "SYSTEM\CurrentControlSet\Service"
00007ffa`4e2c5110  "s\TPM\WMI\Admin"
0:015> du @rbp
00007ffa`4e2c4ce0  "OwnerAuthFull"
```

Figure 33: The first and fifth parameter of CtpmSettingsReaderWriter::WriteStringSetting

`TpmApiGetRandom` generates random values using a software-implemented provider of the CNG library (see Section 2.1.1). `TpmApiGetRandom` uses the `BcryptOpenAlgorithmProvider` to load the default provider for the CNG algorithm `RNG`.[58] This is indicated by the `NULL` value of the second, and the `RNG` value of the third, parameter of `BcryptOpenAlgorithmProvider`.[59] In Figure 34, we depict these values as pseudo-code generated by the `IDA` disassembler. We developed a simple application invoking `BcryptOpenAlgorithmProvider` with the same parameters as those depicted in Figure 34. We observed that a software-implemented CNG provider was loaded.

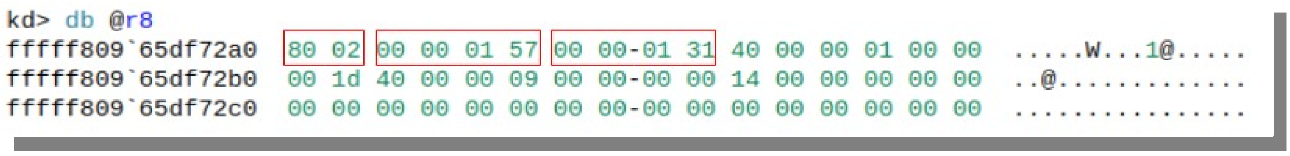
57 https://tools.ietf.org/html/rfc4648 [Retrieved: 22/9/2017]
58 https://msdn.microsoft.com/de-de/library/windows/desktop/aa375534(v=vs.85).aspx [Retrieved: 22/9/2017]
59 See the documentation of `BcryptOpenAlgorithmProvider` for descriptions of its second and third parameter: https://msdn.microsoft.com/de-de/library/windows/desktop/aa375479(v=vs.85).aspx [Retrieved: 22/9/2017]



```
[...]

if ( pbBuffer )
{
    v5 = BCryptOpenAlgorithmProvider(&phAlgorithm, L"RNG", 0i64, 0);
    v6 = RtlNtStatusToDosError(v5);

[...]
```

Figure 34: BCryptOpenAlgorithmProvider in TpmApiGetRandom

After `TpmApiGetRandom` generates the new owner authorization value, `CtpmCoreClass::TPM2_TakeOwnership` is executed. This function first triggers the generation of a new SRK by invoking `Tbsi_Create_Windows_Key`, implemented in `tbs.dll` (*3* in Figure 30). This function issues an IRP containing a TPM command to the TPM driver `tpm.sys` (*4* in Figure 30). The driver handles incoming IRPs in its function `TpmEvtIoDeviceControl`. It handles an IRP containing request for generation of a new SRK by invoking `Tpm20CreatePrimarySrk`. This function constructs a TPM command byte sequence and submits it to the TPM (*5* in Figure 30). Figure 35 depicts a part of this sequence.



```
kd> db @r8
fffff809`65df72a0  80 02 00 00 01 57 00 00-01 31 40 00 00 01 00 00  .....W...1@.....
fffff809`65df72b0  00 1d 40 00 00 09 00 00-00 00 14 00 00 00 00 00  ..@.............
fffff809`65df72c0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

Figure 35: A TPM command sequence for generating an SRK

The sub-sequence `80 02` is a constant value defined as `TPM_ST_SESSIONS` in the Trusted Platform Module Library Part 2: Structures, family 2.0, level 00, revision 01.38 ([TCGLP2 2016], Table 19). The sub-sequence `00 00 01 57` is the size of the TPM command, defined as an integer of a fixed length of 4 bytes ([TCGLP3 2016], Section 24.1.1). The sub-sequence `00 00 01 31` is the TPM command code `TPM_CC_CreatePrimary`, which is defined in the Trusted Platform Module Library Part 2: Structures, family 2.0, level 00, revision 01.38 ([TCGLP2 2016], Section 6.5.2). As specified in the Trusted Platform Module Library Part 3: Commands, family 2.0, level 00, revision 01.38 ([TCGLP3 2016], Section 24.1.1), this command code is a unique identifier of the TPM command `TPM2_CreatePrimary`. This command is used for creating an SRK or an `EK. An` SRK is generated in TPM-context and its private part never leaves the TPM ([TCGLP3 2016], Section 24.1.10). After it is generated, the public part of the SRK is stored in the system's registry at the key `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\TPM\WMI\Admin\SRKPub`.

Through static analysis, we observed that an EK is generated in a conceptually identical manner as an SRK. This involves invoking the function `Tpm20CreatePrimaryEk` implemented in the TPM driver and executing the `TPM_CC_CreatePrimary` command (not depicted in Figure 30). We were unable to dynamically observe an actual generation of an EK. This is because the TPM installed on the platform we worked on was already provisioned with an EK at manufacture time. Once stored in the TPM's memory, an EK cannot be removed by clearing the TPM and a new EK cannot be stored in it ([TCGLP3 2016], Section 24.6).

After an SRK is created, the Base64-encoded random value generated by `TpmApiGetRandom` is passed to `CtpmCoreClass::TPM2_TakeOwnership` and written to the TPM by the function issuing multiple IRPs to the `tpm.sys` driver. This value is the new owner authorization value (*OwnerAuth* in Figure 30). The IRPs issued by `CtpmCoreClass::TPM2_TakeOwnership` contain TPM commands and the new owner authorization value.

`CtpmCoreClass::TPM2_TakeOwnership` issues the IRPs by invoking `Tbsip_Submit_Command_NonBlocking`, implemented in `tbs.dll` (*6* in Figure 30).

`Tbsip_Submit_Command_NonBlocking` is a variant of `Tbsip_Submit_Command`. It submits IRPs to the TPM driver as described in Section 2.1.1, paragraph 'Direct TPM communication' (*7* in Figure 30). The driver handles the IRPs issued by `CtpmCoreClass::TPM2_TakeOwnership` in `TpmEvtIoDeviceControl` and submits the TPM commands to the TPM device (*8* in Figure 30) for storing the new owner authorization value.

2.5.2 Auto-provisioning

The TPM auto-provisioning process is similar to the manual, with the major difference of how the tasks of generating a new SRK and an owner authorization value are triggered. We cleared the TPM by executing the `Clear-Tpm` PowerShell cmdlet (see Section 3.1.2, paragraph 'PowerShell') and enabled TPM auto-provisioning by executing the `Enable-TpmAutoProvisioning` cmdlet. We also enabled storing of the owner authorization value at the registry key `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\TPM\WMI\Admin\OwnerAuthFull` (see Section 2.5.1). We then restarted the operating system.

We note that a requirement for the TPM auto-provisioning process is the `NoPPIProvision` flag to be set to true ([TCGPP 2015], Section 8.1.8). This flag is stored in the firmware and its value is evaluated by Windows if the TPM is deactivated, or disabled.

We observed that the function `Tpm20CreatePrimarySrk` is executed in kernel context during system booting. This function triggers the generation of a new SRK (see Section 2.5.1). `Tpm20CreatePrimarySrk` is invoked within a thread created by the `TpmEvtDevicePrepareHardware` function of the TPM driver `tpm.sys`.

`TpmEvtDevicePrepareHardware` is invoked as part of the initialization procedures performed by the Windows kernel; that is, it is invoked by `FxPnpDevicePrepareHardware::InvokeClient`. This function is implemented in the Driver Framework Runtime driver `Wdf01000.sys` (*1* in Figure 30). This driver is part of the Windows Driver Frameworks platform and acts as a filter driver for the TPM driver `tpm.sys`.[60] Filter drivers are drivers that extent the functionalities of other drivers and are part of their driver stacks when specific requests need to be handled.[61] For example, filter drivers perform initialization tasks.

As described in Section 2.5.1, `Tpm20CreatePrimarySrk` triggers the generation of a new SRK by issuing the `TPM2_CreatePrimary` TPM command. This command is uniquely identified by the command code `TPM_CC_CreatePrimary` (2 in Figure 30).

As mentioned in 2.5.1, we were unable to observe a generation of an EK. This is because the TPM installed on the platform we worked on was already provisioned with an EK at manufacture time. However, through static code analysis, we observed that an EK is generated in a conceptually identical manner as an SRK.

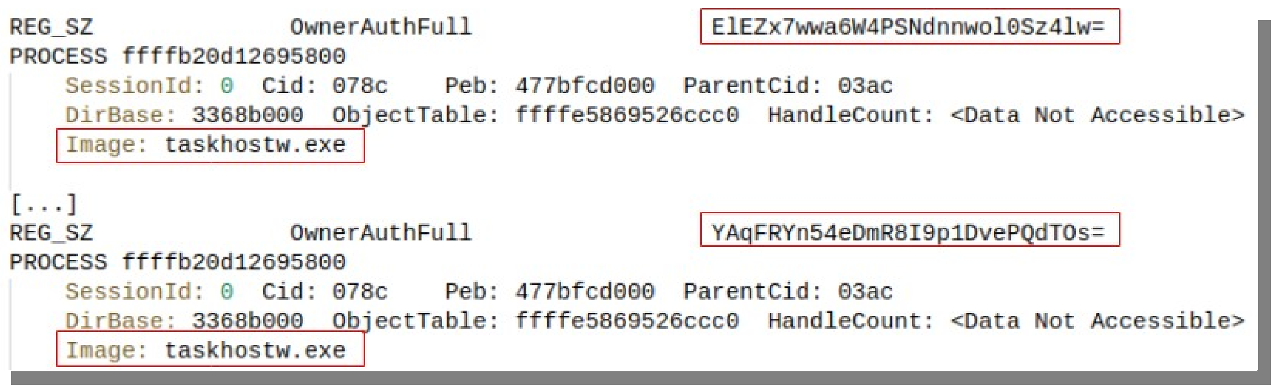
When analyzing the manual TPM provisioning process, we observed that the new owner authorization value is generated in user-land and passed to the TPM driver `tpm.sys` in the form of an IRP. Based on this observation, we identifed the context in which a new owner authorization value is generated as part of the TPM auto-provisioning process. We achieved this by monitoring the value of the `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\TPM\WMI\Admin\OwnerAuthFull` registry key each time an IRP is handled by the TPM driver. To this end, we executed the `windbg` command `bp tpm!TpmEvtIoDeviceControl "!reg querykey \\REGISTRY\\MACHINE\\SYSTEM\\CurrentControlSet\\Services\\TPM\\WMI\\Admin\\; !process -1 0; g"`. This command sets a breakpoint at the function of the TPM driver handling incoming IRPs – `TpmEvtIoDeviceControl` (see Section 2.5.1). It also displays the value of the `OwnerAuthFull` registry key, as well as information on

60 https://msdn.microsoft.com/en-us/library/windows/hardware/ff557565(v=vs.85).aspx [Retrieved: 22/9/2017]
61 https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/filter-drivers [Retrieved: 22/9/2017]

the user process issuing an IRP (if any). Any change in the value of this key indicates the IRP passing a new owner authorization value to the TPM.

We emphasize that the owner authorization value being generated in user-land is not a security flaw. To remind, this value serves as a user password protecting TPM functionalities. Therefore, its presence in user-land is a functional requirement. For example, users are able to specify a new, or modify an existing, owner authorization value (see Section 3.1).

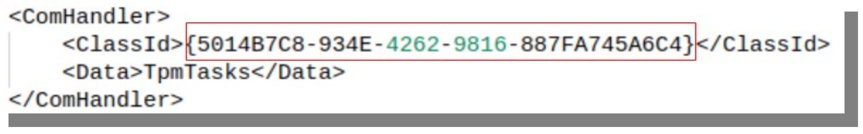
We identified the user process named 'Host Process for Windows Tasks' (executable: `taskhostw.exe`) as the processing issuing the IRP that passes a new owner authorization value to the TPM. Figure 36 depicts the change in the value of the `OwnerAuthFull` registry key identifying `taskhostw.exe`. In Figure 36, the value starting with `ElE` is the old owner authorization value, the value starting with `YAq` is the new owner authorization value, and the `Image` field contains the name of the executable issuing the IRP that passes the new owner authorization value to the TPM.



```
REG_SZ             OwnerAuthFull                ElEZx7wwa6W4PSNdnnwol0Sz4lw=
PROCESS ffffb20d12695800
    SessionId: 0  Cid: 078c    Peb: 477bfcd000  ParentCid: 03ac
    DirBase: 3368b000  ObjectTable: ffffe5869526ccc0  HandleCount: <Data Not Accessible>
    Image: taskhostw.exe

[...]
REG_SZ             OwnerAuthFull                YAqFRYn54eDmR8I9p1DvePQdTOs=
PROCESS ffffb20d12695800
    SessionId: 0  Cid: 078c    Peb: 477bfcd000  ParentCid: 03ac
    DirBase: 3368b000  ObjectTable: ffffe5869526ccc0  HandleCount: <Data Not Accessible>
    Image: taskhostw.exe
```

Figure 36: taskhostw.exe changing the OwnerAuthFull registry key

The `taskhostw.exe` executable executes scheduled tasks. Using the `Task Scheduler` utility (executable: `taskschd.msc`), we discovered the task named `Tpm-Maintenance`. This task is configured to execute at every system startup. We exported information about `Tpm-Maintenance` into an XML format using the `Task Scheduler`. We observed that the task executes functions of a component object model (COM) object that is an instance of the class with an ID `5014B7C8-934E-4262-9816-887FA745A6C4`. Figure 37 depicts the snippet of the XML file containing information about `Tpm-Maintenance` (i.e., about the ID of the instantiated COM class, *ClassId* in Figure 37).



```
<ComHandler>
    <ClassId>{5014B7C8-934E-4262-9816-887FA745A6C4}</ClassId>
    <Data>TpmTasks</Data>
</ComHandler>
```

Figure 37: The TPM-Maintenance task

By exploring the contents of the registry key `HKEY_CLASESS_ROOT/CLSID/{5014B7C8-934E-4262-9816-887FA745A6C4}/InprocServer32/`, we observed that the COM class with an ID `5014B7C8-934E-4262-9816-887FA745A6C4` is implemented in the `%SystemRoot%\System32\TpmTasks.dll` library file. We analyzed the implementation of this file using the `IDA` disassembler observing that at every system startup, it creates a thread executing the function `CtpmTasksHandler::Worker`. This function invokes `CtpmCoreClass::Provision`, implemented in the `TpmCoreProvisioning.dll` library file (*3* in Figure 30).

As described in Section 2.5.1, the generation of a new owner authorization value takes places in `CtpmCoreClass::Provision`. The TPM auto-provisioning process continues as follows: a new owner authorization value is generated by `TpmApiGetRandom` (*4* in Figure 30); this value is submitted to the TPM driver `tpm.sys` (5 and 6 in Figure 30); and the TPM driver stores the new owner authorization value in the TPM device (7 in Figure 30).

In addition to those mentioned above, the `Tpm-Maintenance` task performs other activities. For example, if an AIK named `Windows AIK` is not present, it triggers the generation of a new AIK by invoking `CwindowsAIK::CreateWindowsAIK` (not depicted in Figure 30). An AIK is generated in scenarios where the TPM is manually or automatically provisioned.

`CwindowsAIK::CreateWindowsAIK` is implemented in `TbsCoreProvisioning.dll`. The public key of the new AIK is written to the system's registry at `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\TPM\WMI\WindowsAIKPub`. Figure 38 depicts the generation of an AIK in the form of a pseudo-code generated by the `IDA` disassembler. `CwindowsAIK::CreateWindowsAIK` uses the CNG library to load the `Platform Cryptographic Provider` (*NcryptOpenStorageProvider* and *Microsoft Platform Crypto Provider* in Figure 38)[62] and to generate the AIK named `Windows AIK` using the TPM (*NCryptCreatePersistedKey* and *Windows AIK* in Figure 38)[63]. This indicates that the AIK is generated in TPM context. The presence of an AIK is not a requirement for the TPM device to be used. Therefore, the detailed analysis of the AIK generation process is out of the scope of this work package.

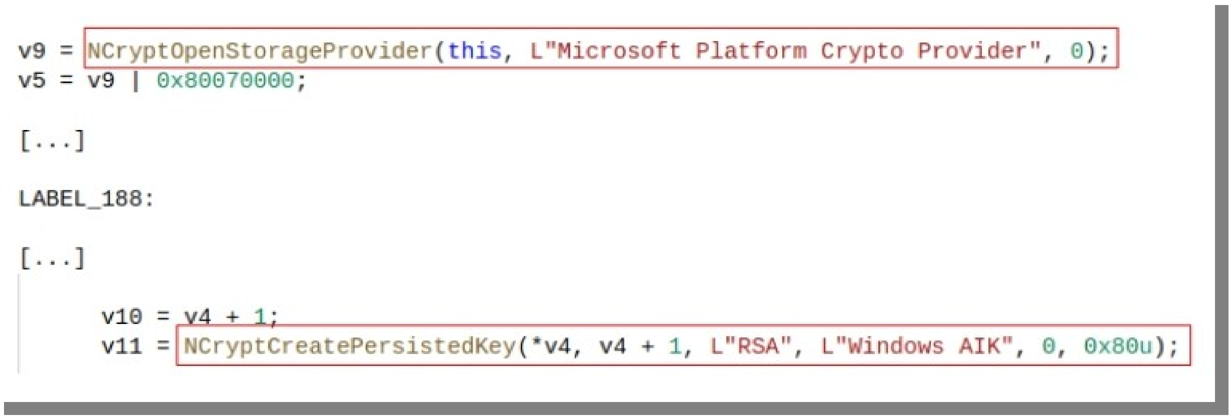

```
v9 = NCryptOpenStorageProvider(this, L"Microsoft Platform Crypto Provider", 0);
v5 = v9 | 0x80070000;

[...]

LABEL_188:

[...]

      v10 = v4 + 1;
      v11 = NCryptCreatePersistedKey(*v4, v4 + 1, L"RSA", L"Windows AIK", 0, 0x80u);
```

Figure 38: Generation of an AIK

2.6 Security Aspects

The analysis presented in Section 2.1 – Section 2.5 was performed by taking security aspects, such as potential attack scenarios, threats, and implemented mitigations, into account. We considered the following attack scenarios:

- Unauthorized modification of the roots of trust for integrity verification: This attack may allow an attacker to compromise the system's integrity verification procedure and enable the loading of malicious executables;
- Unauthorized configuration or provisioning of the TPM: This attack may allow, for example, an attacker to modify the existing TPM owner authorization value and execute restricted TPM commands;
- Unauthorized disabling of image integrity verification: This attack may allow an attacker to modify relevant system executables or load malicious executables;
- Unauthorized modification of integrity measurements: This attack involves modifying the WBCL or measurements in the database of the ELAM driver. It may allow an attacker to modify measured executables without being detected by a remote attestation mechanism or the ELAM driver;

62 https://msdn.microsoft.com/de-de/library/windows/desktop/aa376286(v=vs.85).aspx [Retrieved: 22/9/2017]
63 https://msdn.microsoft.com/de-de/library/windows/desktop/aa376247(v=vs.85).aspx [Retrieved: 22/9/2017]

- Exploitation of an implementation or design flaw: This attack may result in the execution of any unauthorized activity, which includes those mentioned in this section. For example, an attacker may exploit a flaw in the image integrity verification procedure of Windows 10 in order to load a malicious image.

Table 3 provides an ovierview of the considered attack scenarios (column 'Attack scenario'), implemented mitigations for these scenarios (column 'Mitigation'), and references to sections of this work where the mitigations are discussed (column 'Reference').

Attack scenario	Mitigation	Reference
Unauthorized modification of the roots of trust for integrity verification	The roots of trust for integrity verification are hardcoded in the executables implementing the boot manager, the Windows loader, and the kernel. The integrity of these executables, and therefore of the hardcoded roots of trust themselves, is verified by the executables loading them: UEFI loads and verifies the boot manager, the boot manager loads and verifies the Windows loader, and the Windows loader loads and verifies the kernel.	Section 2.2
Unauthorized configuration or provisioning of the TPM	The TPM implements in hardware a concrete procedure for authorizing any processed TPM command. This includes commands for provisioning or modifying the configuration of the TPM.	Section 2.1.1
Unauthorized disabling of image integrity verification	The procedures for image integrity verification are implemented as part of the boot manager, the Windows loader, and the kernel. Therefore, apart from exploiting an implementation flaw, image integrity verification cannot be disabled in an unauthorized manner. We note that image integrity verification can be disabled by authorized users with administrative privileges. To this end, a user may modify the system's boot configuration by issuing the command `bcdedit /set nointegritychecks on`.	Section 2.2
Unauthorized modification of integrity measurements	The database of the ELAM driver is digitally signed and its signature is verified before the database is used. PCR values are used for verification of the integrity of the WBCL by attestation platforms. Values already stored in PCRs cannot be modified, only new values can be extended. All PCR operations can be conducted only by authenticated and authorized TPM users.	Section 2.3 Section 2.4
Exploitation of an implementation or design flaw	The implemented image integrity verification procedures are well-designed; secure hash algorithms are used, and sensitive data and files (e.g., the database of the ELAM driver) are protected by signing. Sensitive TPM activities (e.g., TPM provisioning and attestation reporting) are implemented as TPM-internal processes and their exposure to potentially malicious Windows users is strictly limited. We did not observe any implementation flaws during our analysis. All user data flows to the TPM is sanitized and marshalled in the form of IRPs.	Section 2.1.1 Section 2.2 Section 2.3 Section 2.4 Section 2.5

Table 3: Attack scenarios and mitigations

3 Configuration and Logging Capabilities

In this section, we provide an overview of the capabilities of Windows 10 for configuring the TPM (Section 3.1) and logging TPM events (Section 3.2). We provide recommendations for configuring the TPM and logging TPM events for the purpose of hardening platform security as part of Work Package 11.

3.1 Configuration Capabilities

Table 4 presents relevant configuration activities supported by the TPM configuration capabilities discussed below: the TPM WMI interface (*WMI*), the TBS API (*TBS API*), PowerShell (*PowerShell*), the TPM management utility (*tpm.msc*), group policy (*Policies*), and the system's registry (*Registry*). In Table 4, the configuration activities supported by a given capability are marked with `X`. We structure the activities as follows:

- Command management: Blocking or allowing the execution of TPM commands, for example, by enabling lists of blocked commands (see Section 3.1.2, paragraph 'TPM management utility');
- `OwnerAuth` management: Storing or modification of an existing, or generation of a new, owner authorization value (see Section 1.3);
- TPM provisioning: Partial or complete execution of the TPM provisioning process (see Section 2.5). This may include the generation of an SRK or an EK;
- TPM clearing: Clearing the TPM (see Section 2.5.1);
- TPM querying: Obtaining information about the TPM, for example, whether the TPM is enabled or provisioned;
- TPM lockout management: Managing the TPM lockout mechanism, for example, modifying the authorization threshold value (see Section 1.3).

Configuration activities	**TPM configuration capabilities**					
	WMI	**TBS API**	**Power-Shell**	**tpm.msc**	**Policies**	**Registry**
Command management	X	X		X	X	X
`OwnerAuth` management	X	X	X	X	X	
TPM provisioning	X	X	X	X		
TPM clearing	X	X	X	X		
TPM querying	X	X	X			
TPM lockout management	X	X	X	X	X	

Table 4: Activities supported by TPM configuration capabilities

The activities above serve as general labels and may include multiple sub-activities. For example, OwnerAuth management may include the modification of an existing, and the generation of a new, owner authorization value. We emphasize that not all configuration capabilities supporting a given activity support the same sub-activities. Some capabilities may support more sub-activities than others. The exact sub-activities supported by a given configuration capability are documented or referenced in Section 3.1.1 and Section 3.1.2.

We distinguish between programmatic and non-programmatic configuration capabilities. The former enable the configuration of the TPM through program code, whereas the latter through user interaction with the system.

3.1.1 Programmatic Configuration Capabilities

We discuss in this section the programmatic TPM configuration capabilities of Windows 10: the TPM WMI interface, and the TBS API.

The TPM WMI interface WMI is an infrastructure enabling the programmatic management of components (referred to as managed components, or managed objects), where a component may be a system, an application, or a device. These components are managed by WMI providers, which are COM objects. WMI providers implement WMI classes that declare methods performing management activities. The structure of a WMI class is described in a Managed Object Format (MOF) file.[64]

WMI classes are grouped into WMI namespaces. WMI namespaces are structured in a hierarchical manner, where the root of the hierarchy is the namespace named `root`. WMI namespaces restrict the access of users to the namespaces and classes that reside in it by implementing access control based on user privileges.[65] For more details on the architecture of the WMI infrastructure, we refer to https://msdn.microsoft.com/en-us/library/aa394553(v=vs.85).aspx [Retrieved: 22/9/2017].

The TPM device is a managed component, whose WMI interface is implemented by a WMI provider named `Trusted Platform Module Provider`.[66] This provider implements the `Win32_Tpm` WMI class for managing the TPM device.

By searching through the WMI namespace hierarchy, we identified the namespace in which the `Win32_Tpm` class resides – `root\cimv2\security\MicrosoftTPM`. WMI namespaces that reside under the `root` namespace can be enumerated with the PowerShell command `Get-WmiObject -Namespace "root" -Class "__Namespace" | Select Name`, where `Namespace` can be set to any namespace under `root`. We verifed that the `Win32_Tpm` WMI class resides in the namespace `root\cimv2\security\MicrosoftTPM` by executing the PowerShell command `Get-WmiObject -Namespace "root\cimv2\security\MicrosoftTPM" -List | Select Name`.

The methods declared by `Win32_Tpm` can be used for managing the TPM device in a programmatic manner. An example is `Clear`, which is used for clearing the TPM (see Section 2.5.1). The methods of the `Win32_Tpm` class can be executed by creating a new, or using an existing, instance of the class. This instance is referenced by the namespace in which it resides (i.e., `root\cimv2\security\MicrosoftTPM`).

The `Win32_Tpm` class is implemented in the library DLL file `%SystemRoot%\System32\wbem\Win32_Tpm.dll`. We identified this file by executing the PowerShell command `Get-WmiObject -Namespace root/cimv2/security/MicrosoftTPM -Query "select clsid from __Win32Provider where name='Win32_TpmProvider'" | % { Get-ItemProperty -Path "Registry::HKEY_CLASSES_ROOT\CLSID\$($_.clsid)\InProcServer32" -Name '(default)' }`. This command extracts the name of the library DLL file implementing the TPM WMI provider from the system's registry. The MOF file describing the structure of the `Win32_Tpm` class is located at `%\System32\wbem\Win32_Tpm.mof`.

A comprehensive documentation on the methods of the `Win32_Tpm` class is available at https://msdn.microsoft.com/en-us/library/windows/desktop/aa376484(v=vs.85).aspx [Retrieved: 22/9/2017]. For the sake of brevity and avoiding redundancy, we do not present here descriptions of these methods. We

64 https://msdn.microsoft.com/en-us/library/windows/desktop/aa823192(v=vs.85).aspx [Retrieved: 22/9/2017]
65 https://msdn.microsoft.com/en-us/library/aa822575(v=vs.85).aspx [Retrieved: 22/9/2017]
66 https://msdn.microsoft.com/en-us/library/aa376480(v=vs.85).aspx [Retrieved: 22/9/2017]

analyzed the implementation of the `Win32_Tpm` class in the `Win32_Tpm.dll` library file and identified 10 methods that are not documented on-line: `GetCapLockoutInfo`, `GetDictionaryAttackParameters`, `GetOwnerAuthForEscrow`, `GetOwnerAuthStatus`, `GetTcgLog`, `ImportOwnerAuth`, `IsFIPS`, `IsKeyAttestationCapable`, `OwnerAuthEscrowed`, and `Provision`. We then analyzed the implementation of these methods. We present here information on the methods that are relevant to the topics in the focus of this work package – integrity measurement and TPM provisioning (see Section 'Executive Summary'). We also focus on the methods that are related to the security of the TPM (see Section 1.3). In Table 5, we present the names of these methods and brief descriptions of their functionalities.

Method	Description
`GetDictionaryAttackParameters`	This method returns the configuration of the TPM's dictionary attack prevention mechanism. This includes the number of maximum authentication failures until the TPM is locked, and the amount of time that has to elapse until an authentication attempt is allowed once the TPM has been locked (see Section 1.3).
`GetTcgLog`	This method returns the WBCL (see Section 2.4).
`ImportOwnerAuth`	This method imports an owner authorization value to the system's registry. We discuss the owner authorization value and its location in the registry in Section 2.5.
`Provision`	This methods provisions the TPM as described in Section 2.5.

Table 5: Relevant methods of the Win32_Tpm class

When analyzing the implementation of the `Win32_Tpm` class, we observed that most of its methods related to TPM provisioning act as wrappers of functions implemented in the `TpmCoreProvisioning.dll` library file. In Section 2.5, we discussed relevant functionalities implemented in this file.

The TBS API The Windows operating system provides programmatic access to the TPM via the TBS API. We discussed this API and the way in which communicates with the TPM in Section 2.1.1. The TBS API allows for fine-granular configuration of the TPM by submitting TPM commands using the `Tbsip_Submit_Command` function. A detailed reference of the TBS API is available on-line at: https://msdn.microsoft.com/en-us/library/windows/desktop/aa446794(v=vs.85).aspx [Retrieved: 22/9/2017].

The use of the TBS API for TPM configuration requires knowledge on invoking the TPM commands for configuring the TPM. This includes knowledge on types of parameters and allowed parameter values and valid byte sequences (see Section 1.3). A comprehensive documentation of the commands supported by the TPM 2.0 standard is available in [TCGLP3 2016].

3.1.2 Non-programmatic Configuration Capabilities

We discuss in this section the non-programmatic capabilities of Windows 10 for configuring the TPM: PowerShell, the TPM management utility (executable: `tpm.msc`), group policies, and the system's registry.

PowerShell The TPM can be configured by executing the PowerShell cmdlets for TPM management. These cmdlets are implemented as part of the PowerShell module `TrustedPlatformModule`. A comprehensive description of the cmdlets for configuring the TPM is available on-line at: https://technet.microsoft.com/de-de/library/jj603116(v=wps.630).aspx [Retrieved: 22/9/2017]. In this section, we provide a summarizing description of these cmdlets.[67]

67 A list of cmdlets implemented as part of `TrustedPlatformModule` can be obtained by issuing the PowerShell command `Get-Command -Module TrustedPlatformModule`.

`Clear-Tpm`: This cmdlet clears the TPM and resets it to its factory state; that is, it deletes authorization values, the SRK, and other data stored at the TPM,. The EK is not deleted ([TCGLP3 2016], Section 24.6). For the TPM to be cleared, users have to provide the owner authorization value (see Section 2.5), or this value has to be stored in the system's registry. `ConvertTo-TpmOwnerAuth`: This cmdlet converts a user-specified passphrase to an owner authorization value. We analyzed the implementation of this cmdlet observing that the conversion consists of:

- encoding the passphrase using the Unicode Transformation Format (UTF)-16LE encoding algorithm;[68]
- hashing the encoded passphrase using the SHA-1 hash algorithm;[69] and
- encoding the resulting hash using the Base64 encoding algorithm.

`Disable-TpmAutoProvisioning`: This cmdlet disables TPM auto-provisioning (see Section 2.5.2). If the cmdlet parameter `OnlyForNextRestart` is specified, the auto-provisioning feature will be disabled only for the next time the system restarts. For any subsequent restart, the auto-provisioning feature will be enabled.

`Enable-TpmAutoProvisioning`: This cmdlet enables TPM auto-provisioning.

Property	Description
`TpmReady`	This property indicates whether the TPM complies with Windows Server 2012 standards.
`TpmPresent`	This property indicates whether there is a TPM installed on the platform.
`ManagedAuthLevel`	This property is the level at which the operating system manages the owner authorization value. Possible values are `Full`, `Delegated`, and `None`. A `ManagedAuthLevel` of `Full` indicates that the system stores the TPM owner authorization value and TPM delegation blobs (see Section 1.3) in the system's registry. The owner authorization value is stored at the registry key `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\TPM\WMI\Admin\OwnerAuthFull`. A `ManagedAuthLevel` of `Delegated` indicates that the system stores only TPM delegation blobs in the system's registry. A `ManagedAuthLevel` of `None` indicates that the system does not store the TPM owner authorization value or delegation blobs in the system's registry.
`OwnerClearDisabled`	This property indicates whether the TPM can be cleared by the holder of the owner authorization value. If its value is `True`, this activity is disabled.
`AutoProvisioning`	This property indicates whether TPM auto-provisioning is enabled.
`LockedOut`	This property indicates whether the TPM is locked out (see Section 1.3).
`LockoutCount`	This property provides the current count of authorization failures (see Section 1.3).
`LockoutMax`	This property provides the set maximum count of authorization failures until the TPM is locked out.
`SelfTest`	This property provides internal TPM information about the TPM self-test procedure ([TCGLP1 2016], Section 12.2.3).

Table 6: Get-Tpm cmdlet: Properties

68 https://tools.ietf.org/html/rfc2781 [Retrieved: 22/9/2017]
69 https://tools.ietf.org/html/rfc3174 [Retrieved: 22/9/2017]

`Get-Tpm`: This cmdlet provides information about the TPM. The information consists of the property values presented in Table 6.
`Get-TpmEndorsementKeyInfo`: This cmdlet provides information about the EK stored in the TPM. This information consists of the property values presented in Table 7.

Property	Description
`IsPresent`	This property indicates whether there is an EK stored in the TPM.
`PublicKey`	This property provides the public part of the EK, encoded in the ASN.1 format.
`PublicKeyHash`	This property provides a hashed value of the public part of the EK.
`ManufacturerCertificates`	This property provides the EK and platform certificates installed by the manufacturer (see Section 1.3).
`AdditionalCertificates`	This property provides EK certificates created by other entities.

Table 7: Get-TpmEndorsementKeyInfo: Properties

`Get-TpmSupportedFeature`: This cmdlet provides a list of features supported by the TPM. An example is the 'key attestation' feature indicating that the TPM supports key attestation (see Section 1.3).

`Import-TpmOwnerAuth`: This cmdlet imports an owner authorization value to the registry. This value is stored at the registry key `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\TPM\WMI\Admin\OwnerAuthFull` (see Table 6).

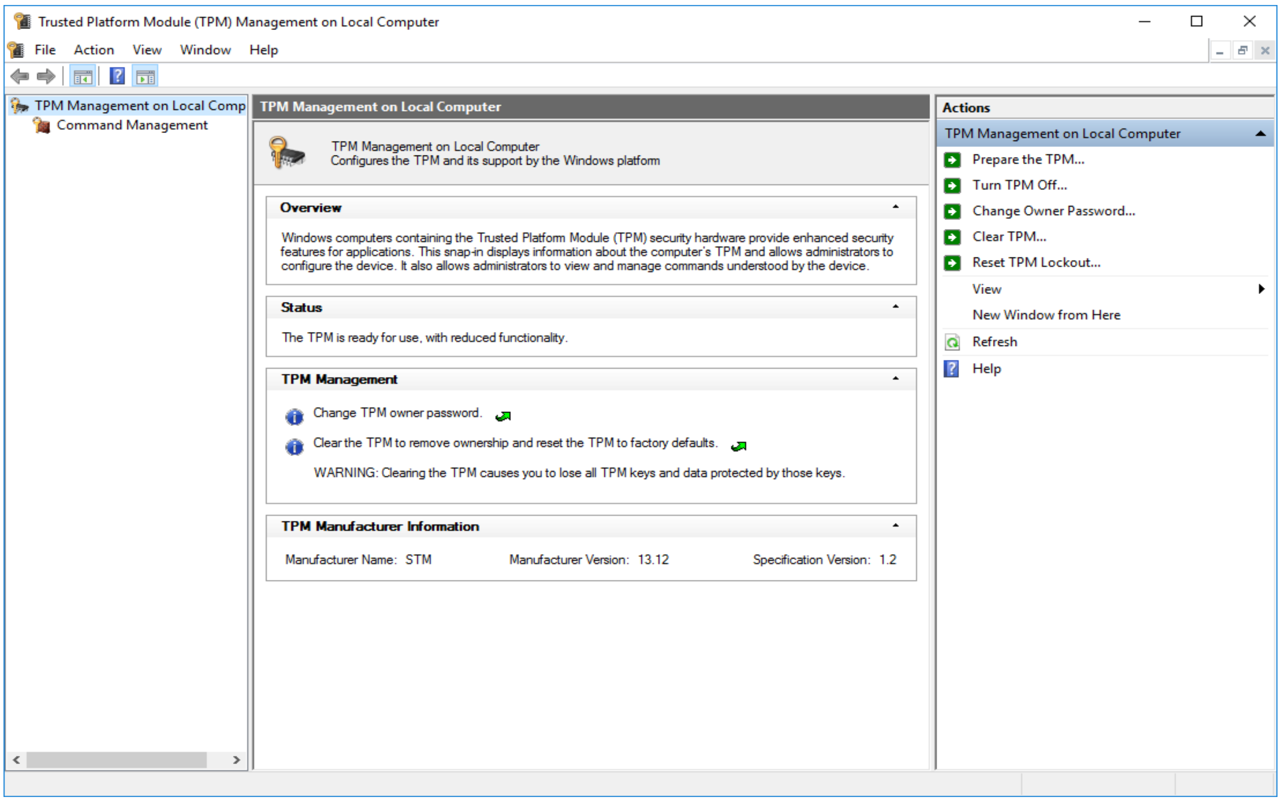
`Initialize-Tpm`: This cmdlet provisions the TPM. We discuss the TPM provisioning process in Section 2.5.

`Set-TpmOwnerAuth`: This cmdlet sets the current TPM owner authorization value to a new value. Its issuer has to provide the old owner authorization value as the `OwnerAuthorization` or `File` cmdlet parameter. If an owner authorization value is not provided, the cmdlet attemps to obtain it from the system's registry. `Set-TpmOwnerAuth` returns information about the TPM as presented in Table 6.

`Unblock-Tpm`: This cmdlet unlocks the TPM. The TPM might be locked due to a reached limit of authorization failures (see Section 1.3). Its issuer has to provide the owner authorization value as the `OwnerAuthorization` or `File` cmdlet parameter. If an owner authorization value is not provided, the cmdlet attemps to obtain it from the system's registry. `Unblock-TPM` returns information about the TPM as presented in Table 6.

TPM management utility The TPM management utility (executable: `tpm.msc`) of Windows 10 provides a graphical user interface for configuring the TPM. Its main window, depicted in Figure 39, provides information about the status of the TPM (*Status* in Figure 39). In addition, it provides information about the manufacturer and version of the TPM (*TPM Manufacturer Information* in Figure 39). It also allows for initializing/provisioning the TPM (*Prepare the TPM...* in Figure 39), turning the TPM off (*Turn TPM Off...* in Figure 39), changing the current owner authorization value (*Change Owner Password...* in Figure 39), clearing the TPM (*Clear TPM...* in Figure 39), and unlocking the TPM if it is in a locked state (*Reset TPM Lockout...* in Figure 39).

By clicking on 'Command management', users can configure access control over TPM commands by blocking a list of such commands. This list is referred to as the 'local list of blocked TPM commands'. The list may differ from the 'default list of blocked TPM commands'. The latter is a list of blocked TPM commands that is preconfigured by Windows 10.


Trusted Platform Module (TPM) Management on Local Computer
File Action View Window Help
TPM Management on Local Comp
Command Management
TPM Management on Local Computer
Actions
TPM Management on Local Computer
Prepare the TPM...
Turn TPM Off...
Change Owner Password...
Clear TPM...
Reset TPM Lockout...
View
New Window from Here
Refresh
Help

TPM Management on Local Computer
Configures the TPM and its support by the Windows platform

Overview

Windows computers containing the Trusted Platform Module (TPM) security hardware provide enhanced security features for applications. This snap-in displays information about the computer's TPM and allows administrators to configure the device. It also allows administrators to view and manage commands understood by the device.

Status

The TPM is ready for use, with reduced functionality.

TPM Management

Change TPM owner password.

Clear the TPM to remove ownership and reset the TPM to factory defaults.

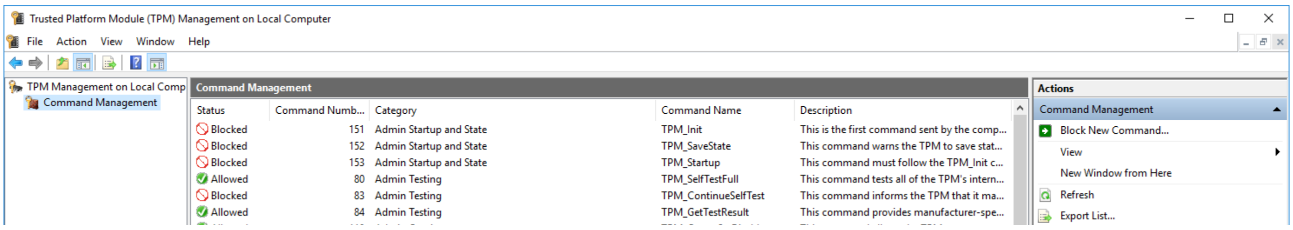
WARNING: Clearing the TPM causes you to lose all TPM keys and data protected by those keys.

TPM Manufacturer Information

Manufacturer Name:	STM	Manufacturer Version:	13.12	Specification Version:	1.2

Figure 39: The main window of the TPM management utility

Figure 40 depicts a snippet of the graphical interface for configuring access control over TPM commands. The 'Command Management' panel, among other things, displays the status of each TPM command (i.e., 'Blocked' or 'Allowed'). Users can block a TPM command by clicking on 'Block New Command...', placed in the 'Actions' panel.



Status	Command Numb...	Category	Command Name	Description
Blocked	151	Admin Startup and State	TPM_Init	This is the first command sent by the comp...
Blocked	152	Admin Startup and State	TPM_SaveState	This command warns the TPM to save stat...
Blocked	153	Admin Startup and State	TPM_Startup	This command must follow the TPM_Init c...
Allowed	80	Admin Testing	TPM_SelfTestFull	This command tests all of the TPM's intern...
Blocked	83	Admin Testing	TPM_ContinueSelfTest	This command informs the TPM that it ma...
Allowed	84	Admin Testing	TPM_GetTestResult	This command provides manufacturer-spe...

Figure 40: TPM command management using the TPM management utility

Group policy Windows 10 comes with seven group policies for configuring the TPM. These policies are located at the policy path `Computer Configuration` → `Administrative Templates` → `System` → `Trusted Platform Module Services`. Table 8 presents the names of the policies and brief descriptions.

Policy	Description
`Configure the list of blocked TPM commands`	If enabled, this policy configures the TPM such that it blocks the TPM commands specified as part of a list. This list is created by the user configuring the policy.
`Ignore the default list of blocked TPM commands`	If enabled, this policy configures the TPM such that it ignores the default list of blocked TPM commands (see Section 3.1.2, paragraph 'TPM management utility').
`Ignore the local list of blocked TPM commands`	If enabled, this policy configures the TPM such that it ignores the local list of blocked TPM commands (see Section 3.1.2, paragraph 'TPM management utility').

Policy	Description
`Configure the level of TPM owner authorization information available to the operating system`	If enabled, this policy configures the system such that it manages the owner authorization value at the level `Full`, `Delegated`, or `None` (see Table 6, property `ManagedAuthLevel`).
`Standard User Lockout Duration`	If enabled, this policy configures the TPM such that it counts failed TPM authorization attempts from users over a given time interval (in minutes). This interval is configured through this policy. If the number of failed TPM authorization attempts over the time interval exceeds a threshold value, the user is prevented from sending commands to the TPM. This policy applies only to regular system users (i.e., not administrators).
`Standard User Individual Lockout Threshold`	If enabled, this policy configures the previously mentioned TPM authorization threshold value (see policy `Standard User Lockout Duration`). The policy configures the maximum number of TPM authorization failures over a given time interval, after which the user is prevented from sending commands to the TPM. This policy applies to each standard system user individually.
`Standard User Total Lockout Threshold`	If enabled, this policy configures the previously mentioned TPM authorization threshold value (see policy `Standard User Lockout Duration`) for all users. The policy configures the maximum number of TPM authorization failures from all users over a given time interval until they are prevented from sending commands to the TPM. This policy applies to all standard system users.

Table 8: Group policies for configuring the TPM

Registry The registry of Windows 10 contains numerous keys storing values for configuring the TPM and values used during TPM operation. These keys are located in the registry hives `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\TPM` and `HKEY_LOCAL_MACHINE\Software\Microsoft\Tpm`.

`HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\TPM` contains registry keys with values primarily referenced by the TPM driver during command execution. They are also referenced in the `TpmCoreProvisioning.dll` library file during the TPM provisioning process (see Section 2.5). An example is the key `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\TPM\WMI\Admin\SRKPub`, which is the public part of the SRK. This key is populated during TPM provisioning, when a new SRK is created (see Section 2.5). In addition to the dynamic analysis of the TPM functionalities that are in the scope of this work package, we verified the use of the keys at `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\TPM` by searching for string references to this hive in library and executable files that implement these TPM functionalities. These executable files include `tpm.sys`, `tbs.dll`, and `TpmCoreProvisioning.dll`. We searched for string references to registy keys using the `Strings` utility, part of the `Sysinternals` suite.

The registry hive `HKEY_LOCAL_MACHINE\Software\Microsoft\Tpm` stores lists of allowed and blocked TPM commands (see Section 3.1.2). Figure 41 depicts the structure of this hive. The keys beginning with 'Allowed' store in the sub-keys 'List' lists of allowed TPM commands, where the name of each list entry is the ID of an allowed command (see Section 1.3) and the value of the entry is 1. The keys beginning with

'Blocked' store in the sub-keys 'List' lists of blocked TPM commands, where the name and value of each list entry is the ID of a blocked TPM command.

```
├───Tpm
│   ├───AllowedW8AppContainerCommands
│   │   └───List
│   ├───AllowedW8Commands
│   │   └───List
│   ├───AllowedW8UserCommands
│   │   └───List
│   ├───BlockedAppContainerCommands
│   │   └───List
│   ├───BlockedCommands
│   │   └───List
│   ├───BlockedUserCommands
│   │   └───List
```

Figure 41: The registry hive HKEY_LOCAL_MACHINE\Software\Microsoft\Tpm

3.1.3 Recommended Configuration Settings

The TPM configuration capabilities that can be configured through policies (see Table 8) can be grouped into three categories: command blocking, lockout, and credential storage. Table 9 describes these categories and provides recommended settings or, where necessary, relevant information for making an informed configuration decision.

Category	**Description**	**Recommended settings**
Command Blocking	Windows allows to block specific TPM commands and per default blocks certain commands.	The blocking of additional commands is not required: In addition to the TPM command blocked by default, the TPM itself is designed to block users from executing the most critical-security activities (e.g. exporting private portions of keys). The default blocking of TPM commands by Windows 10 must not be disabled (see Table 8, policy `Ignore the default list of blocked TPM commands`).

Lockout	The TPM has the capability to restrict/limit authorization attempts in order to prevent dictionary attacks.	The following settings (see Table 8) should be configured to protect the TPM from dictionary attacks:
		
		`Standard User Lockout Duration: 30`
		`Standard User Total Lockout Threshold: 5`
		
		These settings allow five failed authorization attempts over 30 minutes. The settings above have proven to be operationally feasible while providing adequate security benefit.

Credential Storage	Depending on the configuration, Windows 10 may store the owner authorization value in the registry to transparently access the TPM (see Table 8,	Obtaining the owner authorization value by an attacker requires either administrative access to a running system with hard drive encryption or physical access to a system's hard disk without hard drive encryption. If the owner authorization value is stored in the registry and an attacker has administrative access to a running

Category	Description	Recommended settings
	policy `Configure the level of TPM owner authorization information available to the operating system`). Any user and user application with access to the registry can thus also access the TPM. If the owner authorization value is not stored in the registry, it must be entered into a credential prompt when required.	system, the attacker can extract the owner authorization key from the registry and perform privileged operations. This includes resetting the TPM lockout mechanism and/or clearing the TPM. This may result in an unusable system, for example, when the Bitlocker encryption key is cleared from the TPM and the recovery key is not available upon reboot. If the registry key is not stored in the registry, an attacker with administrative privileges is in the position to intercept TPM credential prompts when they appear in order to extract the owner authorization value. If the owner authorization value is stored in the registry and an attacker has access to a system's hard disk without hard drive encryption, the attacker can extract the owner authorization key from the registry and perform privileged operations once the system is running (see the discussion above). Given that obtaining the owner authorization value by an attacker requires either access to a system's hard disk or administrative access - scenarios that lead to a full system compromise - no mandatory setting for the policy `Configure the level of TPM owner authorization information available to the operating system` is given.

Table 9: Recommended settings

3.2 Logging Capabilities

Windows 10 uses the ETW framework for logging TPM events ([ERNW WP2], Section 4). We analyzed the logging functionalities implemented in library and executable files that implement TPM functionalities relevant to this work package. These include `tpm.sys`, `tbs.dll`, and `TpmCoreProvisioning.dll`. We identified 5 ETW providers. Table 10 lists their GUIDs (column 'GUID', see Section), and the library or executable files in which they are implemented (column 'Location').

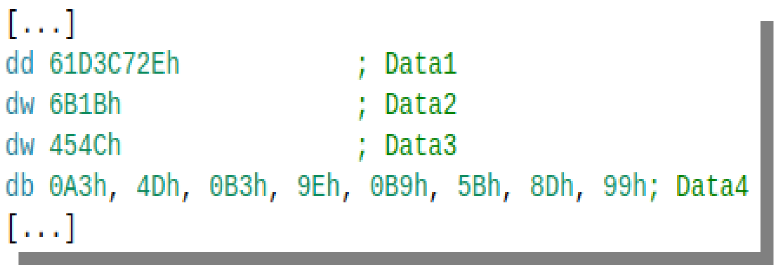
We do not claim complete coverage of all ETW providers for logging TPM events that are implemented as part of Windows 10. This is because we focussed our analysis on discovering ETW providers that log events related to the TPM functionalities investigated in this work package (see Section 'Executive Summary'). We have also identified the ETW providers that provide data to logs, which Windows makes available to system administrators via its `Event Viewer` utility for security monitoring and system maintenance purposes.

The GUIDs listed in Table 10 are pre-stored (i.e., hardcoded) in variables and can be viewed by displaying the contents of the memory locations where they are stored. For example, Figure 42 depicts output from the `IDA` dissassembler showing the GUID `61d3c72e-6b1b-454c-a34d-b39eb95b8d99` stored in a variable declared as part of `tbs.sys`.

GUID	Location
`3a8d6942-b034-48e2-b314-f69c2b4655a3`	`tpm.sys`
`1b6b0772-251b-4d42-917d-faca166bc059`	`tpm.sys`

GUID	Location
`61d3c72e-6b1b-454c-a34d-b39eb95b8d99`	`tbs.dll` `tbs.sys`
`66ba1a86-43c6-41ac-955b-28b520db532a`	`TpmCoreProvisioning.dll`
`7d5387b0-cbe0-11da-a94d-800200c9a66`	`TpmCoreProvisioning.dll`

Table 10: ETW providers for logging TPM events



```
[...]
dd 61D3C72Eh            ; Data1
dw 6B1Bh                ; Data2
dw 454Ch                ; Data3
db 0A3h, 4Dh, 0B3h, 9Eh, 0B9h, 5Bh, 8Dh, 99h; Data4
[...]
```

Figure 42: A GUID of an ETW provider declared in tbs.sys

The provider with a GUID `3a8d6942-b034-48e2-b314-f69c2b4655a3` provides data for logging to an ETW trace listener (i.e., a session) of type autologger. This session type indicates that the session may provide trace data during system booting ([ERNW WP2], Section 3.2). This session can be configured by modifying registry key values at `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\WMI\Autologger\Tpm`. It stores logged data in `%SystemRoot%\System32\LogFiles\WMI\Tpm.etl`.

The providers with GUIDs `1b6b0772-251b-4d42-917d-faca166bc059` and `7d5387b0-cbe0-11da-a94d-800200c9a66` provide data to the `EventLog-System` ETW session of type autologger. This session can be configured by modifying registry key values at `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\WMI\Autologger\Eventlog-System`. In addition, it can be configured using the `Event Viewer` utility as described in ([ERNW WP2], Section 4.3).

The providers with GUIDs `1b6b0772-251b-4d42-917d-faca166bc059` and `7d5387b0-cbe0-11da-a94d-800200c9a66` are registered under the names `TPM` and `Microsoft-Windows-TPM-WMI`, respectively. This can be verified by issuing the command `logman query providers`. This command displays names and GUIDs of ETW providers.

Since the providers with GUIDs `1b6b0772-251b-4d42-917d-faca166bc059` and `7d5387b0-cbe0-11da-a94d-800200c9a66` are registered under names, we can extract the Event IDs under which the providers may log events. Table 11 and Table 12 lists these Event IDs and their descriptions (column 'Event ID' and 'Event message', respectively). In Table 11 and 12, %1 and %2 mark dynamic content generated at run-time. The event descriptions in Table 11 and 12 are as provided by Microsoft. We displayed the Event IDs and the descriptions using the `wevtutil` utility, as described in ([ERNW WP2], Section 4.3).

Event ID	Event Message
2	The TPM self test command failed.
12	The device driver for the Trusted Platform Module (TPM) encountered an error in the TPM hardware, which might prevent some applications using TPM services from operating correctly. Please restart your computer to reset the TPM hardware. For further assistance on this hardware issue, please contact the computer manufacturer for more information.

Event ID	Event Message
14	The device driver for the Trusted Platform Module (TPM) encountered a non-recoverable error in the TPM hardware, which prevents TPM services (such as data encryption) from being used. For further help, please contact the computer manufacturer.
15	The device driver for the Trusted Platform Module (TPM) encountered a non-recoverable error in the TPM hardware, which prevents TPM services (such as data encryption) from being used. For further help, please contact the computer manufacturer.
16	A compatible TPM is not found.
17	The Trusted Platform Module (TPM) hardware failed to execute a TPM command.
18	This event triggers the Trusted Platform Module (TPM) provisioning/status check to run.
19	The system firmware failed to enable overwriting of system memory on restart. The ACPI request could not be interpreted by the firmware. The firmware should be upgraded.
20	A command was sent to the Trusted Platform Module (TPM) successfully resetting the TPM lockout logic. This event is generated when a successful command sent to the TPM resets the TPM lockout logic. With this event, all prior standard user TPM authorization failures are ignored; allowing standard users to use the TPM normally again immediately.
21	A standard user issued Trusted Platform Module (TPM) command returned an authorization failure. This event is generated when a command sent to the TPM by a standard user returns a response indicating an authorization failure. If too many authorization failures occur, standard users may be temporarily prevented from sending TPM commands requiring authorization. This helps prevent the TPM from entering a hardware lockout because of too many authorization failures. User Security ID:%1. Process Path %2.
22	TPM Base Services (TBS) has been configured in a test mode until the next full restart. The TBS will not perform TPM resource virtualization or TPM command blocking until the next full restart.
23	A standard user Trusted Platform Module (TPM) command was blocked because the standard user has exceeded the maximum authorization failures permitted. This event is generated when too many recent TPM commands sent to the TPM by a standard user returned a response indicating an authorization failure. The standard user is currently temporarily prevented from sending TPM commands requiring authorization. This helps prevent the TPM from entering a hardware lockout because of too many authorization failures. User Security ID:%1.
24	The Trusted Platform Module (TPM) status: %1 and %2.
25	Creation of the Windows AIK directory failed.

Table 11: Event IDs generated by the ETW provider TPM

Event ID	Event Message
513	TPM Owner Authorization information was backed up successfully to Active Directory Domain Services.
514	Failed to backup TPM Owner Authorization information to Active Directory Domain Services.

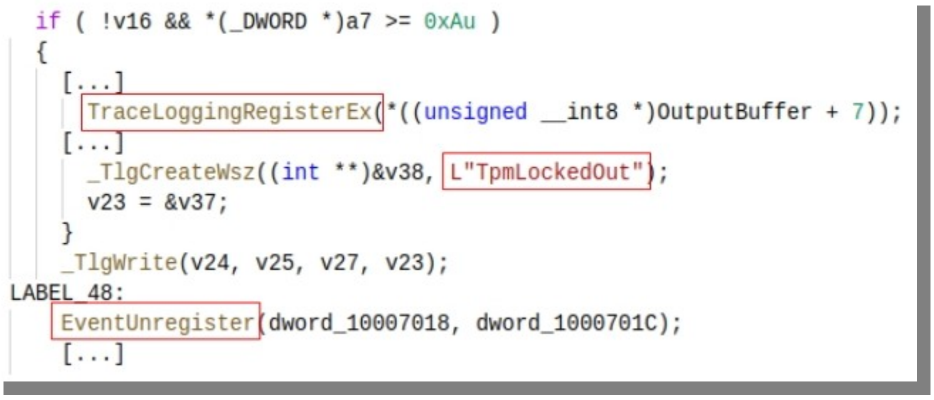
Event ID	Event Message
	Errorcode: %1 Check that your computer is connected to the domain. If your computer is connected to the domain, have your Domain Administrator check that the Active Directory schema is appropriate for backup of Windows 8 TPM Owner Authorization information and that the current Computer object has write permission to the TPM object. Installations of Windows Server 2008 R2 or before need a schema extension in order to be ready for backup of Windows 8 TPM Owner Authorization information. Consult online documentation for more information about setting up Active Directory Domain Services for TPM.
769	TPM Owner Authorization configuration changed from '%1' to '%2'.
1025	The TPM was successfully provisioned and is now ready for use.
1026	The Trusted Platform Module (TPM) hardware on this computer cannot be provisioned for use automatically. To set up the TPM interactively use the TPM management console (Start->tpm.msc) and use the action to make the TPM ready. Error: %1 Additional Information: %2
1027	The Ownership of the Trusted Platform Module (TPM) hardware on this computer was successfully taken (TPM TakeOwnership command) by the system.
1028	The NGC key generation task was successfully triggered.
1029	The triggering of the NGC key generation task failed.
1030	The NGC certificate enrollment task was successfully triggered.
1031	The triggering of the NGC certificate enrollment task failed.
1281	This event triggers the TBS device identifier generation.
1282	The TBS device identifier has been generated.
1537	The Device Health Certificate was successfully provisioned from %1.
1538	The Device Health Certificate provisioning could not connect to %1. %2
1539	The Device Health Certificate could not be provisioned from %1. HTTP status code %2: %3
1793	The Trusted Platform Module (TPM) hardware on this computer is scheduled to be cleared by the system.

Table 12: Event IDs generated by the ETW provider Microsoft-Windows-TPM-WMI

The providers with GUIDs `61d3c72e-6b1b-454c-a34d-b39eb95b8d99` and `66ba1a86-43c6-41ac-955b-28b520db532a` are not registered under names and there are no sessions that are configured to obtain data from them. We verified this by searching the system's registry for the GUIDs. This is where ETW sessions store the GUIDs of the ETW providers they receive data from. We searched the system's registry using the `Registry Editor` utility.

Our analysis revealed that the providers with GUIDs `61d3c72e-6b1b-454c-a34d-b39eb95b8d99` and `66ba1a86-43c6-41ac-955b-28b520db532a` are special-purpose providers that are registered and activated only temporarily when specific activities are taking place. For example, the provider with GUID `61d3c72e-6b1b-454c-a34d-b39eb95b8d99` is primarly used for tracing events related to TPM authorization failures. Figure 43 depicts pseudo-code generated by the `IDA` disassembler implementing the registration and the deregistration of this provider. The provider is registered and deregistered by invoking the functions `TraceLoggingRegisterEx`[70] and `EventUnregister`,[71] respectively.

In Figure 43, the provider with GUID `61d3c72e-6b1b-454c-a34d-b39eb95b8d99` is used for logging a `TPMLockedOut` event. This event ocurrs when the TPM has been locked due to too many authorization failures (see Section 1.3). The pseudo-code depicted in Figure 43 is of the `Tbsip_Submit_Command_Internal` function, implemented in the `tbs.dll` library file (see Figure 4).

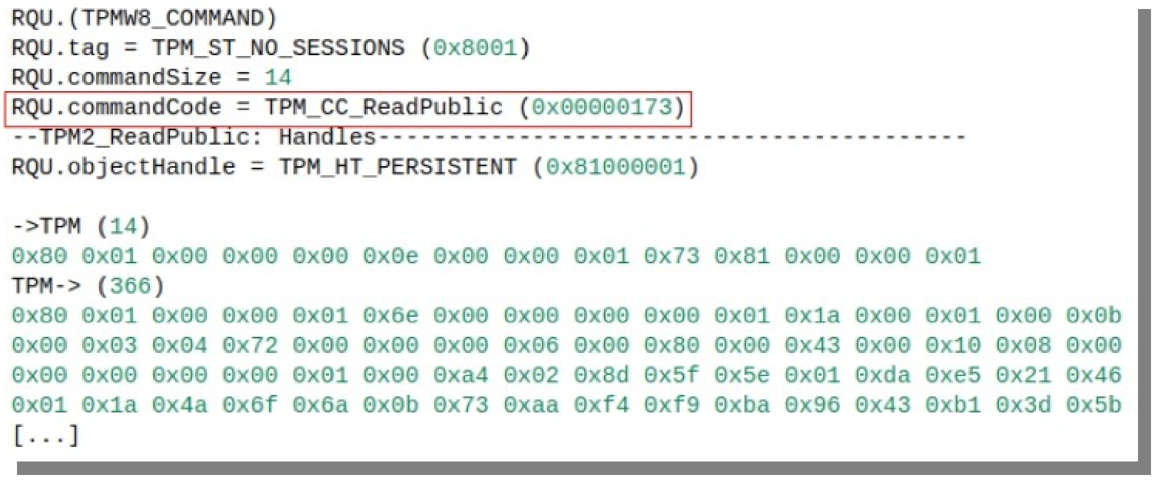


```
  if ( !v16 && *(_DWORD *)a7 >= 0xAu )
  {
    [...]
    TraceLoggingRegisterEx(*((unsigned __int8 *)OutputBuffer + 7));
    [...]
    _TlgCreateWsz((int **)&v38, L"TpmLockedOut");
    v23 = &v37;
    }
    _TlgWrite(v24, v25, v27, v23);
LABEL_48:
    EventUnregister(dword_10007018, dword_1000701C);
    [...]
```

Figure 43: Registration and deregistration of the provider with GUID 61d3c72e-6b1b-454c-a34d-b39eb95b8d99

We note the existence of the mechanism for logging TPM commands issued by the `Platform Cryptographic Provider` (see Section 2.1.1). A user can enable this mechanism by creating a registry key of type `REG_SZ` and with name `ProviderTraces` under the registry hive `HKEY_LOCAL_MACHINE\Software\Microsoft\Tpm`. This key should contain a path to a folder, which is where log files will be stored.

We enabled the above mechanism on Windows 10. We observed that the produced log files contain detailed information on submitted TPM commands. This information is structured in a way such that it presents values of variables and data structures as defined in the TPM specifications provided by the TCG. This includes command codes and command byte sequences [TCGLP2 2016]. The logged information makes the log files a useful and valuable resource for monitoring TPM activities at command level. Figure 44 depicts a snippet of a log file, where 'RQU.commandCode' specifies the command code `TPM_CC_ReadPublic`. This code uniquely identifies the `TPM2_NV_ReadPublic` command ([TCGLP3 2016], Section 31.6).



```
RQU.(TPMW8_COMMAND)
RQU.tag = TPM_ST_NO_SESSIONS (0x8001)
RQU.commandSize = 14
RQU.commandCode = TPM_CC_ReadPublic (0x00000173)
--TPM2_ReadPublic: Handles-----------------------------------------
RQU.objectHandle = TPM_HT_PERSISTENT (0x81000001)

->TPM (14)
0x80 0x01 0x00 0x00 0x00 0x0e 0x00 0x00 0x01 0x73 0x81 0x00 0x00 0x01
TPM-> (366)
0x80 0x01 0x00 0x00 0x01 0x6e 0x00 0x00 0x00 0x00 0x01 0x1a 0x00 0x01 0x00 0x0b
0x00 0x03 0x04 0x72 0x00 0x00 0x00 0x06 0x00 0x80 0x00 0x43 0x00 0x10 0x08 0x00
0x00 0x00 0x00 0x00 0x01 0x00 0xa4 0x02 0x8d 0x5f 0x5e 0x01 0xda 0xe5 0x21 0x46
0x01 0x1a 0x4a 0x6f 0x6a 0x0b 0x73 0xaa 0xf4 0xf9 0xba 0x96 0x43 0xb1 0x3d 0x5b
[...]
```

Figure 44: A snippet of a log file containing TPM command information

70 https://msdn.microsoft.com/en-us/library/windows/desktop/dn933290(v=vs.85).aspx [Retrieved: 22/9/2017]
71 https://msdn.microsoft.com/de-de/library/windows/desktop/aa363749(v=vs.85).aspx [Retrieved: 22/9/2017]

Appendix

Tools

Tool	Availability and Description
`Chipsec`	*Availability*: https://github.com/chipsec [Retrieved: 7/5/2017] *Description*: A framework for analyzing hardware, system firmware (BIOS/UEFI), and platform components.
`IDA`	*Availability:* https://www.hex-rays.com/products/ida/index.shtml [Retrieved: 7/5/2017] *Description*: A disassembly and debugging framework.
Sysinternals Suite	*Availability:* https://technet.microsoft.com/de-de/sysinternals/bb545021.aspx [Retrieved: 7/5/2017] *Description*: A suite of tools for analyzing the Windows system (e.g., analyzing operation of processes, services, and enumeration of loaded libraries by processess).
wevtutil	*Availability:* Distributed with Windows 10 *Description*: A tool for querying running logging mechanisms.
Windows Debugger (`windbg`)	*Availability:* https://developer.microsoft.com/en-us/windows/hardware/download-windbg [Retrieved: 7/5/2017] *Description*: A debugger for the Windows system.
RW	*Availability:* https://rweverything.com/ [Retrieved: 22/9/2017] *Description*: A tool for reading data stored in computer hardware.
diskpart	*Availability:* Distributed with Windows 10 *Description*: A tool for formatting hard disks.
bmzip	*Availability:* https://github.com/coderforlife/bmzip [Retrieved: 22/9/2017] *Description*: A tool for decompressing the boot manager of Windows 10.
radare2	*Availability:* http://rada.re/r/ [Retrieved: 22/9/2017] *Description*: A disassembly and reversing framework.
sdb	*Availability:* *http://www.geoffchappell.com/studies/windows/win32/apphelp/sdb/shimdbdc.htm* [Retrieved: 22/9/2017]

	Description: A decompiler of files of the Windows compatibility database.
PCPtool	*Availability*: *https://github.com/Microsoft/TSS.MSR/tree/master/PCPTool.v11* [Retrieved: 22/9/2017] *Description*: A tool for interacting with the TPM.
openssl	*Availability*: *https://www.openssl.org/* [Retrieved: 22/9/2017] *Description*: A general purpose cryptography tool for working with secure sockets layer (SSL) and transport layer security (TLS) protocols.

TPM Usage Profiler

```
$$ ** Script usage: [function breakpoint] "$$>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 ([ERNW WP2], Section 2.1)
lsass.exe		Local security authority ([ERNW WP2], Section 2.1)
svchost.exe	-k netsvcs	BitLocker Drive Encryption Service
taskhostw.exe		Host process for Windows tasks (see Section 2.5.2)
svchost.exe	-k netsvcs	Microsoft Account Sign-in Assistant

ELAM Database Parser

```
#!/usr/bin/env python3
import argparse
import struct
```

```
from collections import namedtuple
import enum

from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_v1_5
from Crypto.Util.number import bytes_to_long, long_to_bytes

from helperlib import print_hexII, print_hexdump, hexdump


class Item(namedtuple('Item', ['code', 'type', 'trust_code', 'data', 'comment'])):
    def __str__(self):
        l = [
            'Item:',
            f'\tcode: {self.code:X}',
            f'\ttype: {self.type!r}',
            f'\ttrust: {self.trust_code!r}',
            f'\tdata:\n\t' + '\n\t'.join(hexdump(self.data, header=True)),
            f'\tcomment: {self.comment}',
        ]
        return '\n'.join(l)

class EntryType(enum.IntEnum):
    THUMBPRINT_HASH = 1
    CERTIFICATE_PUBLISHER = 1
    ISSUER_NAME = 1
    IMAGE_HASH = 4
    VERSION_INFO = 7

class TrustLevel(enum.IntEnum):
    KnownGoodImage = 0
    KnownBadImage = 1
    UnknownImage_2 = 2
    UnknownImage = 3
    KnownBadImageBootCritical = 4

PUBLIC_EXPONENT = 0x010001
MODULUS = int.from_bytes(struct.pack("256B", *[
    0xb3, 0x95, 0xde, 0x5b, 0xc2, 0xe1, 0x89, 0xf7, 0x56, 0xc2, 0x20,
    0xbf, 0x27, 0xd2, 0x88, 0x1a, 0x0a, 0xac, 0xdb, 0xc7, 0x19, 0x36,
    0x7b, 0xce, 0x37, 0x83, 0xd1, 0xec, 0x42, 0xd3, 0xab, 0x30, 0x54,
    0xa5, 0x51, 0x11, 0xd8, 0xcc, 0xec, 0x80, 0xab, 0x89, 0x5a, 0xae,
    0x18, 0x71, 0x11, 0x7c, 0x85, 0x1a, 0x1a, 0x53, 0x54, 0x46, 0x3e,
    0x55, 0x5c, 0x43, 0x5d, 0x4b, 0x9f, 0xc7, 0x54, 0x57, 0x75, 0xc5,
    0x02, 0xe2, 0x63, 0xa9, 0x94, 0x56, 0xa7, 0x3b, 0xe0, 0xc3, 0xed,
    0x5f, 0x66, 0x9d, 0x60, 0x78, 0x1e, 0xac, 0x92, 0x3d, 0x48, 0xe9,
    0x51, 0x5d, 0x79, 0x2a, 0x22, 0x9a, 0x9e, 0xd3, 0xbc, 0x15, 0xbe,
    0x7a, 0x4e, 0x97, 0xe8, 0x1f, 0x9c, 0x80, 0xf5, 0xfb, 0x94, 0x0b,
    0x5f, 0xb7, 0x6f, 0x0d, 0x57, 0xa0, 0x09, 0x55, 0x68, 0x78, 0xf3,
    0x5d, 0x7b, 0x9a, 0x9b, 0x08, 0xa3, 0xa6, 0x41, 0x18, 0xf0, 0x17,
    0x11, 0x89, 0x9b, 0x71, 0x73, 0x27, 0xa2, 0x55, 0x51, 0xc0, 0xee,
    0xa5, 0x70, 0x6f, 0xb8, 0x40, 0x2a, 0x85, 0xe9, 0x91, 0x20, 0x4b,
    0x0c, 0xd2, 0x29, 0xa2, 0x01, 0x36, 0x96, 0x1c, 0xbb, 0xd5, 0xef,
    0x95, 0x68, 0x43, 0xfb, 0x77, 0x42, 0x88, 0x1a, 0xae, 0x60, 0x14,
    0xfe, 0x0b, 0x0d, 0xd3, 0x28, 0x04, 0x98, 0x15, 0x71, 0x3e, 0xba,
    0xb3, 0x80, 0x65, 0x6d, 0x2b, 0x7f, 0x30, 0xca, 0xf2, 0x6c, 0xa6,
    0x47, 0xd3, 0x3c, 0x57, 0x50, 0x0d, 0xb3, 0xbb, 0xed, 0x6d, 0x75,
    0xf2, 0x0f, 0x26, 0x29, 0xf7, 0xc6, 0xe4, 0x20, 0x5e, 0xaf, 0x87,
    0xf1, 0x8b, 0x8e, 0x57, 0x99, 0x00, 0xf3, 0x84, 0xe5, 0x25, 0x10,
    0x05, 0x2c, 0xeb, 0x77, 0xa3, 0xdb, 0xbd, 0x7e, 0xd4, 0xb5, 0x60,
    0xb6, 0x6a, 0xa0, 0x99, 0x25, 0x59, 0x2f, 0x10, 0x69, 0xf4, 0x62,
    0xe1, 0x8c, 0x2b]), 'big')

MODULUS = int.from_bytes(
bytes.fromhex('b395de5bc2e189f756c220bf27d2881a0aacdbc719367bce3783d1ec42d3ab3054a55111d8
ccec80ab895aae1871117c851a1a5354463e555c435d4b9fc7545775c502e263a99456a73be0c3ed5f669d607
81eac923d48e9515d792a229a9ed3bc15be7a4e97e81f9c80f5fb940b5fb76f0d57a009556878f35d7b9a9b08
a3a64118f01711899b717327a25551c0eea5706fb8402a85e991204b0cd229a20136961cbbd5ef956843fb774
2881aae6014fe0b0dd328049815713ebab380656d2b7f30caf26ca647d33c57500db3bbed6d75f20f2629f7c6
e4205eaf87f18b8e579900f384e52510052ceb77a3dbbd7ed4b560b66aa09925592f1069f462e18c2b'),
'big')

PUBLIC_KEY = RSA.construct((MODULUS, PUBLIC_EXPONENT))

def parse(fp):
    tag = fp.read(1)[0]
    size = struct.unpack('<I', fp.read(3) + b'\0')[0]
    return tag, fp.read(size)

def main(argv=None):
    parser = argparse.ArgumentParser()
    parser.add_argument('FILE', type=argparse.FileType('rb'))
    parser.add_argument('-l', '--level', choices=('debug', 'info', 'warning', 'error'),
default='info')

    args = parser.parse_args(args=argv)
    default_config(level=args.level.upper())
    fp = args.FILE

    code = 0x80000000
    while True:
        try:
            tag, data = parse(fp)
            if tag == 0x5C:
                assert len(data) >= 4
                code = struct.unpack_from('<I', data)[0]
            elif tag == 0x5D:
                code = 0x80000000
            elif tag == 0xA9:
                assert len(data) >= 4
                offset = struct.unpack_from('<I', data)[0] + 4
                assert offset < len(data)
                some_type = struct.unpack_from('<B', data[offset:])[0]
                if some_type == 9:
                    data_type, trust_code = struct.unpack_from('<BB', data[4:])
                    some_data = data[6:offset]
                    item = Item(code, EntryType(data_type), TrustLevel(trust_code),
                                some_data, data[offset + 2:])
                    print(str(item))
            elif tag == 0xAC:
                print("Encrypted Signature:")
                print_hexdump(data, colored=True, header=True)
                signature = PUBLIC_KEY.encrypt(bytes(reversed(data)), None)[0]
                print("Decrypted Signature (DER):")
                print_hexdump(signature, colored=True, header=True, folded=True)
                signature = signature.split(b'\x00', 1)[1]
                try:
                    assert signature[0] == 0x30
                    l = signature[1]
                    signature = signature[2:2+l]
                    assert signature[0] == 0x30

                    l = signature[1]
                    algo = signature[2:2+l]
                    hashsum = signature[2+l:]

                    assert algo[0] == 0x6
                    l = algo[1]
                    algo = algo[2:l+2]

                    a = algo[0]
                    b = a % 40
                    a = a // 40
                    oid = [a, b] + list(algo[1:])
                    print("HashingAlgorithm:", '.'.join(map(str, oid)))

                    assert hashsum[0] == 0x4
                    l = hashsum[1]
                    hashsum = hashsum[2:l+2]
                    print("Hash:", hashsum.hex())
                except:
                    pass
            else:
                raise ValueError("Unknown tag {:02x}".format(tag))
        except IndexError:
            break

if __name__ == '__main__':
       main()
```

WBCL Parser

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Xml;
using System.Collections.Generic;

namespace WCBL_parse
{
    class Program
    {
        static void Main(string[] args)
        {
            XmlDocument doc = new XmlDocument();
            doc.Load(@"C:\wbcl_xml.log");

            System.Console.WriteLine("** PCRs to event types **");

            foreach(XmlNode EvNode in doc.DocumentElement.ChildNodes[1].ChildNodes)
            {
                Console.WriteLine("Event: " + EvNode.Name + "; PCR: " +
                EvNode.Attributes["PCR"].Value);
            }

            System.Console.WriteLine("** Measured executables **");

            XmlNodeList files = doc.GetElementsByTagName("FilePath");
            foreach (XmlNode file in files)
                Console.WriteLine(file.InnerXml );
        }



    }
}
```

Measured Executables

In the list below, the string '%drivers%' is an alias of the path `%SystemRoot\System32\drivers`.

\EFI\Microsoft\Boot\en-US\bootmgfw.efi.MUI

\Windows\system32\winload.efi
%drivers\cng.sys
%drivers\NETIO.SYS
%drivers\vmbus.sys
%drivers\WdBoot.sys
%drivers\isapnp.sys
%drivers\vmbkmcl.sys
%drivers\CLASSPNP.SYS
%drivers\Wdf01000.sys
%drivers\amdsata.sys
%drivers\usbehci.sys
\Windows\system32\kd.dll
%drivers\mvumis.sys
%drivers\vdrvroot.sys
%drivers\partmgr.sys
%drivers\nvstor.sys
%drivers\volsnap.sys
%drivers\megasr.sys
%drivers\acpiex.sys
%drivers\bxvbda.sys
%drivers\volmgr.sys
%drivers\lsi_sss.sys
%drivers\msisadrv.sys
\Windows\system32\hal.dll
%drivers\CLFS.SYS
%drivers\PCIIDEX.SYS
%drivers\werkernel.sys
%drivers\volmgrx.sys
%drivers\WdFilter.sys
%drivers\atapi.sys
%drivers\fvevol.sys
%drivers\pcw.sys
%drivers\USBPORT.SYS
%drivers\sbp2port.sys
%drivers\sdbus.sys
%drivers\NTFS.sys
\Windows\boot\resources\bootres.dll
%drivers\ataport.SYS
%drivers\lsi_sas3i.sys
%drivers\amdsbs.sys
\Windows\system32\ntoskrnl.exe
%drivers\Wof.sys
%drivers\usbhub.sys
%drivers\storport.sys
%drivers\pdc.sys
%drivers\sdstor.sys
%drivers\tcpip.sys
%drivers\UsbHub3.sys
%drivers\NDIS.SYS
\Windows\system32\CI.dll
%drivers\stexstor.sys
%drivers\EhStorClass.sys
%drivers\rdyboost.sys
%drivers\tm.sys
%drivers\mountmgr.sys
%drivers\cmimcext.sys
%drivers\sisraid4.sys
%drivers\HpSAMD.sys
%drivers\USBD.SYS
%drivers\SiSRaid2.sys
%drivers\ucx01000.sys
%drivers\usbccgp.sys
\Windows\system32\mcupdate_GenuineIntel.dll
%drivers\evbda.sys
%drivers\arcsas.sys
%drivers\msrpc.sys
%drivers\hvsocket.sys
%drivers\FLTMGR.SYS
%drivers\clipsp.sys
%drivers\storufs.sys
%drivers\ntosext.sys
%drivers\ADP80XX.SYS
%drivers\vsmraid.sys
%drivers\ksecdd.sys
%drivers\USBXHCI.SYS
%drivers\uaspstor.sys
\Windows\system32\PSHED.dll
%drivers\storvsc.sys
%drivers\iorate.sys
\Windows\system32\BOOTVID.dll
%drivers\tpm.sys
%drivers\CEA.sys
%drivers\intelpep.sys
%drivers\vstxraid.sys
%drivers\hwpolicy.sys
%drivers\iaStorV.sys
%drivers\mup.sys
%drivers\USBSTOR.SYS
%drivers\stornvme.sys
%drivers\pci.sys

%drivers\fileinfo.sys
%drivers\3ware.sys
%drivers\cht4sx64.sys
%drivers\vmstorfl.sys
%drivers\disk.sys
%drivers\amdxata.sys
%drivers\WDFLDR.SYS
%drivers\winhv.sys
%drivers\WindowsTrustedRTProxy.sys
%drivers\Fs_Rec.sys
%drivers\megasas.sys
%drivers\cnghwassist.sys
%drivers\scmbus.sys
%drivers\WppRecorder.sys
%drivers\pciide.sys
\Windows\system32\en-US\winload.efi.MUI
%drivers\wfplwfs.sys
%drivers\lsi_sas2i.sys
%drivers\ksecpkg.sys
%drivers\intelide.sys
%drivers\fwpkclnt.sys
%drivers\storahci.sys
%drivers\ACPI.sys
%drivers\pcmcia.sys
%drivers\iaStorAV.sys
\Windows\system32\ApiSetSchema.dll
%drivers\spaceport.sys
%drivers\EhStorTcgDrv.sys
%drivers\percsas2i.sys
%drivers\nvraid.sys
%drivers\WMILIB.SYS
%drivers\WindowsTrustedRT.sys
%drivers\percsas3i.sys
%drivers\lsi_sas.sys
%drivers\volume.sys

Reference Documentation

ERNW WP2	ERNW GmbH: SiSyPHuS Win10 (Studie zu Systemaufbau, Protokollierung, Härtung und Sicherheitsfunktionen in Windows 10): Work Package 2
TCGLP1 2016	Trusted Computing Group (TCG): Trusted Platform Module Library Part 1: Architecture
Butterworth 2013	Butterworth, John; Kallenberg, Corey; Kovah, Xeno; Herzog, Amy : Problems with the Static Root of Trust for Measurement
TCGF 2016	Trusted Computing Group (TCG): TCG PC Client Platform Firmware Profile Specification
TCGAIK 2011	Trusted Computing Group (TCG): A CMC Profile for AIK Certificate Enrollment
TCGCMC 2013	Trusted Computing Group (TCG): CMC Profile for EK/PlatformCertificate Enrollment for TPMv1.2
TCGLP3 2016	Trusted Computing Group (TCG): Trusted Platform Module Library Part 3: Commands
TCGLP2 2016	Trusted Computing Group (TCG): Trusted Platform Module Library Part 2: Structures
Proudler 2014	Proudler, Graeme; Chen, Liqun; Dalton, Christopher: Trusted Computing Platforms: TPM 2.0 in Context
TCGPro 2014	Trusted Computing Group (TCG): Protection Profile, PC Client Specific Trusted Platform Module, TPM Family 1.2
TCGACPI 2017	Trusted Computing Group (TCG): TCG ACPI Specification
Russinovich 2012	Russinovich, Mark E.; Solomon, David A.; Ionescu, Alex: Windows Internals, Part 2
MicXCA 2017	Microsoft Corporation: Xpress Compression Algorithm
UEFIFE	UEFI Forum: Unified Extensible Firmware Interface (UEFI) Specification
Mic 2008	Microsoft Corporation: Windows Authenticode Portable Executable Signature Format
ERNW WP7	ERNW GmbH: SiSyPHuS Win10 (Studie zu Systemaufbau, Protokollierung, Härtung und Sicherheitsfunktionen in Windows 10): Work Package 7
Thom 2016	Thom, Stefan; Loeser, Jork; Aigner, Ron; England, Paul; Spiger, Rob; Morgan, Jim: Using the Windows 8 Platform Crypto Provider and Associated TPM Functionality
TCGEP 2016	Trusted Computing Group (TCG): TCG EFI Protocol Specification
TCGPCI 2013	Trusted Computing Group (TCG): TCG PC Client Specific TPM Interface Specification (TIS)
ERNW WP4	ERNW GmbH: SiSyPHuS Win10 (Studie zu Systemaufbau, Protokollierung, Härtung und Sicherheitsfunktionen in Windows 10): Work Package 4
Gulley 2013	Gulley, Sean; Gopal, Vinodh; Yap, Kirk; Feghali, Wajdi; Guilford, Jim; Wolrich, Gil : Intel® SHA Extensions
TCGPP 2015	Trusted Computing Group (TCG): TCG PC Client Platform: Physical Presence Interface Specification

Keywords and Abbreviations

Abbreviations........ 77
Abstract Syntax Notation One........ 27, 59
Advanced Configuration and Power Interface........ 17ff., 45, 65, 75
application programming interface........ 8, 10, 17f., 27, 29, 55ff.
attestation identity key........ 13, 53, 65
attestation identity keys........ 13f.
binary large object........ 15
binary large objects........ 58
Boot Configuration Database........ 23
Bundesamt für Sicherheit in der Informationstechnik........ 7, 11
certificate authority........ 13f.
Certificate Management Messages Over CMS........ 14
component object model........ 52, 56
core root of trust for measurement........ 13, 44f.
Cryptography API: Next Generation........ 8, 20ff., 39, 49, 53
Distinguished Encoding Rules........ 25f., 72
Dynamic Link Library........ 12, 21f., 26, 28, 56, 69, 74f.
early launch anti-malware........ 7ff., 30, 36ff., 42, 53f., 70
endorsement key........ 9, 13ff., 46f., 50f., 58f.
endorsement keys........ 13
Event Tracing for Windows........ 10, 29, 63ff.
Extensible Firmware Interface........ 43ff., 73
Extensible Markup Language........ 30, 42ff., 52
functional device objects........ 17
global unique identifier........ 10, 63f., 66f.
global unique identifiers........ 45, 63f., 66
input/output........ 9, 29
input/output request packet........ 17ff., 22, 50ff., 70
input/output request packets........ 9, 17ff., 22, 50ff., 54, 70
long-term servicing branch........ 7, 11
Managed Object Format........ 56
operating system........ 41f., 45
physical device objects........ 17
platform configuration register........ 13, 42ff., 54
platform configuration registers........ 13, 41, 43ff., 54
portable executable........ 33f.
power-on self-test........ 45
process ID........ 22, 70
read-only memory........ 45
remote procedure call........ 34
remote procedure calls........ 34
root of trust for measurement........ 8, 13
root of trust for reporting........ 8, 13
root of trust for storage........ 8, 13
Secure Hash Algorithm........ 34, 40, 43, 46, 58
*secure sock*ets layer........ 69
static root of trust for measurement........ 13, 44f.
storage root key........ 9, 13f., 46f., 50f., 58, 61
TPM Base Services........ 8, 10, 17f., 20ff., 42, 55ff., 65f.
*transport l*ayer security........ 69
Trusted Computing Group........ 13, 17, 19, 42ff., 67

Trusted Platform Module........7ff., 27, 29, 35f., 40ff., 46ff., 69f.
Unicode Transformation Format........58
*Unified Ext*ensible Firmware Interface........7, 11, 23ff., 41, 44f., 54, 68
Windows Boot Configuration Log........9, 42ff., 53f., 57
Windows Boot Configuration Logs........44
Windows Management Instrumentation........10, 47ff., 53, 55f., 58f., 61, 64, 66