Federal Office
for Information Security

Work Package 6: Virtual Secure Mode

Version: 1.0

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
© Federal Office for Information Security 2018

Table of Contents

1 Introduction 5
1.1 Zusammenfassung 5
1.2 Executive Summary 10
1.3 Concepts and Terms 15
1.3.1 Virtualization 15
1.3.2 Virtual Secure Mode 17
2 Technical Analysis of Functionalities 20
2.1 VSM Initialization 20
2.1.1 OslSetVsmPolicy 21
2.1.2 OslArchHypervisorSetup 23
2.1.3 OslArchHypercallSetup 25
2.1.4 OslFwProtectSecureBootVariables 26
2.1.5 OslVsmSetup 27
2.1.6 Instantiation of IUM Applications 28
2.2 Communication Interfaces 28
2.2.1 IUM System Calls 29
2.2.2 Hypercalls 31
2.2.3 Secure Services 33
2.2.4 Normal-mode Services 34
2.2.5 Security aspects 38
2.3 Threats and Mitigations 43
3 Configuration and Logging Capabilities 46
3.1 Hardware and Software Requirements 46
3.2 Configuration Capabilities 47
3.3 Logging Capabilities 49
Appendix 52
Tools 52
Secure Services 52
Reference Documentation 56
Keywords and Abbreviations 57

Figures

Figure 1: Architecture of a virtualized Windows environment 15
Figure 2: Address translation 16
Figure 3: Architecture of a VSM-enabled Windows environment 17
Figure 4: Memory region mapped to lsaIso.exe 19
Figure 5: The booting process of Windows 10 with VSM enabled 20
Figure 6: Pseudo-code of the implementation of OslSetVsmPolicy 21
Figure 7: The layout of HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\DeviceGuard 22
Figure 8: Function stack: Windows loader loads the hypervisor loader 23
Figure 9: Function stack: Image loading and verification 24
Figure 10: Guest virtual address of a hypercall page 25
Figure 11: The content of a hypercall page 26
Figure 12: Pseudo-code of the implementation of *OslFwProtectSecConfigVars* 26

Figure 13: Pseudo-code of the implementation of OslPrepareTarget invoking OslVsmSetup 27
Figure 14: Implementation of IumPostMailbox in IUMDLL.dll 29
Figure 15: Implementation of NtCreateUserProcess in NTDLL.dll 29
Figure 16: MSR 0xC0000082 storing KiSystemCall64 (secure kernel) 29
Figure 17: MSR 0xC0000082 storing KiSystemCall64 (normal kernel) 29
Figure 18: The implementation of KiSystemCall64 30
Figure 19: The implementation of SkiSecureServiceTable 30
Figure 20: The contents of a MSR 0x40000001 31
Figure 21: The contents of a hypercall page 32
Figure 22: Functions implementing hypercall functionalities 32
Figure 23: Issuing a VTL call 33
Figure 24: Contents of the VTL call data structure 34
Figure 25: Execution of the HvCallVtlReturn hypercall in SkCallNormalMode 34
Figure 26: ZwTerminateProcess invoking KiServiceInternal 35
Figure 27: Contents of IumSyscallDispEntries 35
Figure 28: Executing normal-mode services by functions with prefix Nk (NkTerminateProcess) 36
Figure 29: Invocation of VslpDispatchIumSyscall 37
Figure 30: VslpDispatchIumSyscall invoking the NtCreateEvent system call 37
Figure 31: The IumSyscallArgFcnTable array 38
Figure 32: A value of HvPartitionPropertyPrivilegeFlags 39
Figure 33: The workflow of mailbox-based data sharing 40
Figure 34: Pseudo-code of the implementation of IumSecureStorageGet 42
Figure 35: The layout of HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\DeviceGuard 48

Tables

Tabelle 1: Anforderungen für VSM Initialisierung 7
Tabelle 2: Überblick Bedrohungen und Mitigierungen 9
Tabelle 3: ETW Providers für VSM-bezogene Ereignisse 9
Table 4: Software requirements for VSM and its features (summarizing overview) 12
Table 5: Threats and mitigations (summarizing overview) 14
Table 6: ETW providers logging VSM-related events (summarizing overview) 14
Table 7: Threats and mitigations 45
Table 8: Software requirements for the core VSM entities and the VSM features 47
Table 9: Policy options for configuring HVCI and Credential Guard 48
Table 10: Values of registry keys for configuring HVCI and Credential Guard 49
Table 11: ETW providers logging VSM-related events 50
Table 12: Event IDs generated by the ETW provider Microsoft-Windows-IsolatedUserMode 50
Table 13: Event IDs generated by the ETW provider Microsoft-Windows-Wininit 50
Table 14: Event IDs generated by the ETW provider Microsoft-Windows-DeviceGuard 50

1 Introduction

1.1 Zusammenfassung

Dieses Kapitel stellt das Ergebnis von Arbeitspaket 6 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 des Virtual Secure Mode (VSM) des Microsoft Windows 10 Betriebssystems. Wie durch das BSI vorgegeben wird Windows 10 Build 1607, 64-bit, long-term servicing branch (LTSB), Deutsch betrachtet.

Die beschriebenen Analysen wurden dynamisch und statisch durchgeführt; jeweils unter Nutzung von `windbg` oder entsprechend `IDA` Dissassembler. Die technischen Beschreibungen umfassen Call Stacks von Funktionen und Pseudo Code. Die Call Stacks enthalten der Übersichtlichkeit wegen nur Funktionen, die für das Arbeitspaket relevant sind. Der Pseudo Code ist vom echten Code abstrahiert und nicht in einer bestimmten Programmiersprache definiert, basiert aber auf C/C++.

Die folgenden Abschnitte fassen die erarbeiteten Ergebnisse zusammen und verweisen auf die jeweiligen Kapitel für die ausführlichere Beschreibung.

Architekturüberblick (Kapitel 1.3) VSM ist eine Windows Technologie zur Erstellung und Verwaltung einer sicheren Windows-basierter Umgebung, die von der traditionellen Windows Umgebung isoliert ist. Diese sichere, isolierte Umgebung wurde entworfen um sicherheitskritische Funktionalität zu betreiben und diese vor Angriffen zu schützen, die von weniger vertrauenswürdigen oder exponierteren Komponenten ausgehen. Sicherheitskritische Funktionalität umfasst dabei die Speicherung sensibler Daten und die Ausführung kryptografischer Operationen.

VSM nutzt Hyper-V Virtualisierung als Basis für die implementierte Isolation. Hyper-V ist in den beiden Executables `%SystemRoot%\System32\hvix64.exe` und `%SystemRoot%\System32\hvax64.exe` implementiert. Der Hypervisor virtualisiert die System-Hardware und führt eine oder mehrere virtuelle Maschinen aus, die Partitionen genannt werden. Die sogenannte Root-Partition wird für die Verwaltung anderer Partitionen verwendet und bietet diesen Dienste an. Jede Partition wird innerhalb ihrer eigenen Grenzen ausgeführt; Hauptspeicher, Geräte und Central Processing Unit (CPU) sind zwischen den Partitionen durch den Hypervisor isoliert. Dadurch kann eine Partition nicht auf Ressourcen anderer Partitionen zugreifen.

In VSM Umgebungen bietet Hyper-V neben den genannten Partitionen den zusätzlichen Isolationsmechanismus Virtual Trust Level (VTL). Das Konzept der VTLs erzwingt Isolation in den folgenden Bereichen:

- Speicherzugriff: Der Speicherzugriff zwischen VTLs ist durch Zugriffskontrollmechanismen unterbunden. Speicherzugriffe über VTL-Grenzen hinweg sind damit nicht möglich.
- Virtuelle Prozessoren: Jeder virtuelle Prozessor hat einen Zustand pro VTL und jede VTL kann nur bestimmte virtuelle Prozessor-Register nutzen.
- Interrupts: Jede VTL hat ein dediziertes System für die Bearbeitung von Interrupts um Seiteneffekte über VTL Grenzen hinweg zu vermeiden.

Hyper-V implementiert zwei VTLs: VTL 0 und VTL 1. In VTL 0 wird die traditionelle Windows-Umgebung ([ERNW WP2], Kapitel 2.1) betrieben. In diesem Dokument wird diese Umgebung als *normal environment* und der darin ausgeführte Kernel als *normal kernel* bezeichnet. In VTL 1 wird die sichere, isolierte Windows-Umgebung für die Ausführung sicherheitskritischer Funktionalität betrieben. Diese Umgebung wird als *secure environment*, der darin betriebene Kernel als *secure kernel* bezeichnet. Die *secure environment* umfasst dabei:

- Einen Kernel und die benötigten Treiber: Der Kernel des *secure environments* wird in diesem Dokument als *secure kernel* bezeichnet. Die zugehörige ausführbare Datei ist `%SystemRoot%\System32\securekernel.exe`, die Treiber sind `%SystemRoot%\System32\skci.dll` und `%SystemRoot%\System32\cng.sys`. Der Kernel führt bestimmte sicherheitskritische Funktionen aus, wie beispielsweise kryptographische Operationen.
- Eine Userland-Umgebung: Das Laden und Ausführen eines Prozesses, welcher in dieser Umgebung ausgeführt wird, ist gesichert. Das bedeutet unter anderem gesicherte Inter-Prozess-Kommunikation (IPC) und verifizierbare Code Integrität. Die Umgebung ist als Isolated User Mode (IUM) bekannt, die darin ausgeführten Prozesse als Trustlets oder IUM Applikationen. Trustlets führen sicherheitskritische Funktionen aus, wie beispielsweise die Speicherung von Zugangsdaten. Eine typische IUM Applikation lädt die zentrale IUM Bibliothek `iumbase.dll` und die Bibliothek `iumcrypt.dll`, welche wiederum `iumdll.dll` laden. `iumdll.dll` implementiert die native IUM System Call Application Programming Interface (*API*), welche direkt den *secure kernel* anspricht. IUM Applikationen können ebenso traditionelle Windows-Bibliotheken wie kernel32.dll laden um Standard-Windowsfunktionalität zu nutzen. Diese Funktionalität wird nicht durch den *secure kernel* bereitgestellt sondern über entsprechende Aufrufe an den *normal kernel* durchgereicht.

Eine VSM Windows Umgebung besteht aus zentralen VSM Entitäten und weiteren VSM Features. Zentrale VSM Entitäten werden automatisch initialisiert und ausgeführt wenn Hyper-V aktiviert wird und umfassen: Das *normal environment*, den *secure kernel* und zugehörige Module (`skci.dll` und `cng.sys`) sowie die zentrale IUM-Bibliothek. VSM Features müssen dediziert konfiguriert werden und sind als IUM Applikationen oder als Treiber im *secure kernel* implementiert. Exemplarische Features sind beispielsweise Hypervisor Code Integrity (HVCI) und Credential Guard. HVCI bietet die Validierung von Code Integrität und ist Teil von `skci.dll` des *secure kernel*. Credential Guard verwaltet Zugangsdaten auf sichere Weise und ist im `lsaIso.exe` trustlet implementiert.

VSM Initialisierung: Prozess, Sicherheitsaspekte und Voraussetzungen (Kapitel 2.1 und Kapitel 3.1) Der Boot-Prozess einer VSM Windows Umgebung besteht aus den folgenden vier Phasen ([ERNW WP5], Kapitel 2.2):

- Phase 1: Der Boot Manager lädt den Windows Loader, der Windows Loader darauf den Hypervisor Loader.
- Phase 2: Der Hypervisor Loader lädt Hyper-V und über gibt die Bootkontrolle dann zurück an den Windows Loader;
- Phase 3: Der Windows Loader lädt den *secure kernel*;
- Phase 4: Der Windows Loader lädt den *normal kernel*. Danach booten sowohl *secure* als auch *normal kernel* in einen funktionsbereiten Zustand.

Jede Entität in der Prozesskette verifiziert die Integrität derjenigen Entität, die als nächstes geladen wird. Beispielsweise verifiziert der Hypervisor Loader die Integrität des Hyper-V Executables. Die Integrität wird via Authenticode verifiziert ([ERNW WP5], Kapitel 2.2); Boot Manager, Windows Loader, Hypervisor Loader und beide Kernel müssen von Microsoft signiert sein. Falls verfügbar wird die Vertrauenskette in die Unified Extensible Firmware Interface (*UEFI*) Firmware erweitert, die dann die Integrität des Boot Managers verifiziert.

Der Windows Loader (`%SystemRoot%\System32\winload.efi`) startet den VSM Initialisierungsprozess. Die Funktion `OslPrepareTarget` startet die Initialisierung basierend auf Konfigurationsparametern aus der Registry. Falls Secure Boot aktiv ist, werden die Parameter zusätzlich gegen *UEFI* Variablen verifiziert. Secure Boot ist keine Voraussetzung für die Initialisierung der zentralen *VSM* Entitäten, wird jedoch zwingend für *HVCI* und Credential Guard benötigt. Falls sich die Konfigurationsparameter aus der Registry von denen in den *UEFI* Variablen unterscheiden, werden die *UEFI* Variablen verwendet; Differenzen zwischen den Werten werden nicht geloggt.

Tabelle 1 enthält einen Überblick der Anforderungen für die Initialisierung zentraler VSM Entitäten sowie der VSM Features HVCI und Credential Guard:

Anforderung	**Für Initialisierung von...**
Hyper-V	Zentrale VSM Entitäten
Zentrale VSM Entitäten UEFI und Secure Boot	Credential Guard HVCI

Tabelle 1: Anforderungen für *VSM* Initialisierung

Die folgenden Hardware-Anforderungen gelten für Hyper-V: i) Second-level Address Translation (SLAT); ii) 64-bit CPU mit Virtualisierungsfunktionalität und No Execute(NX)-Bit; und iii) mindestens 4 Gigabyte (GB) Hauptspeicher [mic_hvreq]. Eine VSM-basierte Windows-Umgebung kann nicht ohne Hyper-V ausgeführt werden, da nur Hyper-V das Konzept von VTLs sowie die zugehörigen Kommunikationsschnittstellen (vgl. nächster Abschnitt) zwischen VTL 0 und VTL 1 bereitstellt.

Nach der erfolgreichen Initialisierung werden IUM Applikationen initialisiert. Dafür ruft der normal kernel die Funktion `NtCreateUserProcess` auf. Die Initialisierung einer Applikation als *IUM* Applikation ist an bestimmte Anforderungen gebunden, die durch externe Applikationen nicht erfüllt werden. Die Nutzung von allgemein nutzbaren Funktionen der Windows-*API* ([ERNW WP2], Kapitel 2.1), wie beispielsweise `CreateUserProcess`, resultieren nicht in *IUM* Applikationen. Für die *IUM*-Instanziierung muss ein bestimmtes Flag gesetzt sein, das nur durch den *normal kernel* für die Ausführung von *IUM* Applikationen gesetzt wird. Dieses Flag wird im vorliegenden Dokument als *IUM* Application Flag bezeichnet. Für die erfolgreiche *IUM*-Instanziierung müsste ein angepasster Loader implementiert werden, der direkt die Funktion `NtCreateUserProcess` aufruft, der das *IUM* Application Flag setzt. Weiterhin muss das zu instanziierende Executable von Microsoft mit Authenticode signiert sein.

VSM **Kommunikationsschnittstellen: Überblick und Sicherheitsaspekte** (Kapitel 2.2) Eine VSM-basierte Windows-Umgebung implementiert verschiedene Kommunikationsschnittstellen:

- IUM System Calls: Schnittstelle zwischen IUM Anwendungen und *secure kernel*, wobei der *secure kernel* Funktionalität für IUM Applikationen anbietet. Diese Schnittstelle kann nur durch IUM Anwendungen genutzt werden.;
- Normal-mode Dienste: Schnittstelle zwischen dem secure und dem normal kernel, wobei der normal kernel Dienste für den secure kernel anbietet. Diese Dienste bieten Kernel Funktionalität die nicht im secure kernel implementiert sind, aber für den Betrieb notwendig sind. Die angebotene Funktionalität umfasst Semaphoren und Prozess Verwaltung und Registry und Filesystem Ein-/Ausgabe. Für den Aufruf eines normal-mode Dienstes muss der Ausführungskontext von VTL 1 zu VTL 0 wechseln; dieser Vorgang wird VTL return genannt;
- Secure Dienste: Schnittstelle zwischen dem secure und dem normal kernel, wobei der secure kernel Dienste für den normal kernel anbietet. Diese Dienste implementieren sicherheitskritische Funktionalität die in der sicheren, isolierten Umgebung ausgeführt wird. Für die Nutzung dieser Funktionalität muss der Ausführungskontext von VTL 0 zu VTL 1 wechseln; dieser Vorgang wird VTL call genannt;
- Hypercalls: Schnittstelle zwischen den Kerneln und dem Hypervisor, wobei der Hypervisor Funktionalität für die Kernel anbietet.

Zusätzlich existiert die klassische, nicht- VSM-spezifische System Call Schnittstelle zwischen Nutzer-Applikationen und *normal kernel*. Windows 10 implementiert Sicherheitsmechanismen für alle genannten Schnittstellen:

- Einschränkung von VTL Calls: Hyper-V schränkt den Aufruf von VTL Calls auf validierte Eingabe-Werte ein, die aus dem höchst-privilegierten CPU-Ring stammen müssen ([Mic 2017], Section 15.6.1.1);

- Marshalling und Sanitization: Der secure kernel wendet Marshalling und Sanitization auf Ein- und Ausgabe-Daten von VTL returns an und stellt somit die Übergabe von Daten in definierten und überprüften Strukturen fest. Dieser Ansatz reduziert die erfolgreiche Ausnutzung von Implementierungsfehlern. Weiterhin zeigt dieser Ansatz, dass Daten aus der traditionellen Umgebung als nicht-vertrauenswürdig eingestuft werden;
- Hypercall Zugriffskontrolle: Hyper-V wendet Zugriffskontrollmechanismen auf die Ausführung von Hypercalls an, die sicherstellen, dass eine Partition die notwendigen Zugriffsrechte besitzt.
- IUM System Call Zugriffskontrolle: Die IUM System Calls `IumSecureStorageGet`, `IumSecureStoragePut`, `IumCreateSecureSection`, `IumGetDmaEnabler`, `IumOpenSecureSection` und `IumProtectSecureIo` können nur ausgeführt werden, wenn die aufrufende IUM Applikation die notwendigen Privilegien besitzt;
- Sicherer Datenaustausch zwischen Trustlets und der normalen Umgebung: Windows 10 implementiert Mailboxen für Trustlets für den Austausch mit der normalen Windows Umgebungen. Mailboxen erfordern eine schlüssel-basierte Authentifizierung.
- Sicherer Datenaustausch zwischen Trustlets: Windows 10 implementiert sicheren Binär-Storage für den Datenaustausch zwischen Trustlets. Der Binär-Storage erfordert ebenfalls schlüssel-basierte Authentifizierung.

Bedrohungen und Mitigierungen (Kapitel 2.3) Tabelle 2 enthält einen Überblick Mitigierungen einer VSM-basierten Windows-Umgebung. Die betrachteten Bedrohungen können in den folgenden Phasen auftreten: während der VSM Initialisierung; während dem VSM Betrieb; bei der Kommunikation zwischen Entitäten.

	Bedrohung	Mitigierung
VSM Initialisierung	Laden bösartiger Executables	*Integritätsvalidierung:* Die Integrität aller VSM-relevanter Executables wird via Authenticode validiert.
	Manipulation von VSM Konfiguration	*UEFI-basierte Validierung*: VSM Konfigurationsparameter werden zusätzlich in *UEFI* Variablen gespeichert. Beim Systemstart werden die *UEFI*-Variablen zur Verifizierung der Konfiguration verwendet.
VSM Betrieb	Unautorisierter Speicherzugriff auf secure environment	*Speicherisolation*: Hyper-V implementiert eine Trennung des Speichers zwischen normal und secure environment.
	Unautorisierter Zugriff auf Trustlet-Daten	*Mailboxen und sicherer Binär-Storage* (vergleiche Kapitel 2.2.5.5 und 2.2.5.6): Windows 10 implementiert Mailboxen und sicheren Binär-Storage für den Datenaustausch. Beide Mechanismen erfordern schlüssel-basierte Authentifizierung.
	Ausnutzung von Implementierungs- oder Design-Fehlern	*Minimierung der Angriffsfläche*: Der secure kernel implementiert ausschließlich sicherheitskritische Funktionalität. Alle anderen Kernel-Funktionen werden im normal kernel ausgeführt. *Marshalling und Hardening* (Kapitel 2.2.5.2): Der *secure kernel* strukturiert und überprüft sämtliche Daten im Austausch mit dem *normal environment*. *Sichere Software Entwicklung:* Es wurden keine offensichtlichen Verstöße oder Design-Fehler im Rahmen der Analyse identifiziert. Microsoft entwickelt Kern-Komponenten wie Kernel und Hypervisor

	Bedrohung	Mitigierung
		([Kelbley 2010], Kapitel 'Hypervisor Security') entsprechend dem Microsoft Software Development Lifecycle (SDL) [ms_sdl].
VSM Kommunikation	Unautorisierte Ausführung von Hypercalls	*Zugriffskontrolle*: Hyper-V erzwingt Zugriffskontrolle für den Aufruf von Hypercalls.
	Unautorisierte Ausführung von VTL calls	*Zugriffskontrolle*: Hyper-V schränkt die Ausführung von VTL calls auf berechtigte Komponenten ein.
	Unautorisierte Ausführung von IUM System Calls	*Zugriffskontrolle*: Der *secure kernel* implementiert Zugriffskontrolle für die Ausführung von IUM System Calls.

Tabelle 2: Überblick Bedrohungen und Mitigierungen

Konfiguration und Logging (Kapitel 3.2 und Kapitel 3.3) Die VSM Features HVCI und Credential Guard können via Gruppenrichtlinie unter dem folgenden Pfad konfiguriert werden: `Computer Configuration` → `Administrative Templates` → `System` → `DeviceGuard` → `Turn On Virtualization Based Security`. Es existieren drei relevante Einstellungen: `Select Platform Security Level`, `Virtualization Based Protection of Code Integrity` und `Credential Guard Configuration`. Diese Optionen erlauben unter anderem die Aktivierung und Deaktivierung von *HVCI* oder Credential Guard.

Die Werte der beschriebenen Gruppenrichtlinien werden in der Registry im folgenden Schlüssel gespeichert, was auch die direkte Konfiguration ermöglicht: `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\DeviceGuard`. Zusätzlich existieren die Schlüssel `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Lsa\LsaCfgFlags` und `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Lsa\RunasPPL` für die Konfiguration von Credential Guard.

Windows 10 nutzt das Event Tracing for Windows (ETW) Framework für das Logging VSM-bezogener Ereignisse. Tabelle 3 enthält die Namen und Globally Unique Identifiers (GUIDs) der ETW Provider:

ETW Provider	GUID
`Microsoft-Windows-IsolatedUserMode`	`73a33ab2-1966-4999-8add-868c41415269`
`Microsoft-Windows-Wininit`	`206f6dea-d3c5-4d10-bc72-989f03c8b84b`
`Microsoft-Windows-DeviceGuard`	`f717d024-f5b4-4f03-9ab9-331b2dc38ffb`

Tabelle 3: *ETW* Providers für *VSM*-bezogene Ereignisse

Hyper-V implementiert ebenfalls umfassende *ETW-bas*ierte Logging-Funktionalität. Geloggte Hyper-V Events können mit dem `Event Viewer` Werkzeug angezeigt werden [msblog_el].

Zusammenfassung Die Windows *VSM*-Technologie nutzt erfolgreich Virtualisierungstechnologie für die Isolation sicherheitskritischer Systemfunktionalität. *VSM* weist keine Design-Schwachstellen auf und mitigiert verschiedene Bedrohungen wie bspw. das Laden bösartiger Executables oder unautorisierter Kommunikation zwischen Trust-Leveln. Das *secure environment* wurde nach dem Prinzip Minimal Machine entworfen und weist nur eine minimierte Angriffsoberfläche auf. Die Nutzung von *VSM*-Funktionalität durch externe Parteien ist von Microsoft durch strikte Validierung von Signaturen sowie Einschränkung auf Signaturen von Microsoft unterbunden. Dies führt dazu, dass *VSM* eine vertrauenswürdige Ausführungsplattform bereitstellt.

1.2 Executive Summary

This chapter implements the work plan outlined in Work Package 6 of the project “SiSyPHuS Win10: Studie zu Systemaufbau, Protokollierung, Härtung und Sicherheitsfunktionen in Windows 10“ (orig., ger.). The project is 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 6 has been conducted by ERNW GmbH in the time period between November 2017 and May 2018, 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 virtual secure mode (VSM) feature of Windows 10. As required by the German Federal Office for Information Security, the release of the Windows 10 system in focus is build 1607, 64-bit, long-term servicing branch (LTSB), German language.

The analysis presented in this work was performed by applying static and dynamic code analysis methods using the `windbg` debugger and the `IDA` disassembler. The technical discussions in this work include depictions of function call stacks and pseudo-code. These call stacks, for the sake of brevity, present only functions that are relevant to the discussions. The depicted pseudo-code is a high abstraction of real code and does not consistently follow the syntax of a particular programming language. It loosely follows a C and C++-like programming language syntax.

The following paragraphs provide a summarizing overview of relevant analysis results. The referenced sections provide more details on the discussed topics.

Architecture overview (Section 1.3) VSM is a Windows technology for creating and managing a secure Windows-based environment. This environment is isolated from the traditional Windows environment. The secure, isolated environment is designed to host security-critical functionalities, protecting them from attacks from less trusted or more exposed components. Security-critical functionalities include storage of sensitive data and performing cryptographic operations.

VSM uses virtualization as a basis for isolation. The Hyper-V hypervisor provides virtualization features. Hyper-V is implemented in the `%SystemRoot%\System32\hvix64.exe` and `%SystemRoot%\System32\hvax64.exe` executables. Hyper-V virtualizes hardware and hosts one or multiple virtual machines, referred to as partitions. A partition known as the root partition is used for managing and providing services to other co-located partitions. Each partition operates within its own isolation boundary with regard to memory, devices, and central processing unit (CPU). Isolation boundaries between partitions are implemented by the hypervisor. The hypervisor allocates separate memory spaces and virtualize hardware resources to each partition. This means that a given partition cannot access the memory and hardware resources allocated to another partition.

In a VSM-enabled Windows environment, Hyper-V hosts the root partition. This partition hosts two kernel- and user-mode environments. Each kernel- and user-mode environment operates within an isolation domain, called virtual trust level (VTL). Hyper-V implements the VTLs. The concept of VTLs enforces isolation in multiple domains ([Mic 2017], Section 15.2.1):

- memory access: each VTL has a set of memory access protections associated with it. This prevents memory associated with a given VTL from being accessed by an entity operating in another VTL;
- virtual processor states: each virtual processor maintains a per-VTL state, where each VTL has a set of private virtual processor registers associated with it;
- interrupts: each VTL has a separate interrupt system for preventing interference in interrupt delivery and procession from entities operating in other VTLs.

Hyper-V implements two VTLs: VTL 0 and VTL 1. VTL 0 hosts the traditional Windows environment ([ERNW WP2], Section 2.1). This work refers to this environment as the normal environment, and to the kernel running in it as the normal kernel. VTL 1 hosts the secure, isolated Windows environment for

performing security-critical functionalities. This work refers to this environment as the secure environment. The secure environment consists of:

- a kernel and its required modules: The kernel of the secure environment is referred to as the secure kernel in this work. It is implemented in the `%SystemRoot%\System32\securekernel.exe` executable. The kernel's required modules are implemented in the `%SystemRoot%\System32\skci.dll` and `%SystemRoot%\System32\cng.sys` executables. This kernel performs a limited set of security-critical functionalities, such as cryptographic operations.
- a user-mode environment: The loading and execution of the processes running in this environment is secured. This includes communication over encrypted inter-process communication (IPC) and verifiable code integrity. This environment is known as the isolated user mode (IUM) and the processes running in it as IUM applications, or trustlets. Trustlets perform security-critical functionalities, such as credential storage.

  A typical IUM application loads the core IUM library implemented in the `iumbase.dll` and `iumcrypt.dll` library files. These, in turn, load the `iumdll.dll` file. This file implements the native IUM system call application programming interface (API) interacting directly with the secure kernel. An IUM application may load the traditional Windows libraries, such as `kernel32.dll`, implementing standard Windows functionalities. The secure kernel does not perform these functionalities itself, but relays the trustlets' invocations of the library functions implementing them to the normal kernel.

The architecture of a VSM-enabled Windows environment consists of core VSM entities and VSM features. Core VSM entities are initialized and executed once Hyper-V is enabled. These entities are: the normal environment, the secure kernel and its required modules (`skci.dll` and `cng.sys`), and the core *IUM* library. *VSM features are VSM entities that need to be* explicitly configured to operate. *VSM features* are implemented as *IUM applications or as part of modules of the secure kernel.* Example *VSM features are* hypervisor code integrity (*HVCI*) and Credential Guard. *HVCI provides code integrity* verification functionalities and is implemented as part of the `skci.dll` module of the secure kernel. Credential Guard manages user credentials in a secure way. It is implemented in the `lsaIso.exe` trustlet.

VSM initialization: Process, security aspects, and requirements (Section 2.1 and Section 3.1) The booting process of a VSM-enabled Windows environment consists of four phases ([ERNW WP5], Section 2.2):

- Phase 1: The boot manager loads the Windows loader. The Windows loader then loads the hypervisor loader;
- Phase 2: The hypervisor loader loads Hyper-V. Once Hyper-V is loaded, execution control is switched back to the Windows loader;
- Phase 3: The Windows loader loads the secure kernel;
- Phase 4: The Windows loader loads the normal kernel. The secure and the normal kernel load Windows 10 to its full extent, making it ready for use.

Each entity participating in this process verifies the integrity of the entity that it loads. For example, the hypervisor loader verifies the integrity of the executable implementing Hyper-V. Integrity is verified using the Authenticode digital signing technology ([ERNW WP5], Section 2.2). The boot manager, the Windows loader, the hypervisor loader, the normal kernel, and the secure kernel must be signed by Microsoft. If present, the Unified Extensible Firmware Interface (UEFI) firmware extends the trust chain securing the booting process of Windows by serving as the first root of trust in this chain. UEFI verifies the integrity of the boot manager.

The Windows loader, implemented in the `%SystemRoot%\System32\winload.efi` executable, starts the VSM initialization process. Its function `OslPrepareTarget` performs VSM initialization activities using *VSM* configuration parameters stored in the registry. Values of configuration parameters for initializing the core *VSM entities, HVCI, and Credential Guard are verified against configuration values stored as UEFI variables when Secure Boot is enabled.* Secure Boot is not a requirement for initializing the core *VSM entities* and is a requirement for initializing *HVCI and Credential Guard.* In case of a mismatch between

values stored in the registry and *UEFI variables, the values stored as UEFI variables are used for initialization;* mismatches are not logged.

Table 4 provides a summarizing overview of the software requirements for initializing the core VSM entities, and the VSM features HVCI and Credential Guard.

Requirement	Applies to
Hyper-V	Core VSM entities
Core VSM entities UEFI and Secure Boot	Credential Guard HVCI

Table 4: Software requirements for VSM and its features (summarizing overview)

The hardware requirements for VSM and its features apply only to the Hyper-V hypervisor. These are: i) the second-level address translation (SLAT) CPU features, ii) a 64-bit CPU with virtualization extensions, with hardware-assisted virtualization and the “no execute” (NX) bit set; that is, hardware-enforced data execution prevention (DEP) enabled; and iii) at least 4 GBs main memory [mic_hvreq]. A VSM-enabled Windows environment cannot operate without Hyper-V as Hyper-V implements the concept of VTLs and the communication interfaces (refer to next paragraph) between VTL 0 and VTL 1.

Once the normal and the secure kernel are loaded and executed, they load and execute IUM applications. For an IUM application to be loaded, the normal kernel invokes the `NtCreateUserProcess` function. Third parties cannot instantiate an application as an IUM application without fulfilling certain requirements. The use of functions implemented as part of the Windows API and exposed to third parties ([ERNW WP2], Section 2.1), such as `CreateUserProcess`, do not instantiate executables as IUM applications. This is because such functions do not set a concrete flag used by the normal kernel for instantiating an executable as an IUM application. This flag is referred to as the IUM application flag in this work. In order to instantiate an application as an IUM application, third parties need to implement custom executable loaders that directly invoke `NtCreateUserProcess` such that the IUM application flag is set. In addition, for an application to be instantiated as an IUM application, it has to be signed by Microsoft using the Authenticode digital signing technology.

VSM communication interfaces: Overview and security aspects (Section 2.2) A VSM-enabled Windows environment implements multiple communication interfaces:

- IUM system calls: Interface between IUM applications and the secure kernel, where the secure kernel provides services to IUM applications. Since only IUM applications may invoke IUM system calls, the conditions for instantiating IUM applications by third parties apply to invoking IUM system calls;
- normal-mode services: Interface between the secure and the normal kernel, where the normal kernel provides services to the secure kernel. These services implement kernel operations that are not implemented by the secure kernel, however, are necessary for this kernel, or the IUM applications that it hosts, to function. Example normal-mode services include semaphore and process management, and registry and filesystem input/output. For a normal-mode service to be invoked by the secure kernel, execution context has to be switched from VTL 1 to VTL 0. This process is known as VTL return;
- secure services: Interface between the secure and the normal kernel, where the secure kernel provides services to the normal kernel. These services implement security-critical kernel operations that are executed in the secure, isolated environment. For a secure service to be invoked by the normal kernel, execution context has to be switched from VTL 0 to VTL 1. This process is known as VTL call;
- hypercalls: Interface between the normal and the secure kernel, and the hypervisor, where the hypervisor provides services to the normal and the secure kernel.

In addition to the interfaces mentioned above, there is the traditional non-VSM-specific system call interface between user applications and the normal kernel.

Windows 10 implements mechanisms for securing the above communication interfaces:

- restrictions on issuing VTL calls: Hyper-V restricts the issuing of VTL calls. For a VTL call to be issued, among other things, it has to be initiated from the most privileged CPU mode and the input values have to be valid ([Mic 2017], Section 15.6.1.1);
- marshalling and sanitization: The secure kernel marshalls and sanitizes the input and output data of VTL returns. This is a security measure for checking, controlling, and managing the data coming in, and going out of, the secure kernel. This significantly reduces the risk of exploiting implementation or design errors involving the malicious manipulation of this data. It also indicates that data originating from the normal kernel is not explicitly trusted by the secure kernel;
- access control over hypercalls: The Hyper-V hypervisor enforces access control over hypercall execution. For a partition to execute a hypercall protected by access control, it has to possess the required privileges;
- access control over IUM system calls: The IUM system calls `IumSecureStorageGet`, `IumSecureStoragePut`, `IumCreateSecureSection`, `IumGetDmaEnabler`, `IumOpenSecureSection`, and `IumProtectSecureIo` are protected by access control. They can be executed only by IUM applications that possess the required privileges;
- secure data sharing between trustlets and entities running in the normal environment: Windows 10 implements mailboxes for trustlets to share data with entities running in the normal environment in a secure manner. Mailboxes secure shared data with key-based authentication;
- secure data sharing between trustlets: Windows 10 implements secure storage blobs for trustlets to share data between each other in a secure manner. Secure storage blobs secure shared data with key-based authentication.

Threats and mitigations (Section 2.3) Table 5 provides an overview of the capability of a VSM-enabled Windows environment to mitigate common threats. The threats may be realized during:

- the initialization process of VSM ('VSM initialization' in Table 5);
- the operation of the VSM-enabled Windows environment, once it has been initialized ('VSM operation' in Table 5);
- communication and exchange of data ('VSM communication' in Table 5).

	Threat	Mitigation(s)
VSM initialization	Loading of malicious executables	*Integrity verification*: The integrity of the executables implementing the core VSM entities and the VSM features is verified using the Authenticode digital signing technology.
	Setting malicious VSM configuration parameters at system start-up	*UEFI-based verification*: Values of configuration parameters for initializing VSM, which are stored in the registry, are verified at system start-up against configuration values stored as UEFI variables. The configuration parameters stored as *UEFI* variables are defined at the previous system shutdown and reflect the *VSM* configuration at that time. In case of a mismatch, the values stored as UEFI variables are used for initialization.
VSM operation	Unauthorized access to memory allocated to the secure environment	*Memory isolation*: Hyper-V enforces isolation between the memory allocated to the normal and the secure environment.
	Unauthorized access to shared trustlet data	*Mailboxes and secure storage blobs*: Windows 10 implements mailboxes and secure storage blobs for trustlets to share data in a secure manner. Mailboxes

	Threat	Mitigation(s)
		and secure storage blobs secure shared data with key-based authentication.
	Exploitation of implementation or design vulnerabilities of the virtualization and the secure environment	*Minimal exposure*: The secure kernel implements only security-critical functionalities. All other kernel functionalities are relayed to the normal kernel for execution. *Marshalling and hardening* (Section 2.2.5.2): The secure kernel marshalls and sanitizes data passed to, and returned from, the normal kernel. *Secure software development practices*: There are no obvious implementation or design flaws in the functionalities analyzed for the purpose of this work. Microsoft develops core system components, such as kernels and the Hyper-V hypervisor ([Kelbley 2010], Section 'Hypervisor Security') by following strict software development practices [ms_sdl].
VSM communication	Unauthorized invocation of hypercalls	*Access control*: Hyper-V enforces access control over hypercall execution.
	Unauthorized invocation of VTL calls	*Access control*: Hyper-V restricts the issuing of VTL calls.
	Unauthorized invocation of IUM system calls	*Access control*: The secure kernel implements access control over the execution of IUM system calls.

Table 5: Threats and mitigations (summarizing overview)

Configuration and logging capabilities (Section 3.2 and Section 3.3) Windows 10 comes with a group policy for configuring the VSM features *HVCI* and Credential Guard. This policy can be located with the `Group Policy Object Editor` utility, at the policy path `Computer Configuration` → `Administrative Templates` → `System` → `DeviceGuard` → `Turn On Virtualization Based Security`. Users may configure three policy options: `Select Platform Security Level`, `Virtualization Based Protection of Code Integrity`, and `Credential Guard Configuration`. Among other things, these options allow for users to enable or disable *HVCI* or Credential Guard.

The values of the above policy options are stored in the registry of Windows 10, at the key `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\DeviceGuard`. Therefore, this registry key can be used for configuring the *VSM* features. In addition, the registry keys `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Lsa\LsaCfgFlags` and `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Lsa\RunasPPL` are used for configuring Credential Guard.

Windows 10 uses the Event Tracing for Windows (ETW) framework for logging VSM-related events. Table 6 provides the names and the globally unique identifiers (GUIDs) of ETW providers logging such events.

ETW provider	GUID
`Microsoft-Windows-IsolatedUserMode`	`73a33ab2-1966-4999-8add-868c41415269`
`Microsoft-Windows-Wininit`	`206f6dea-d3c5-4d10-bc72-989f03c8b84b`
`Microsoft-Windows-DeviceGuard`	`f717d024-f5b4-4f03-9ab9-331b2dc38ffb`

Table 6: *ETW providers* logging *VSM*-related events (summarizing overview)

Hyper-V implements extensive *ETW-based logging functionality. Logged events that are related to the operation of Hyper-V can be viewed* with the `Event Viewer` utility [msblog_el].

Evaluation summary The *VSM* Windows technology successfully isolates security-critical system functionalities by leveraging virtualization technology. In general, *VSM* is well-designed. Its architecture and operating principles take into account and mitigate a variety of common threats, such as loading of malicious executables and unauthorized communication between the secure and the normal environment. The secure environment is designed with minimalism in mind, exposing a minimal attack surface. The secure environment communicates with the normal environment and the underlying hypervisor in a procedural manner. Usage of *VSM* functionalities by third parties is controlled and restricted by Microsoft through strict cryptographic requirements that are currently fulfilled only by implementations developed by Microsoft. This makes the *VSM*-enabled Windows environment a trustworthy computing platform.

1.3 Concepts and Terms

This section introduces concepts and terms relevant for understanding the contents of this work. Section 1.3.1 focuses on the virtualization technology implemented in Windows 10. Section 1.3.2 focuses on VSM. VSM is a Windows 10 technology for creating and managing a secure operating system environment, isolated from the traditional Windows environment ([ERNW WP2], Section 2.1). The secure isolated environment is designed to host security-critical functionalities, protecting them from attacks targeting the operating system [ms_vbsi]. VSM uses virtualization as a basis.

1.3.1 Virtualization

Figure 1 depicts the architecture of a virtualized Windows environment. The Hyper-V hypervisor virtualizes hardware and hosts one or multiple virtual machines, also known as partitions (`Partition A` and `Partition B` in Figure 1). The hypervisor provides virtualized hardware resources to each partition and manages these resources. This includes memory resources and virtual CPUs. The hypervisor is implemented in the `%SystemRoot%\System32\hvix64.exe` (for Intel-based platforms) and `%SystemRoot%\System32\hvax64.exe` (for AMD-based platforms) executables.

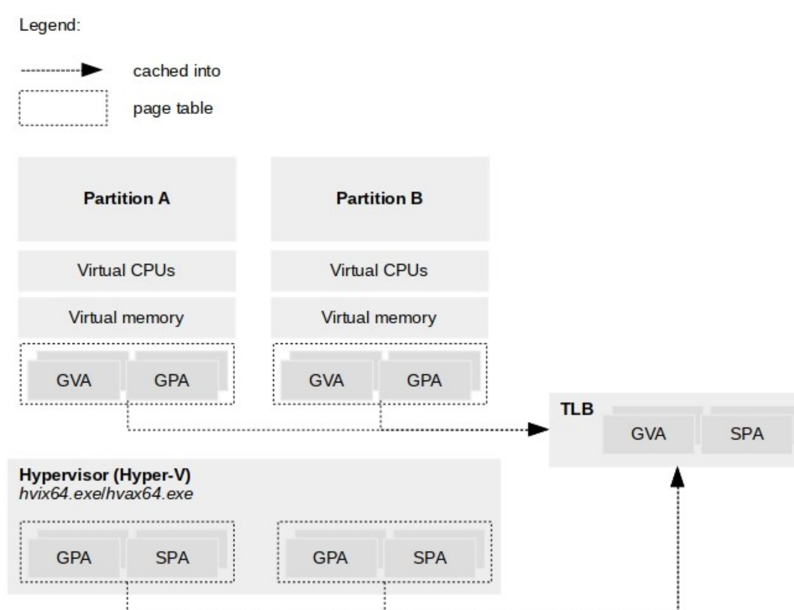

Figure 1: Architecture of a virtualized Windows environment

Each partition hosts an operating system environment. If Windows-based, this environment has an architecture consisting of the Windows parts: system support processes, services, applications, the Windows subsystem, `ntdll.dll`, drivers, kernel, and the hardware abstraction layer ([ERNW WP2], Section 2.1). Each partition operates within its own isolation boundary. Isolation boundaries between partitions are created and maintained by the hypervisor. Partition isolation boundaries are realized such that the hypervisor allocates separate memory spaces and virtualized hardware resources to each partition. This implies that a partition cannot access the memory allocated to another partition.

In a virtualized environment, based on Hyper-V, a partition known as the root partition is used for managing and providing services to other co-located partitions. For example, the root partition hosts virtualization services provided by the hypervisor and it provides these services to other co-located partitions. The root partition also hosts device drivers since it is the only partition that has direct access to hardware resources. There are three independent memory address spaces ([Mic 2017], Section 1.8):

- System physical address space: System physical addresses (SPAs, SPA in Figure 1) define the physical address space of the hardware memory resources as seen by the CPUs and the hypervisor. This space is known as the system physical space. There is only one system physical address space for the platform on which the virtualized Windows environment operates;
- Guest physical address space: Guest physical addresses (GPAs, GPA in Figure 1) define the physical address space as seen by a partition. This space is known as the guest physical space. The guest physical space is a virtualized abstraction of the system physical address space. This space is virtualized by the hypervisor, which can map GPAs to SPAs. There is one guest physical address space per partition;
- Guest virtual address space: Guest virtual addresses (GVAs, GVA in Figure 1) define the virtual address space as seen by a partition. This space is known as the guest virtual space. The guest virtual space is a virtualized abstraction of the guest physical space. There is one guest virtual address space per partition.

Translations between addresses that define different address spaces are performed using page tables (see Figure 1 and Figure 2). Page tables are constructs mapping addresses between different address spaces. They are used by the memory management units (MMUs) of CPUs and kernel routines performing memory operations.

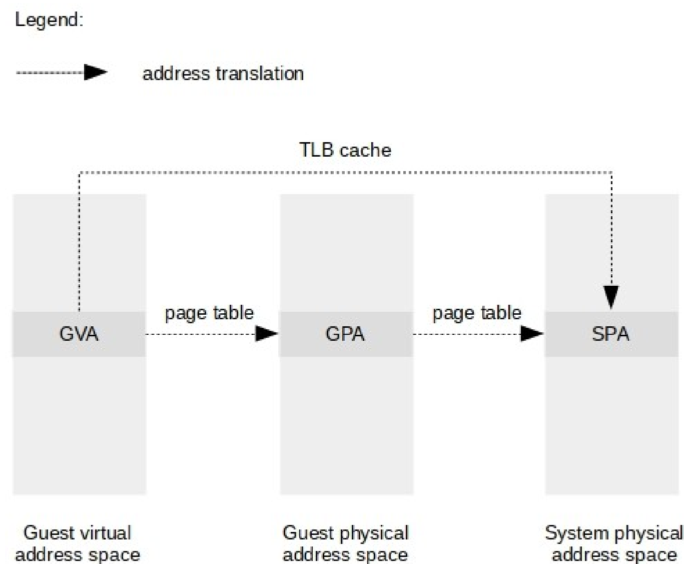

Figure 2: Address translation

Traditionally, each partition leverages software-implemented page tables for mapping GVAs to GPAs when referencing memory locations. These page tables are used by software-implemented MMUs of virtual CPUs and routines of the partitions' kernel performing memory operations. The GPAs are subsequently translated

to SPAs by the hypervisor. This is done by using page tables leveraged by the MMUs of hardware CPUs and routines of the hypervisor performing memory operations. The hypervisor maintains a copy of the page tables used by the partitions. This is because the hypervisor has to keep track of changes of these page tables made by the partitions so that it updates its page table accordingly. The copies of the partitions' page tables are software constructs.

The use and maintenance of software-implemented page tables is performance-expensive. This is because when a memory location is referenced, its address has to be translated twice – once using the software-implemented page tables and once using the page tables implemented in hardware. Page table updates are also expensive operations in terms of performance. Therefore, Hyper-V uses the SLAT CPU features. SLAT maps GVAs to SPAs and caches such mappings in the translation lookaside buffer (TLB in Figure 1, `TLB cache` in Figure 2). This is done by caching both GVA to GPA, and GPA to SPA mappings in the TLB. The use of the SLAT technology significantly speeds-up the use and management of page tables.

1.3.2 Virtual Secure Mode

Figure 3 depicts the architecture of a VSM-enabled Windows environment. Hyper-V hosts the root partition. This partition hosts two kernel- and user-mode environments. Each kernel- and user-mode environment operates within an isolation domain, referred to as VTL. The concept of VTLs enforces isolation at multiple aspects ([Mic 2017], Chapter 15), ([ERNW WP2], Section 3.3):

- memory access: each VTL has a set of memory access protections associated with it. This prevents memory associated with a given VTL from being accessed by an entity operating in another VTL;
- virtual processor states: each virtual processor maintains a per-VTL state, where each VTL has a set of private virtual processor registers associated with it;
- interrupts: each VTL has a separate interrupt system for preventing interference in interrupt delivery and procession from entities operating in other VTLs.

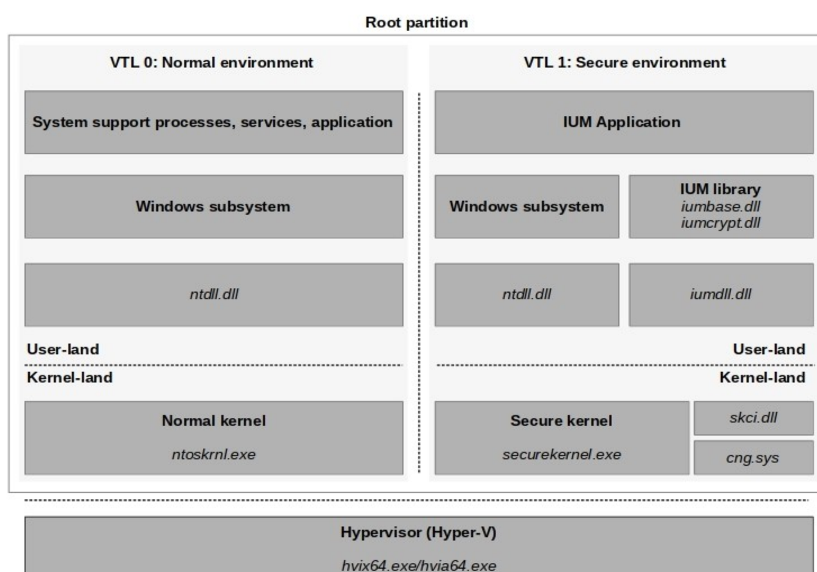

Figure 3: Architecture of a VSM-enabled Windows environment

The isolation described above is implemented and enforced by Hyper-V as the underlying entity managing the execution of the VSM-enabled Windows environment. At the time of the analysis, Hyper-V implements two VTLs: VTL 0 and VTL 1. VTL 0 hosts the traditional Windows environment consisting of the operating system parts: system support processes, services, applications, the Windows subsystem, `ntdll.dll`, drivers, and the kernel ([ERNW WP2], Section 2.1). This work refers to this environment as the normal environment and to the kernel running in it as the normal kernel.

VTL 1 hosts a Windows environment for performing security-critical functionalities. We refer to this environment as the secure environment. The secure environment consists of:

- a kernel and its required modules: The kernel running in the secure environment is referred to as the secure kernel in this work. It is implemented in the `%SystemRoot%\system32\securekernel.exe` executable. The kernel's required modules are implemented in the `%SystemRoot%\system32\skci.dll` and `%SystemRoot%\system32\cng.sys` executables. This kernel performs a limited set of security-critical functionalities, such as cryptographic operations.
- a user-mode environment: There are strict security requirements for the processes running in this environment. This includes encrypted IPC and verifiable code integrity. This environment is known as the IUM and the processes running in it as IUM applications, or trustlets. Trustlets perform security-critical functionalities, such as credential storage. Some trustlets are:
  - `lsaIso.exe`: This trustlet implements functionalities of the Local Security Authority (LSA) support process (`lsass.exe`, [ERNW WP2], Section 2.1). This process manages user authentication. When VSM is enabled, the local security authority process running in the normal environment does not perform the actual credential verification. This task is delegated to the secure, isolated counterpart of this process running in the secure environment – the IUM application `lsaIso.exe`. `lsass.exe` and `lsaIso.exe` communicate over encrypted IPC channels [ms_809132]. The isolation of the security-critical functionalities of the LSA prevents abuses of the local security authority for the purpose of accessing user credentials in an unauthorized manner. This includes abuses from a user with administrator privileges.
  - `BioIso.exe`: This trustlets implements security-critical functionalities of the Windows Hello biometrics service [mic_biom]. This service manages user authentication via biometric features. Similar to `lsass.exe`, the Windows Hello biometrics service delegates security-critical tasks to the IUM application `BioIso.exe`.

  The trustlets above are provided by Microsoft and are distributed with Windows 10.

  A typical IUM application loads the core IUM library implemented in the `iumbase.dll` and `iumcrypt.dll` library files. These, in turn, load the `iumdll.dll` file. The latter implements the native IUM system call API interacting directly with the secure kernel.

  An IUM application may load standard, traditional Windows libraries to use functionalities of the Windows system by invoking functions implemented in these libraries. For example, `lsaIso.exe` loads the Windows cryptography libraries implemented in the library files `bcrypt.dll`, `cryptsp.dll`, and `cryptdll.dll`. These load the native system service API implemented in the `kernelbase.dll` and `ntdll.dll` library files ([ERNW WP2], Section 2.1). Standard Windows functionalities are performed by the normal kernel. The secure kernel does not perform these functionalities itself, but relays the trustlets' invocations of the library functions implementing them to the normal kernel. Section 2.2 discusses this topic in more detail.

Figure 4 depicts the contents of a memory region beginning at the address `0x2779cbb0000`. This region is part of a memory dump of a VSM-enabled Windows environment and is mapped to the `lsaIso.exe` trustlet. The memory dump is a snapshot of the memory allocated to the root partition hosting the normal and the secure environment, generated after a controlled system crash was triggered. The question mark characters ('?') denote unreadable memory. The memory is unreadable since it cannot be accessed outside of the isolation boundaries implemented by VTL 1, in which `lsaIso.exe` operates. This demonstrates the VTL-based memory access protections enforced by Hyper-V.

The architecture of a VSM-enabled Windows environment consists of core VSM entities and VSM features. The core VSM entities are initialized and executed once Hyper-V is enabled (see Section 3.2). The core *VSM* entities are: the normal environment, the secure kernel and its required modules (`skci.dll` and `cng.sys`), and the core *IUM* library. As mentioned earlier, Hyper-V implements *VTL-based isolation between the normal environment and the other core VSM entities.*

```
kd> dc 2779cbb0000
00000277`9cbb0000  ???????? ???????? ???????? ????????  ????????????????
00000277`9cbb0010  ???????? ???????? ???????? ????????  ????????????????
00000277`9cbb0020  ???????? ???????? ???????? ????????  ????????????????
00000277`9cbb0030  ???????? ???????? ???????? ????????  ????????????????
00000277`9cbb0040  ???????? ???????? ???????? ????????  ????????????????
00000277`9cbb0050  ???????? ???????? ???????? ????????  ????????????????
00000277`9cbb0060  ???????? ???????? ???????? ????????  ????????????????
00000277`9cbb0070  ???????? ???????? ???????? ????????  ????????????????
```

Figure 4: Memory region mapped to lsaIso.exe

VSM features are *VSM entities that need to be* explicitly configured to operate. *VSM features* are implemented as *IUM applications or as part of modules of the secure kernel.* Sample *VSM features are HVCI* and Credential Guard. *HVCI provides code integrity verif*ication functionalities and is implemented as part of the `skci.dll` module of the secure kernel ([ERNW WP2], Section 3.4). Credential Guard manages user credentials and is implemented in the `lsaIso.exe` trustlet.

2 Technical Analysis of Functionalities

2.1 VSM Initialization

This section described the process for VSM initialization activities performed by the Windows loader ([ERNW WP5], Section 2.2) when Windows 10 is booted. The Windows loader is the Windows boot entity that initializes VSM with respect to values of configuration parameters.

Figure 5 depicts the booting process of Windows 10 with VSM enabled. In this process, each entity verifies the integrity of, and loads, the next entity in the booting chain. The UEFI firmware with Secure Boot enabled extends the trust chain securing the booting process of Windows. It serves as the first root of trust in this chain ([ERNW WP5], Section 2.2). The UEFI firmware verifies the integrity of and loads the boot manager. The following activities of the booting process can be structured into four phases:

- Phase 1: The boot manager loads the Windows loader. The Windows loader then loads the hypervisor loader (*[1]* in Figure 5);
- Phase 2: The hypervisor loader loads the Hyper-V hypervisor. Once Hyper-V is loaded, execution control is switched back to the Windows loader (*[2]* in Figure 5);
- Phase 3: The Windows loader loads the secure kernel (*[3]* in Figure 5);
- Phase 4: The Windows loader loads the normal kernel. The secure and the normal kernel load Windows 10 to its full extent, making it ready for use (*[4]* in Figure 5).

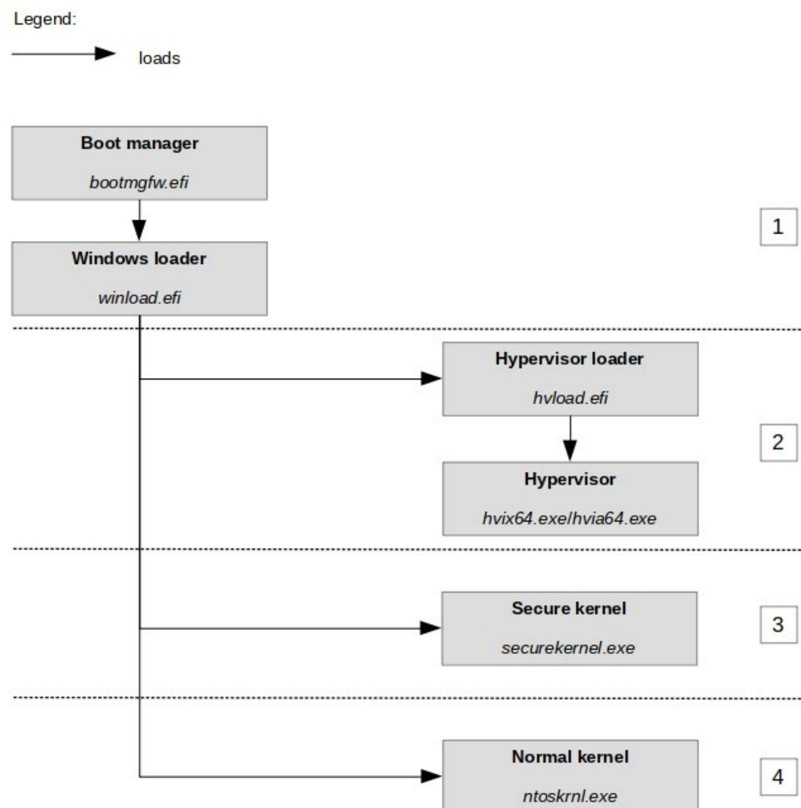

Figure 5: The booting process of Windows 10 with VSM enabled

The Windows loader starts the VSM initialization process. On an UEFI-enabled platform, the Windows loader is implemented in the `%SystemRoot%\System32\winload.efi` executable. The

`OslPrepareTarget` function implemented as part of the Windows loader performs VSM initialization activities. Section 2.1.1-Section 2.1.5 discuss such activities performed in relevant functions invoked by `OslPrepareTarget`: `OslSetVsmPolicy`, `OslArchHypervisorSetup`, `OslArchHypercallSetup`, `OslFwProtectSecureBootVariables`, and `OslVsmSetup`. These functions are invoked by `OslPrepareTarget` in the order as presented in this section. When these functions are finished executing, the Windows loader loads the secure and the normal kernel, which then instantiate IUM applications (see Section 1.3.2). Section 2.1.6 discusses the requirements for an executable to be instantiated as an IUM application.

2.1.1 OslSetVsmPolicy

`OslSetVsmPolicy` processes configuration parameters for enabling or disabling the core VSM entities and configuring the VSM features *HVCI* and Credential Guard (see Section 3.2). The parameters for configuring *HVCI* and Credential Guard are stored in the system's registry. Figure 6 depicts pseudo-code of the implementation of `OslSetVsmPolicy`.

```
OslSetVsmPolicy( [...] )
{

    if ( BlSecureBootGetBootPrivateVariable("VbsPolicyDisabled", [...] ))
    {
        if ( OslGetVsmEnabled( [...] ))
        {
            [...]
            OslHiveReadWriteControlDword("DeviceGuard", "RequirePlatformSecurityFeatures", [...] );

            OslHiveReadWriteControlDword("DeviceGuard", "Mandatory", [...]);

            OslGetVbsHvciConfiguration( [...] );

            OslHiveReadWriteControlDword("DeviceGuard", "RequireMicrosoftSignedBootChain", [...] );
            [...]

        }
        BlVsmSetSystemPolicy( [...] );
        [...]

    }
    [...]
}
```

Figure 6: Pseudo-code of the implementation of OslSetVsmPolicy

`OslSetVsmPolicy` first invokes `BlSecureBootGetBootPrivateVariable`. This function evaluates whether the UEFI variable `VbsPolicyDisabled` is defined. If defined, the core VSM entities and the VSM features will not be initialized.

If `VbsPolicyDisabled` is not set, `OslSetVsmPolicy` parses the registry key `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\DeviceGuard` (see Figure 7). The `OslGetVsmEnabled` function evaluates the values stored in the keys `Scenarios\HypervisorEnforcedCodeIntegrity\Enabled`, `Scenarios\HypervisorEnforcedCodeIntegrity\Locked`, `EnableVirtualizationBasedSecurity`, `Locked`, and `HyperVVirtualizationBasedSecurityOptOut`. The impact of specific values of these keys is provided in Section 3.2.[1]

1 When set, the HyperVVirtualizationBasedSecurityOptOut value enables virtual machines that run Windows with VSM enabled to disable VSM features at run-time [mic_setvsm].

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\DeviceGuard
├─ Scenarios
│   └─ HypervisorEnforcedCodeIntegrity
│         ├─ Enabled
│         ├─ Level
│         └─ Locked
├─ EnableVirtualizationBasedSecurity
├─ Locked
├─ HyperVVirtualizationBasedSecurityOptOut
├─ RequirePlatformSecurityFeatures
├─ Mandatory
└─ RequireMicrosoftSignedBootChain
```

Figure 7: The layout of HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\DeviceGuard

In addition to evaluating the above registry values, `OslGetVsmEnabled` evaluates the value of the `BcdOSLoaderInteger_HypervisorLaunchType` variable. This variable is stored in the system's *BCD*. It has to be set for the hypervisor to be loaded [mc_aa362670].

When `OslGetVsmEnabled` is finished executing, `OslSetVsmPolicy` processes further configuration information. This information is stored in the registry values `RequirePlatformSecurityFeatures`, `Mandatory`, and `RequireMicrosoftSignedBootChain` (see Figure 7 and the invocations of `OslHiveReadWriteControlDword` in Figure 6). In addition, `OslSetVsmPolicy` invokes `OslGetVbsHvciConfiguration`. This function evaluates the `HypervisorEnforcedCodeIntegrity` registry value. This value is used for initializing the *VSM feature HVCI*. Section 3.2 discusses the potential values of `HypervisorEnforcedCodeIntegrity\Locked` and `HypervisorEnforcedCodeIntegrity\Enabled`. The depiction of the layout of `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\DeviceGuard` in Section 3.2 shows less registry values than the one in Figure 7. This is because there are pre-defined values to use per default if there is no registry key defined; those pre-defined values are not part of the registry and thus not listed by `regedit.exe`.

Once `OslSetVsmPolicy` is finished processing the configuration parameters stored in the registry, it passes these parameters to the function `BlVsmSetSystemPolicy`. This function then evaluates the value stored in the `BcdOSLoaderInteger_SafeBoot` variable. This variable is stored in the system's *BCD*. It can have one of the following values: `SafemodeMinimal`, `SafemodeNetwork`, or `SafemodeDsRepair` [ms_aa362656]. The *VSM initialization* procedure continues only if `BcdOSLoaderInteger_SafeBoot` has the value of `SafemodeMinimal`.

`BlVsmSetSystemPolicy` then initializes the variable `BlVsmpSystemPolicy`. This variable stores the configuration parameters previously read from the registry in the form of flags that are part of a bitmask value. Before storing them in `BlVsmpSystemPolicy`, `BlVsmSetSystemPolicy` compares the configuration parameters with configuration parameters stored in the *UEFI* variable `VbsPolicy.` In case of a mismatch, the values stored in the *UEFI variable* are used for initializing the core *VSM entities and VSM features.* This indicates that *UEFI serves as the root of trust for evaluating* at system start-up the integrity of configuration parameters stored in the registry. The configuration parameters stored in `VbsPolicy` are defined at the previous system shutdown and reflect the *Vauch M* configuration at that time (see Section 3.2).

`VbsPolicy` is not defined when Secure Boot is disabled. In the scenario where the *VSM* features *HVCI* and Credential Guard are not configured to operate (see Section 3.2), a default set of configuration parameters are stored in `BlVsmpSystemPolicy`. These parameters are for initializing only the core *VSM* entities (see

Section 1.3.2). This indicates that Secure Boot is not a requirement for initializing these entities. Secure Boot is a requirement for initializing the *VSM features* Credential Guard and *HVCI*.

In the scenario where Credential Guard, and/or *HVCI are configured to be enabled*, and Secure Boot is not enabled, the core *VSM entities, HVCI, and/*or Credential Guard are not initialized.

Once populated with configuration parameters, `BlVsmpSystemPolicy` is stored in the `LOADER_PARAMETER_BLOCK` structure ([Russinovich 2012], Chapter 13). The Windows loader passes this structure to the normal and secure kernel when loaded.

2.1.2 OslArchHypervisorSetup

`OslArchHypervisorSetup` loads and executes the hypervisor loader. This section discusses the integrity verification process conducted as part of this activity. The integrity of the hypervisor loader executable is verified using the Authenticode digital signing technology. The hypervisor loader has to be signed by Microsoft. [ERNW WP5] provides a description of Authenticode signatures and how they are verified in the context of the boot manager, the Windows loader, and the Windows kernel. This section focuses on the cryptographic requirements that the digital Authenticode signature of the hypervisor loader must fulfill such that the loader is considered authentic.

On an *UEFI-enabled platform, t*he hypervisor loader is implemented in the `%SystemRoot%\System32\hvloader.efi` executable (i.e., image). The Windows loader loads this executable in the `ImgpLoadPEImage` function. Figure 8 (label 1) depicts the functions preceding `ImgpLoadPEImage`. Once the hypervisor loader is loaded, it is executed in the `Archpx64TransferTo64BitApplicationAsm` function. Figure 8 (label 2) depicts the functions preceding `Archpx64TransferTo64BitApplicationAsm`.

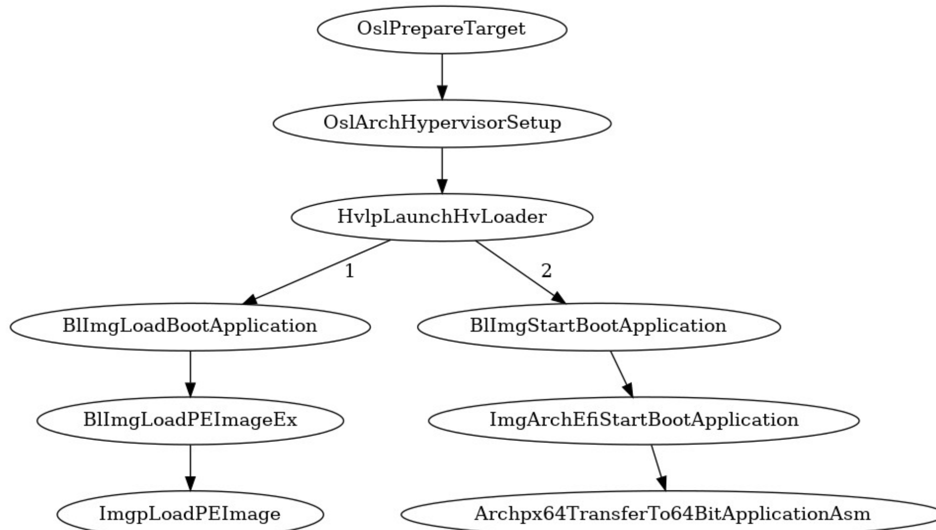

Figure 8: Function stack: Windows loader loads the hypervisor loader

`ImgpLoadPEImage` invokes three functions that are relevant to the image verification process: `ImgpGetScenarioFromImageFlags`, `ImgpGetHashAlgorithmForScenario`, and `ImgpValidateImageHash`.

Before the function `ImgpLoadPEImage` initiates the image verification process it must identify and store the signing requirements in configuration variables. This involves identifying and storing the required extended key usage (*EKUs*) that have to be present in the certificate of the signer of the image. It also involves identifying and storing the required hash algorithm used for signing certificates stored as part of the Authenticode signature.

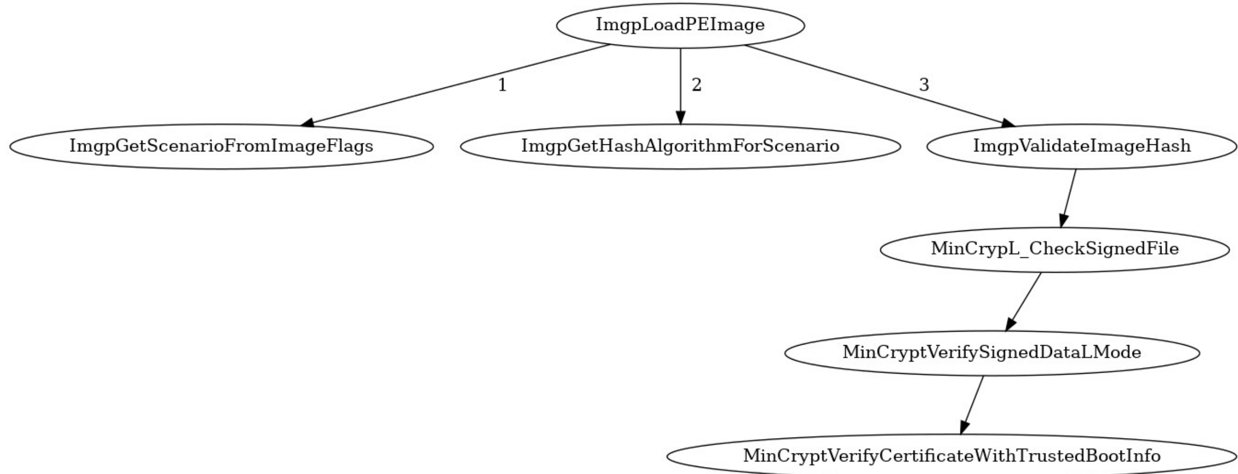

Figure 9: Function stack: Image loading and verification

`ImgpGetScenarioFromImageFlags` (see Figure 9, *[1]*) extracts a value indicating the required hash algorithm, such as Secure Hash Algorithm (SHA)-1, and stores it as an integer value. This work refers to this integer value as signing scenario. The signing scenario is stored as a flag that is part of a bitmask value. This work refers to this bitmask value as image loading parameters.

In addition to `ImgpGetScenarioFromImageFlags`, `ImgpLoadPEImage` extracts from the image loading parameters a value indicating the *EKUs* that have to be present in the certificate of the signer of the image. This work refers to this value as *EKU flag* and to the associated *EKUs as required signer's EKUs.* A certificate's *EKUs express the purposes for which the public key that is stored as part of certificate may be used. Code signing certificates issued by Microsoft contain specific EKUs that describe the use of the certificates' public keys for verifying signatures of specific executables. For example, the* `1.3.6.1.4.1.311.10.3.37` (`Isolated User Mode`) *EKU is stored in part of certificates used for verifying the signatures of IUM applications (see Section* 2.1.6).

The function `ImgpGetHashAlgorithmForScenario` (see Figure 9, [2]) maps the signing scenario extracted by `ImgpGetScenarioFromImageFlags` to a code identifying the required hash algorithm. This work refers to this code as hash code. For example, `ImgpGetScenarioFromImageFlags` extracts a signing scenario of `0` from the image loading parameters of the hypervisor loader. `ImgpGetHashAlgorithmForScenario` maps the signing scenario of `0` to the hash code `0x800c` if Secure Boot is enabled. This function maps the signing scenario of `0` to the hash code `0x8004` if Secure Boot is disabled. The hash codes `0x8004` and `0x800c` identify the SHA-1 and SHA-256 hash algorithms, respectively.

The `ImgpLoadPEImage` function initiates the integrity verification process by invoking the `ImgpValidateImageHash` function (see Figure 9, [3]). Based on the EKU flag, `ImgpValidateImageHash` stores the required signer's EKUs, extracted by `ImgpLoadPEImage`, in a variable. When the hypervisor loader is loaded, the required signer's EKU is `1.3.6.1.4.1.311.10.3.6`. In addition, it stores the EKU `1.3.6.1.5.5.7.3.3` in this variable. This is an *EKU that has to be present in all code signing certificates issued by Microsoft. The hypervisor loader is signed by Microsoft.*

The `MinCryptVerifyCertificateWithTrustedBootInfo` function (see Figure 9, [3]), invoked by `ImgpValidateImageHash`, verifies whether the required signer's *EKUs are present in the certificate* of the signer of the image being loaded. If a required *EKU is missing,* `MinCryptVerifyCertificateWithTrustedBootInfo` returns the error code `0xc0000428` (`INVALID_IMAGE_HASH`) [ms_cc704588]. The certificate of the signer of the image being loaded is also verified against its root certificate. This process is described in ([ERNW WP5], Section 2.2). The root certificate is hardcoded in the Windows loader executable. The fact that hardcoded contents of Windows loader executable are used for verification of the certificates of the signer of the hypervisor loader, shows that the root of trust for verifying the integrity of the loader is the Windows loader itself.

We analyzed the possibility for mitigating the previously described integrity verification process by modifying the system's boot configuration. This can be done by issuing the command `bcdedit /set nointegritychecks on` [mc_bcdedit]. The command sets the variable `BcdLibraryBoolean_DisableIntegrityChecks` to 1. This variable is stored in the system's *BCD*. We observed that when the hypervisor loader is loaded (see [1] in Figure 8), the value of `BcdLibraryBoolean_DisableIntegrityChecks` is evaluated, however, ignored. The integrity of the hypervisor loader is always verified.

Once the hypervisor loader is loaded, execution control is transferred to it by executing `Archpx64TransferTo64BitApplicationAsm` (see [2] in Figure 8). The hypervisor loader then loads the Hyper-V executable (`hvix64.exe` or `hvax64.exe`). The image integrity verification process implemented in the hypervisor loader is conceptually identical to the one implemented in the Windows loader. The hypervisor has to be signed by Microsoft. The integrity of the Hyper-V executable is verified using the Authenticode digital signing technology ([ERNW WP5], Section 2.2).

2.1.3 OslArchHypercallSetup

`OslArchHypercallSetup` maps a page aligned at a guest physical memory address to a guest virtual address (see Section 1.3.1). This page is allocated to the partition hosting the normal and secure kernel (see Section 1.3.2) and is populated by the hypervisor with code. This code is for the kernels to invoke hypervisor services, referred to as hypercalls. The page storing the code is referred to as the hypercall page. Section 2.2.2 discusses the topic of hypercalls in more detail.

`OslArchHypercallSetup` invokes `BlMmMapPhysicalAddressEx`. This function performs the mapping of the guest physical address, at which the hypercall page is aligned, to a guest virtual address. Figure 10 depicts the guest virtual address at which a hypercall page is aligned (`fffff802'206ea000` in Figure 10).

When `OslArchHypercallSetup` is finished executing, the guest virtual address of the hypercall page is stored in the `HvlpHypercallCodePageVa` variable. Once the hypercall page is populated by Hyper-V, `HvlpHypercallCodePageVa` is stored in the `LOADER_PARAMETER_BLOCK` structure ([Russinovich 2012], Chapter 13). This structure is passed to the normal and secure kernel when they are loaded and executed. Figure 11 depicts the content of a populated hypercall page, aligned at the virtual address `fffff802'206ea000`. It is extracted from the context of the normal kernel, once it has been loaded by the Windows loader. Section 2.2.2 discusses in detail the content of the hypercall page.

```
rcx=00000000`001c3d10
[...]
winload!BlMmMapPhysicalAddressEx:

kd> dps 00000000`001c3d10 L1
00000000`001c3d10  fffff802`206ea000
```

Figure 10: Guest virtual address of a hypercall page

```
0: kd> u poi(nt!HvlpHypercallCodeVa)
fffff802`206ea000 0f01c1          vmcall
fffff802`206ea003 c3              ret
fffff802`206ea004 8bc8            mov     ecx,eax
fffff802`206ea006 b811000000      mov     eax,11h
fffff802`206ea00b 0f01c1          vmcall
fffff802`206ea00e c3              ret
fffff802`206ea00f 488bc1          mov     rax,rcx
fffff802`206ea012 48c7c111000000  mov     rcx,11h
[...]
```

Figure 11: The content of a hypercall page

2.1.4 OslFwProtectSecureBootVariables

Among other parameters, `OslFwProtectSecureBootVariables` verifies configuration parameters of the *VSM feature Credential Guard. The verified parameters* are stored in the system's registry, and their values are verified against their counterparts stored as *UEFI variables*. In case of a mismatch, the values stored as *UEFI variables* are used for initialization of Credential Guard. The configuration parameters stored as *UEFI* variables are written into the *UEFI* context at the first system shutdown after enabling VSM and reflect the Credential Guard configuration at that time (see Section 3.2). *UEFI serves as the root of trust for evaluating* at system start-up the integrity of configuration parameters of Credential Guard stored in the registry. Secure Boot is a requirement for Credential Guard. If Secure Boot is not enabled, Credential Guard is not initialized (see Section 2.1.1).

`OslFwProtectSecureBootVariables` invokes the `OslFwProtectSecConfigVars` function. Figure 12 depicts pseudo-code of the implementation of `OslFwProtectSecConfigVars`. `OslFwProtectSecConfigVars` evaluates the values of the registry keys `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\LSA\RunasPPL` and `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\LSA\LsaCfgFlags` against their counterparts stored as *UEFI variables*.

```
OslFwProtectSecConfigVars( [...] )
{
    [...]

    OslHiveReadWriteControlDword( "Lsa", "RunasPPL", [...] );
    [...]

    EfiGetVariable( "Kernel_Lsa_Ppl_Config", [...] );
    [...]

    if ( BlSecureBootGetBootPrivateVariable("Kernel_Lsa_Cfg_Flags_Cleared", [...]) )
    {
        [...]

        if ( OslHiveReadWriteControlDword( [...], "Lsa", "LsaCfgFlags", &getRegValue ) )
        {

            if ( EfiGetVariable( "Kernel_Lsa_Cfg_Flags", &getEfiValue, [...] ))
            {
                [...]
            }
            else
            {
                [...]
                if ( ( getRegValue | getEfiValue ) == getEfiValue )
                [...]
            }
            [...]
        }
    }
    [...]
}
```

Figure 12: Pseudo-code of the implementation of OslFwProtectSecConfigVars

OslFwProtectSecConfigVars first evaluates the value stored in the key RunasPPL (OslHiveReadWriteControlDword and EfiGetVariable in Figure 12). RunasPPL configures the *LSA process* to be executed under Protected Process Light (*PPL*) protection [ms_runasppl]. *PPL is a security mechanism* protecting the memory space of a process from accesses by other untrusted processes ([ERNW WP2], Section 2.4.7). If the RunasPPL registry key is set and Secure Boot is enabled, the OslFwProtectSecConfigVars compares the value read from the registry with the *UEFI variable* Kernel_Lsa_Ppl_Config.

OslFwProtectSecConfigVars then evaluates the value stored in the key LsaCfgFlags [ms_lsacfgflag]. Among other things, the LsaCfgFlags registry key indicates if Credential Guard should be enabled. The value stored in the LsaCfgFlags registry key (OslHiveReadWriteControlDword and getRegValue in Figure 12) is compared with the value stored in the *UEFI* variable Kernel_Lsa_Cfg_Flags (EfiGetVariable and getEfiValue in Figure 12). The comparison is done with a logical OR operation (getRegValue | getEfiValue in Figure 12). In addition, OslFwProtectSecConfigVars evaluates the value of the *UEFI variable* Kernel_Lsa_Cfg_Flags_Cleared. If this variable is set, the configuration parameters of Credential Guard will be cleared and Credential Guard will not be initialized.

2.1.5 OslVsmSetup

Figure 13 depicts pseudo-code of the implementation of OslPrepareTarget invoking the OslVsmSetup function. The primary task of OslVsmSetup is to load and execute the secure kernel and its modules. After the secure kernel is loaded, the Windows loader loads the normal kernel. The secure and the normal kernel then load Windows 10 to its full extent, making it ready for use (see Figure 5 and [ERNW WP5], Section 2.2.2).

```
[...]
if ( !_bittest(&BlVsmpSystemPolicy, 0xBu)
    && (HvlpResourceInitialized || HviIsHypervisorMicrosoftCompatible( [...] )) )
{
    OslVsmSetup( [...] );
}
[...]
```

Figure 13: Pseudo-code of the implementation of OslPrepareTarget invoking OslVsmSetup

The secure kernel is implemented in the %SystemRoot%\system32\securekernel.exe executable. The secure kernel is loaded if the 11th bit in the BlVsmpSystemPolicy variable is not set (bittest and 0xBu in Figure 13, see Section 2.1.1). In addition, the HvlpResourceInitialized variable needs to be set. HvlpResourceInitialized is set only when the hypervisor loader has loaded the Hyper-V hypervisor. If this variable is not set, for the secure kernel to be loaded, the presence of Hyper-V has to be determined. Hyper-V sets the 31-st bit of the value stored in the ECX register ([Mic 2017], Section 2.2). The HviIsHypervisorMicrosoftCompatible function verifies that this bit is set.

OslVsmSetup invokes the OslpVsmLoadModules and OslLoadImage functions to load and verify the integrity of the securekernel.exe as well as its required modules (see Section 1.3.2). The integrity of securekernel.exe and its required modules is verified using the Authenticode digital signing technology ([ERNW WP5], Section 2.2). The secure kernel has to be signed by Microsoft. The verification of cryptographic requirements, such as *EKUs* that have to be present in the certificate of the signer of the image, is conceptually identical to the one described in Section 2.1.2.

Before OslpVsmLoadModules is invoked, OslVsmSetup evaluates the value stored in the BCDE_OSLOADER_TYPE_VSM_LAUNCH_TYPE variable. This variable is stored in the system's *BCD*. It can have the value Off or Auto. OslpVsmLoadModules is invoked only if BCDE_OSLOADER_TYPE_VSM_LAUNCH_TYPE has the value of Auto.

2.1.6 Instantiation of IUM Applications

Once the normal and the secure kernel are loaded and executed, they instantiate IUM applications (i.e., trustlets). For an IUM application to be instantiated, the normal kernel invokes the `NtCreateUserProcess` function. Among other things, this function initializes relevant kernel structures for process management, such as the structure of type `EPROCESS`. This structure contains relevant process information, such as process ID ([ERNW WP2], Section 4.1). `NtCreateUserProcess` collaborates with the secure kernel to finish instantiating the IUM application.

Third parties cannot instantiate an application as a trustlet without fulfilling certain requirements. The use of functions implemented as part of the Windows API and exposed to third parties ([ERNW WP2], Section 2.1), such as `CreateUserProcess`, do not instantiate executables as IUM applications. This is because such functions do not set a concrete flag used by the normal kernel for instantiating an executable as an IUM application. This flag is referred to as the IUM application flag in this work. In order to instantiate an application as an IUM application, third parties need to implement custom executable loaders that directly invoke `NtCreateUserProcess` such that the IUM application flag is set.

In addition to the IUM application flag being set, for an application to be instantiated as a trustlet, it has to be properly signed. Each executable implementing an IUM application has to be signed by Microsoft using the Authenticode digital signing technology ([ERNW WP5], Section 2.2). The certificate issued by the signer of the IUM application has to possess the following EKUs (textual EKU descriptions provided in round brackets, see Section 2.1.2):

- `1.3.6.1.5.5.7.3.3` (`Code Signing`)
- `1.3.6.1.4.1.311.10.3.6` (`Windows System Component Verification`)
- `1.3.6.1.4.1.311.10.3.37` (`Isolated User Mode`)
- `1.3.6.1.4.1.311.10.3.24` (`Protected Process Verification`)

The EKU `Isolated User Mode` is used specifically for marking certificates that can be used only for verifying signatures of executables implementing IUM applications. As part of the Authenticode signature verification process of a given IUM application, the EKUs stored in the certificate used for signing the application are evaluated against *E*KUs hardcoded in the `ci.dll` library file. In case of a mismatch, the application is considered not authentic.

2.2 Communication Interfaces

A VSM-enabled Windows environment implements multiple communication interfaces:

- IUM system calls: Interface between IUM applications and the secure kernel, where the secure kernel provides services to IUM applications (Section 2.2.1);
- normal-mode services: Interface between the secure and the normal kernel, where the normal kernel provides services to the secure kernel (Section 2.2.4);
- secure services: Interface between the secure and the normal kernel, where the secure kernel provides services to the normal kernel (Section 2.2.3); and
- hypercalls: Interface between the normal and the secure kernel, and the hypervisor, where the hypervisor provides services to the normal and the secure kernel (Section 2.2.2).

In addition to the interfaces mentioned above, there is the traditional non-VSM-specific system call interface enabling communication between user applications and the normal kernel. ([ERNW WP2], Section 2.1) discusses this interface. Therefore, the detailed analysis of it is out of the scope of this work package. Section 2.2.1 focuses on the execution path of IUM system calls, comparing it with that of traditional system calls.

2.2.1 IUM System Calls

IUM system calls implement services that the secure kernel exposes to IUM applications. This includes critical system services enabling the operation of IUM applications, such as memory management services.

The execution path of IUM system calls is conceptually identical to that of traditional system calls (see Figure 4, [ERNW WP2]). An IUM application executes IUM system calls by invoking functions implemented in the `IUMDLL.dll` library file (see Section 1.3.2). These functions have names starting with `Ium` and implement execution context switching between IUM applications and the secure kernel. Figure 14 depicts the implementation of the `IumPostMailbox` IUM system call in `IUMDLL.dll`. For comparison purposes, Figure 15 depicts the implementation of the traditional system call `NtCreateUserProcess` in the `NTDLL.dll` library file.

Same as traditional system calls, each IUM system call can be uniquely identified by a system service index. Indexes specifying IUM system calls have the highest bit set. An example is `0x800000A`, a system service index specifying the `IumPostMailbox` IUM system call (see line 3 in Figure 14). Once the system service index is set, the `syscall` instruction switches the execution context to the secure kernel (see line 4 in Figure 14 and line 8 in Figure 15).

```
1   iumdll!IumPostMailbox:
2   mov      r10,rcx
3   mov      eax,800000Ah
4   syscall
5   ret
```

Figure 14: Implementation of IumPostMailbox in IUMDLL.dll

```
1   ntdll!NtCreateUserProcess:
2   mov      r10,rcx
3   mov      eax,0C1h
4   test     byte ptr [SharedUserData+0x308],1
5   [...]
6
7   ntdll!NtCreateUserProcess+0x12:
8   syscall
9   ret
10  [...]
```

Figure 15: Implementation of NtCreateUserProcess in NTDLL.dll

Once the execution context is switched to the secure kernel, it invokes the `KiSystemCall64` routine. The address of this function is stored in the model-specific register (MSR) `0xC0000082` when the `ShvlpInitProcessor` function is invoked (see Figure 16). This function is invoked during the initialization of the secure kernel. For comparison purposes, Figure 17 depicts the use of the MSR `0xC0000082` for the same purpose in the context of traditional system calls. The `KiInitializeBootStructures` is invoked during the initialization of the normal kernel.

```
ShvlpInitProcessor( [...] )
{
    [...]
    __writemsr(0xC0000082, KiSystemCall64);
    [...]
}
```

Figure 16: MSR 0xC0000082 storing KiSystemCall64 (secure kernel)

```
KiInitializeBootStructures( [...] )
{
    [...]
    __writemsr(0xC0000082, KiSystemCall64);
    [...]
}
```

Figure 17: MSR 0xC0000082 storing KiSystemCall64 (normal kernel)

`KiSystemCall64` routes invocations of a given IUM system call to the corresponding handler function implementing the actual service functionality. Figure 18 depicts the implementation of `KiSystemCall64`. This function first evaluates whether the highest bit of the system service index is set (see line 5 in Figure 18). If set, `KiSystemCall64` loads the address of the kernel structure `SkiSecureServiceTable` (see line 13 in Figure 18). This structure is an array of functions implementing the service functionalities of IUM system calls. Figure 19 depicts the implementation of `SkiSecureServiceTable`. After loading

`SkiSecureServiceTable`, `KiSystemCall64` executes the handler function indexed by the system service index identifying the invoked IUM system call (see line 22 and line 33 in Figure 18).

If the highest bit of the system service index is not set, that is, if the evaluation at line 5 in Figure 18 fails, the `KiSystemCall64` routes an invocation of a normal-mode service. Section 2.2.4 discusses normal-mode services.

Since only IUM applications may invoke IUM system calls, the conditions for instantiating IUM applications by third parties (see Section 2.1.6) apply also to invoking IUM system calls by these parties.

```
1   securekernel!KiSystemCall64:
2   [...]
3   mov     ebx,eax
4   and     eax,0FFFh
5   test    ebx,8000000h
6   je      securekernel!KiSystemServiceStart+0x28
7
8   securekernel!KiSystemServiceStart+0x13:
9   cmp     eax,dword ptr [securekernel!SkiSecureServiceLimit]
10  jae     securekernel!KiSystemServiceExit+0x145
11
12  securekernel!KiSystemServiceStart+0x1f:
13  lea     r10,[securekernel!SkiSecureServiceTable]
14  jmp     securekernel!KiSystemServiceStart+0x3b
15
16  securekernel!KiSystemServiceStart+0x28:
17  lea     r10,[securekernel!IumSyscallDispatchTable]
18  cmp     eax,dword ptr [securekernel!IumSyscallDescriptorLimit]
19  jae     securekernel!KiSystemServiceExit+0x145
20
21  securekernel!KiSystemServiceStart+0x3b:
22  movsxd  r11,dword ptr [r10+rax*4]
23  mov     rax,r11
24  sar     r11,4
25  add     r10,r11
26  nop     dword ptr [rax]
27  and     eax,0Fh
28  sub     eax,4
29  jle     securekernel!KiSystemServiceCopyEnd
30  [...]
31  securekernel!KiSystemServiceCopyEnd:
32  mov     eax,ebx
33  call    r10
34  [...]
```

Figure 18: The implementation of KiSystemCall64

```
SkiSecureServiceTable dq offset IumCreateSecureDevice
                                        ; DATA XREF: SkiSystemStartup+D6?o
                                        ; KiSystemCall64+C3?o
                dq offset IumCreateSecureSection
                dq offset IumCrypto
                dq offset IumDmaMapMemory
                dq offset IumFlushSecureSectionBuffers
                dq offset IumGetDmaEnabler
                dq offset IumGetExposedSecureSection
                dq offset IumGetIdk
                dq offset IumMapSecureIo
                dq offset IumOpenSecureSection
                dq offset IumPostMailbox
                dq offset IumProtectSecureIo
                dq offset IumQuerySecureDeviceInformation
                dq offset IumSecureStorageGet
                dq offset IumSecureStoragePut
                dq offset IumUnmapSecureIo
                dq offset IumUpdateSecureDeviceState
```

Figure 19: The implementation of SkiSecureServiceTable

2.2.2 Hypercalls

Hypercalls implement services that the hypervisor exposes to partitions. This involves critical system services enabling the operation of virtual systems, such as memory management services. The hypercalls implemented by the Hyper-V hypervisor are listed in ([Mic 2017], Appendix A). Each Hyper-V hypercall can be uniquely identified by an identification number, referred to as a call code.

Partitions can invoke hypercalls only from kernel-mode. In a VSM-enabled Windows environment, this includes the execution context of the normal and the secure kernel. The `winhv` and `winhvr` drivers implement wrapper functions enabling the straightforward invocation of hypercalls. For example, the functions implement assignment of call codes and management of hypercall input and output values. The activities that need to be performed for a Hyper-V hypercall to be executed are documented in ([Mic 2017], Section 3).

A crucial prerequisite for Hyper-V hypercalls to be invoked is the existence of the hypercall page in the context of the partition. A hypercall page is a memory page that stores code for invoking hypercalls as per the Hyper-V specification. This page is exposed by the hypervisor to each partition. During the initialization process, each partition reserves a memory page and stores its GPA in the MSR `0x40000001` ([Mic 2017], Section 3.13). Hyper-V then populates this page with code. A populated hypercall page cannot be modified in order to prevent unauthorized modifications of the code stored in it. Figure 20 depicts the contents of a MSR `0x40000001`.

```
[...]

kd> rdmsr 0x40000001
msr[40000001] = 00000000`0020e003

[...]
```

Figure 20: The contents of a MSR 0x40000001

When the hypercall page is loaded in the context of a partition, the kernel running in the partition can invoke hypercalls. This typically involves activities such as loading the hypercall page, allocating memory buffers for hypercall input and output values, and setting these values. Finally, the code stored in the hypercall page is executed so that the execution context is switched to the hypervisor. For example, the `WinHvpHypercall` function of the `winhvr` driver results in the execution of code stored in the hypercall page.

Figure 21 depicts the contents of a hypercall page accessed in the `WinHvpHypercall` function. It contains page-aligned code for invoking hypercalls, padded with “no operation” (`nop`) instructions. The page contains several sets of instructions ending with the instruction sequence “`vmcall ret`”. The `vmcall` instruction is implemented in Intel processors and it switches execution context to the hypervisor.

The sets of instructions stored in the hypercall page can be understood as trampolines for abstracting the switching of execution context to the hypervisor in different scenarios. These trampolines accommodate the execution of any hypercall, and of the hypercalls with call codes `0x11` and `0x12`, on both 32-bit and 64-bit platforms. The instructions preceding the `vmcall` instructions (if any) save the contents of the `eax`, or the `rcx`, register and store a hypercall call code in this register. The `eax`, or the `rcx`, register stores a hypercall call code on 32-bit and 64-bit platforms, respectively. The use of specific registers for storing hypercall input and output values, as well as call codes, is documented in ([Mic 2017], Section 3.7) and ([Mic 2017], Section 3.8).

The sets of instructions in the hypercall page where the values `0x11` and `0x12` are stored in the `eax`, or the `rcx`, register are used for invoking the hypercalls with call codes `0x11` and `0x12`. These hypercalls are used for invoking normal-mode and secure services. Section 2.2.3 and Section 2.2.4 discuss these services. The first

set of instructions, containing only the "`vmcall ret`" instruction sequence, is used for invoking any other hypercall.

```
[...]

kd> u fffff800`b3948000 L20
fffff800`b3948000 0f01c1          vmcall
fffff800`b3948003 c3              ret
fffff800`b3948004 8bc8            mov     ecx,eax
fffff800`b3948006 b811000000      mov     eax,11h
fffff800`b394800b 0f01c1          vmcall
fffff800`b394800e c3              ret
fffff800`b394800f 488bc1          mov     rax,rcx
fffff800`b3948012 48c7c111000000  mov     rcx,11h
fffff800`b3948019 0f01c1          vmcall
fffff800`b394801c c3              ret
fffff800`b394801d 8bc8            mov     ecx,eax
fffff800`b394801f b812000000      mov     eax,12h
fffff800`b3948024 0f01c1          vmcall
fffff800`b3948027 c3              ret
fffff800`b3948028 488bc1          mov     rax,rcx
fffff800`b394802b 48c7c112000000  mov     rcx,12h
fffff800`b3948032 0f01c1          vmcall
fffff800`b3948035 c3              ret
fffff800`b3948036 90              nop
fffff800`b3948037 90              nop
fffff800`b3948038 90              nop

[...]
```

Figure 21: The contents of a hypercall page

When the `vmcall` instruction is executed, the execution context is switched to Hyper-V. The hypervisor then performs access control checks. If a given hypercall is protected by access control, the partition invoking it has to possess the required privilege ([Mic 2017], Section 3.11). Section 2.2.5.3 discusses access control over hypercalls. If the hypercall is to be executed, Hyper-V loads an array that contains entries of a fixed size. Each entry is indexed by a hypercall call code and contains a pointer to a function implementing the functionality of the hypercall identified by the call code. Hyper-V then executes the function indexed by the call code of the invoked hypercall. After this, the execution context is switched back to the kernel that has invoked the hypercall. Figure 22 depicts a portion of the array containing functions implementing hypercall functionalities indexed by call codes (see, for example, `sub_FFFFF800002129D8` [function] and `5Dh` [call code] in Figure 22). This array is implemented as part of `hvix64.exe`.

```
[...]

CONST:FFFFF80000C008B8          dq offset sub_FFFFF800002129D8
CONST:FFFFF80000C008C0          db  5Dh ; ]
CONST:FFFFF80000C008C1          db    0
CONST:FFFFF80000C008C2          db    0
CONST:FFFFF80000C008C3          db    0
CONST:FFFFF80000C008C4          db    8
CONST:FFFFF80000C008C5          db    0

[...]

CONST:FFFFF80000C008CF          db    0
CONST:FFFFF80000C008D0          dq offset sub_FFFFF800002BF034
CONST:FFFFF80000C008D8          db  5Eh ; ^
CONST:FFFFF80000C008D9          db    0
CONST:FFFFF80000C008DA          db    0
CONST:FFFFF80000C008DB          db    0
CONST:FFFFF80000C008DC          db  10h
CONST:FFFFF80000C008DD          db    0

[...]
```

Figure 22: Functions implementing hypercall functionalities

2.2.3 Secure Services

The secure kernel exposes services to the normal kernel, referred to as secure services in this work. They implement security-critical kernel operations that are executed in the secure, isolated environment. For a secure service to be invoked by the normal kernel, the kernel has to switch from VTL 0 to VTL 1 (see Section 1.3.2). This process is known as VTL call. In its essence, a VTL call is an execution context switch from a lower to a higher VTL. ([Mic 2017], Section 15.6.1) provides details on the VTL call process.

VTL calls are performed by the normal kernel issuing a hypercall with call code `0x11` – the `HvCallVtlCall` hypercall ([Mic 2017], Section 17). The normal kernel issues `HvCallVtlCall` by invoking the function chain `VslpEnterIumSecureMode → HvlSwitchToVsmVtl1 → HvlpVsmVtlCallVa`. The `VslpEnterIumSecureMode` function is invoked in the functions implemented as part of the normal kernel that require a secure service. `HvlpVsmVtlCallVa` is a variable storing a function referencing the trampoline of the hypercall page for invoking the hypercall with call code `0x11` (see Section 2.2.2, Figure 21). Figure 23 depicts this trampoline executed in the `HvlSwitchToVsmVtl1` function.

```
[...]

nt!HvlSwitchToVsmVtl1+0xa3:
fffff802`289cc633 ffd0              call    rax
0: kd> t
fffff802`2873f00f 488bc1            mov     rax,rcx
0: kd> p
fffff802`2873f012 48c7c111000000    mov     rcx,11h
0: kd> p
fffff802`2873f019 0f01c1            vmcall

[...]
```

Figure 23: Issuing a VTL call

Each secure service can be uniquely identified by an identification number, referred to as secure service call number (SSCN). In the context of the normal kernel, a SSCN is specified as the second parameter of `VslpEnterIumSecureMode`. The SSCN is then passed to the secure kernel as part of a data structure stored in the `rdx` register when a VTL call is issued. This structure is referred to as the VTL call data structure in this work. The table presented in the section 'Secure Services' of the Appendix lists the functions implemented as part of the normal kernel (column 'Function') that invoke secure services identified by SSCNs (column ' SSCN').

In addition to secure services, a VTL call supports the specification of other operations that can be executed by the secure kernel. Each operation is uniquely identified by an operation code, which is stored in the VTL call data structure. The operations are:

- managing the execution of a thread relevant to the secure kernel (operation code – `0x0`): Section 2.2.4 discusses more on this topic;
- invocation of a secure service (operation code – `0x01`);
- flushing the TLB (operation code – `0x02`): With respect to the design of VSM, the flushing of the TLB is considered a security-critical activity and is therefore executed in the secure environment. The TLB is involved in translations between virtual and physical addresses (see Section 1.3.1).

Figure 24 depicts the contents of the VTL call data structure when a VTL call is issued. In Figure 24, `0x01` is an operation code, indicating invocation of a secure service, and `0xD1` is a SSCN.

```
[...]

5: kd> db @rdx
ffffe201`1cfcfb00  01 00 d1 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
ffffe201`1cfcfb10  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
ffffe201`1cfcfb20  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
ffffe201`1cfcfb30  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
ffffe201`1cfcfb40  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
ffffe201`1cfcfb50  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
ffffe201`1cfcfb60  00 00 00 00 00 00 00 00-4f 40 48 8d 55 38 41 b0  ........O@H.U8A.
ffffe201`1cfcfb70  af e4 b7 2b 8f dd ff ff-57 40 0f b7 48 34 66 89  ...+....W@..H4f.

[...]
```

Figure 24: Contents of the VTL call data structure

In the context of the secure kernel, a VTL call is processed in the function `IumInvokeSecureService`, invoked by the `SkCallNormalMode` function. `IumInvokeSecureService` extracts the SSCN from the VTL call data structure and invokes the function(s) implementing the actual secure service identified by the SSCN. The secure kernel then continues the execution of `SkCallNormalMode`. This function invokes the trampoline of the hypercall page for invoking the hypercall with call code `0x12`. This is done for returning relevant data to the normal kernel and switching the execution context back to VTL 0. The hypercall with call code `0x12` is the `HvCallVtlReturn` hypercall ([Mic 2017], Section 17). It is used for switching from a higher to a lower VTL. This process is opposite to a VTL call and is referred to as VTL return. ([Mic 2017], Section 15.7.1) provides details on the VTL return process. In `SkCallNormalMode`, the trampoline for invoking `HvCallVtlReturn` is stored in the `ShvlpVtlReturn` variable (see Figure 25). This variable is populated during the initialization of the secure kernel, in the `ShvlpInitializeVsmCodeArea` function.

```
[...]

.text:000000014005FB95                    mov     cr2, rdx
.text:000000014005FB98                    mov     ecx, 1
.text:000000014005FB9D
.text:000000014005FB9D InvokeReturnHcall:
.text:000000014005FB9D                    call    cs:ShvlpVtlReturn

[...]
```

Figure 25: Execution of the HvCallVtlReturn hypercall in SkCallNormalMode

2.2.4 Normal-mode Services

The normal kernel exposes services to the secure kernel, referred to as normal-mode services in this work. These services implement kernel operations that are not implemented by the secure kernel, however, are necessary for this kernel or the IUM applications that it hosts to function. The secure kernel implements only a limited set of security-critical functionalities (see Section 2.2.3). This is because this kernel is designed to expose a minimal interface. It has a significantly smaller codebase than the one of the normal kernel. This reduces the risk of breaches due to design or implementation errors.

Example normal-mode services include semaphore and process management, and registry and filesystem input/output. The traditional system calls implemented as part of the normal kernel are invoked as normal-mode services by the secure kernel.

In the context of the secure kernel, normal-mode services that are implemented as system calls in the normal kernel, are invoked by executing functions with names starting with `Nt` or `Zw`. These functions may be invoked by IUM applications requesting kernel functionalities or the secure kernel itself. The functions

with names starting with Zw invoke the KiServiceInternal function. The system service index is stored in the eax register (see Section 2.2.1). Figure 26 depicts the invocation of KiServiceInternal by the function ZwTerminateProcess such that the system service index is 0x2C. This is the index of the NtTerminateProcess system call implemented in the normal kernel.

```
1   ZwTerminateProcess proc near
2
3   [...]
4
5   mov     eax, 2Ch
6   jmp     KiServiceInternal
7   retn
8   ZwTerminateProcess endp
```

Figure 26: ZwTerminateProcess invoking KiServiceInternal

KiServiceInternal invokes KiSystemServiceStart, a code segment of the KiSystemCall64 function (see Figure 18, Section 2.2.1). In KiSystemServiceStart, the secure kernel loads the variable IumSyscallDispatchTable (which is different than the one in the normal kernel). This is because the highest bit of the system service index is set (see line 17 in Figure 18). IumSyscallDispatchTable potentially contains pointers to functions implemented as part of the IumSyscallDispEntries array. IumSyscallDispEntries stores pointers to functions with prefix Nt, indexed by a system service index. Figure 27 depicts a portion of the contents of IumSyscallDispEntries.

```
[...]

.data:0000000140078580 IumSyscallDispEntries dq offset NtWorkerFactoryWorkerReady
.data:0000000140078580
.data:0000000140078580
.data:0000000140078588                db    1
.data:0000000140078589                db    0

[...]

.data:000000014007858F                db    0
.data:0000000140078590                dq offset NtWaitForSingleObject
.data:0000000140078598                db    4
.data:0000000140078599                db    0

[...]

.data:000000014007859F                db    0
.data:00000001400785A0                dq offset NtReleaseSemaphore
.data:00000001400785A8                db   0Ah
.data:00000001400785A9                db    0

[...]
```

Figure 27: Contents of IumSyscallDispEntries

After loading IumSyscallDispatchTable, KiSystemServiceStart invokes the function with prefix Nt indexed by the system service index stored in the eax register. The functions with prefix Nt invoke stubs for executing normal-mode services. These stubs are implemented in functions with names starting with Nk. For example, NtSetEvent invokes NkSetEvent.

Figure 28 depicts the process of executing normal-mode services by functions with prefix Nk. Figure 28 depicts the concrete example of NkTerminateProcess executing the system call NtTerminateProcess as a normal-mode service. NkTerminateProcess invokes IumGenericSyscall such that the first parameter is a system service index with the highest bit set. NkTerminateProcess sets the first parameter of IumGenericSyscall to 0x8000002C. 0X2C is the system service index of the NtTerminateProcess system call implemented in the normal kernel. IumGenericSyscall invokes SkSyscall such that its first parameter is the system service index

(`SysCallID` in Figure 28). `SkSyscall` sets the highest bit of the system service index to 0 (`SysCallID&0x7FFFFFFF` in Figure 28). The system service index is then passed to the `SkCallNormalMode` function as part of a data structure (`param` in Figure 28).

```
NkTerminateProcess ( [...] )
{
    return IumGenericSyscall(0x8000002C, [...] );
}

IumGenericSyscall(SysCallID, [...] )
{
    [...]

    return SkSyscall(SysCallID, [...] );
}

SkSyscall(SysCallID, [...] )
{
    [...]

    SysCallID = SysCallID & 0x7FFFFFFF;

    IumApi_NtGENERIC ( [...] );
    IumApi_NtGENERIC ( [...] );

    [...]

    WORD(param) = SysCallID;
    SkCallNormalMode(&param);
    IumApi_NtGENERIC ( [...] );

    [...]
}
```

Figure 28: Executing normal-mode services by functions with prefix Nk (NkTerminateProcess)

`SkCallNormalMode` executes a VTL return; that is, it switches from VTL 1 to VTL 0 (see Section 2.2.3). `SkCallNormalMode` passes the data structure provided by `SkSyscall` to VTL 0 (`param` in Figure 28). This structure is referred to as the VTL return data structure in this work. `SkCallNormalMode` executes a VTL return by invoking the hypercall with call code `0x12` (see Section 2.2.3).

In the context of the normal kernel, normal-mode services requested by IUM applications or by the secure kernel are handled in the `VslpEnterIumSecureMode` function. The `VslpDispatchIumSyscall` function, invoked by `VslpEnterIumSecureMode`, executes normal-mode services implemented as system calls in the normal kernel. The `PsDispatchIumService` function, invoked by `VslpEnterIumSecureMode`, executes other normal-mode services.

`VslpDispatchIumSyscall` and `PsDispatchIumService` are executed in the context of worker threads. These threads act as agents of entities running in the secure environment for executing normal-mode services. Normal-mode services requested by the secure kernel are executed in the context of a thread owned by the `Secure System` process. Normal-mode services requested by IUM applications are executed in the context of threads owned by these applications. Figure 29 depicts the invocation of `VslpDispatchIumSyscall` in the context of threads owned by the `Secure System` process (*[1]* in Figure 29) and the `BioIso.exe` IUM application (*[2]* in Figure 29). Next, the operation of the worker thread owned by `BioIso.exe` is discussed.

The thread enters the `VslpEnterIumSecureMode` function. This function issues VTL calls in a loop, by executing the hypercall with call code `0x11`. These VTL calls are issued by invoking `HvlSwitchToVsmVtl1`, such that the SSCN is set to 0 and the operation code is set to `0x0` (see Section 2.2.3). At a given point in time, the data returned from the VTL call contains either a system service index or

a normal-mode service code. Normal-mode service codes are used for uniquely identifying normal-mode services that are not implemented as system calls in the normal kernel.

[1]

```
Breakpoint 0 hit
nt!VslpDispatchIumSyscall:
fffff802`fdf616e0 56              push    rsi
0: kd> !thread
THREAD ffffbc829a699340  Cid 0004.0008  Teb: 0000000000000000 Win32Thread: 0000000000000000 RUNNING on processor 0
Not impersonating
Owning Process            ffffbc829a620040       Image:         System
Attached Process          ffffbc829cfe1040       Image:         Secure System

[...]

Priority 31 BasePriority 8 PriorityDecrement 0 IoPriority 2 PagePriority 5
Child-SP          RetAddr           : Args to Child                                                           : Call Site
ffff8f80`eb3bb9a8 fffff802`fdfb546f : fffff802`fe13aea0 00000000`00000001 ffffbc82`9a699340 fffff802`00000000 : nt!VslpDispatchIumSyscall
[...]
ffff8f80`eb3bbb90 fffff802`fdf6a456 : fffff802`fe15d180 ffffbc82`9a699340 fffff802`fdeb747c ffffffff`00057fc8 : nt!PspSystemThreadStartup+0x41
ffff8f80`eb3bbbe0 00000000`00000000 : ffff8f80`eb3bc000 ffff8f80`eb3b5000 00000000`00000000 00000000`00000000 : nt!KiStartSystemThread+0x16
```

[2]

```
Breakpoint 0 hit
nt!VslpDispatchIumSyscall:
fffff802`fdf616e0 56              push    rsi
3: kd> !thread
THREAD ffffbc82a0853080  Cid 09bc.09c4  Teb: 00000007eeccb000 Win32Thread: 0000000000000000 RUNNING on processor 3
Not impersonating
Owning Process            ffffbc82a0864800       Image:         BioIso.exe

[...]

Priority 8 BasePriority 8 PriorityDecrement 0 IoPriority 2 PagePriority 5
Child-SP          RetAddr           : Args to Child                                                           : Call Site
ffff8f80`ef2217e8 fffff802`fdfb546f : 00000000`00000001 00000000`00000001 ffffbc82`a0853080 ffffbc82`a0864800 : nt!VslpDispatchIumSyscall
[...]
ffff8f80`ef221a80 00007ffb`53a55e70 : 00000000`00000000 00000000`00000000 00000000`00000000 00000000`00000000 : nt!KiStartUserThreadReturn
00000007`eeacfa78 00000000`00000000 : 00000000`00000000 00000000`00000000 00000000`00000000 00000000`00000000 : ntdll!RtlUserThreadStart
```

Figure 29: Invocation of VslpDispatchIumSyscall

If the returned data contains a system service index, the `VslpDispatchIumSyscall` function invokes the corresponding system service routine. If the returned data contains a normal-mode service code, the `PsDispatchIumService` function invokes the corresponding normal service. `PsDispatchIumService` implements multiple condition blocks for invoking specific functions for a given normal service code.

Figure 30 depicts the presence of a system service index in the data returned from a VTL call issued in `VslpEnterIumSecureMode`. The system service index `0x48`, which specifies the `NtCreateEvent` system call, results in `VslpDispatchIumSyscall` invoking the `NtCreateEvent` system service routine. This routine is implemented in the normal kernel.

```
[....]

ffffe301`f7c9f8a0  00 04 00 00 0a 00 00 00-00 00 00 00 00 00 00 00  ................
ffffe301`f7c9f8b0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................

[....]

ffffe301`f7c9f8a0  00 02 48 00 0a 00 00 00-00 00 d7 6a 87 02 00 00  ..H........j....
ffffe301`f7c9f8b0  03 00 1f 00 00 00 00 00-00 00 00 00 00 00 00 00  ................

[...]

. .
Breakpoint 0 hit
nt!VslpDispatchIumSyscall:
fffff800`c3b636e0 56              push    rsi

[...]

6: kd> pc
nt!VslpDispatchIumSyscall+0x32:
fffff800`c3b63712 41ffd2          call    r10
6: kd> t
nt!NtCreateEvent:
fffff800`c3e06e40 48895c2408      mov     qword ptr [rsp+8],rbx
```

Figure 30: VslpDispatchIumSyscall invoking the NtCreateEvent system call

Once a normal-mode service is handled in VslpDispatchIumSyscall or PsDispatchIumService, the loop issuing VTL calls with operation code 0x0 is continued. At some point, the worker threads owned by BioIso.exe is put to sleep and terminated.

2.2.5 Security aspects

This section discusses implemented mechanisms for securing the communication interfaces analyzed in Section 2.2.1 - Section 2.2.4.

2.2.5.1 Restrictions on Issuing VTL Calls

Hyper-V restricts the issuing of VTL calls (see Section 2.2.3). For a VTL call to be issued, among other things, it has to be initiated from the most privileged CPU mode. For example, an entity invoking a VTL call have to execute with a Current Privilege Level (CPL) of 0 , which is assigned by the processor executing the entity. Further, the input values have to be valid. ([Mic 2017], Section 15.6.1.1) describes the restrictions on issuing VTL calls in greater detail.

2.2.5.2 Marshalling and Sanitization

The secure kernel marshalls and sanitizes the input and output data of VTL returns (see Section 2.2.4). An example is the invocation of functions referenced in the IumSyscallArgFcnTable array. These functions are invoked in the IumApi_NtGENERIC function. This function, in turn, is invoked in SkSyscall before and after a VTL return is issued by SkCallNormalMode. SkSyscall is where the secure kernel issues VTL returns to request normal-mode services (see Section 2.2.4). The functions referenced in the IumSyscallArgFcnTable array marshall and sanitize data passed to, and returned from, the normal kernel. This is a security measure for checking, controlling, and managing the data coming in, and going out of, the secure kernel in a centralized way. This significantly reduces the risk of exploiting implementation or design errors involving the malicious manipulation of this data. It also indicates that data originating from the normal kernel is not explicitly trusted by the secure kernel.

Figure 31 depicts the contents of the IumSyscallArgFcnTable array. The array references functions that represent data marshallers and sanitizers for data of simple types and on a per-type basis for data of complex types. The latter involve marshallers and sanitizers of data of specific data structures. The IumArg_GENERIC function (see Figure 31) performs generic marshalling and sanitization. IumArg_PALPC_MESSAGE_ATTRIBUTES and IumArg_PPORT_MESSAGE are examples of per-type marshallers and sanitizers of data of type ALPC_MESSAGE_ATTRIBUTES and PORT_MESSAGE.

```
.rdata:000000014006D1D0 IumSyscallArgFcnTable dq offset IumArg_GENERIC
.rdata:000000014006D1D0
.rdata:000000014006D1D8                 dq offset IumArg_PALPC_MESSAGE_ATTRIBUTES
.rdata:000000014006D1E0                 dq offset IumArg_PHANDLE
.rdata:000000014006D1E8                 dq offset IumArg_POBJECT_ATTRIBUTES
.rdata:000000014006D1F0                 dq offset IumArg_PPORT_MESSAGE
.rdata:000000014006D1F8                 dq offset IumArg_PSID
.rdata:000000014006D200                 dq offset IumArg_PWSTR
.rdata:000000014006D208                 dq offset IumArg_PUNICODE_STRING
.rdata:000000014006D210                 dq offset IumArg_PWORKER_FACTORY_DEFERRED_WORK
.rdata:000000014006D218                 dq offset IumArg_PWSTR
```

Figure 31: The IumSyscallArgFcnTable array

2.2.5.3 Access Control: Hypercalls

Hyper-V enforces access control over hypercall execution. For a partition to execute a hypercall protected by access control, it has to possess the required privileges. These privileges are assigned by the hypervisor to each partition in the form of flags declared as part of a bitmask. The bitmask is stored in the `HvPartitionPropertyPrivilegeFlags` Hyper-V variable ([Mic 2017], Section 4.2.2).

Figure 32 depicts the value of `HvPartitionPropertyPrivilegeFlags` assigned to the partition hosting the normal and the secure kernel. The `HvPartitionPropertyPrivilegeFlags` queries the value of `HvPartitionPropertyPrivilegeFlags`. This function is implemented as part of the `winhvr` driver (see Section 2.2.2). When the second parameter of `WinHvGetPartitionProperty` is `0x10000`, the function queries the value of `HvPartitionPropertyPrivilegeFlags` from the hypervisor (`003b800000002e7f` in Figure 32).

```
[...]

winhvr!WinHvGetPartitionProperty:
fffff80a`482d1e04 488bc4          mov     rax,rsp
0: kd> ?? @rdx
unsigned int64 0x10000

[...]

0: kd> dp 0xffffab81`6f5ee420
ffffab81`6f5ee420  003b8000`00002e7f 00000000`00000001
ffffab81`6f5ee430  ffffab81`6f5ee4d9 fffff80a`47a870c8
ffffab81`6f5ee440  fffff80a`47a6a5f0 ffff988a`16c90000

[...]
```

Figure 32: A value of HvPartitionPropertyPrivilegeFlags

([Mic 2017], Section 4.2.2) provides information on the layout of the privilege flags that are part of `HvPartitionPropertyPrivilegeFlags` and how the flags can be interpreted. For example, the `CreatePartitions` privilege flag implements access control over the execution of the `HvCreatePartition` hypercall. The `PostMessages` privilege flag implements access control over the execution of the `HvPostMessage` hypercall.

2.2.5.4 Access Control: IUM System Calls

The secure kernel implements access control over the execution of IUM system calls (see Section 2.2.1). The IUM system calls `IumSecureStorageGet`, `IumSecureStoragePut`, `IumCreateSecureSection`, `IumGetDmaEnabler`, `IumOpenSecureSection`, and `IumProtectSecureIo` are protected by access control. These functions evaluate flag values and return the error code `0xC0000022` (`STATUS_ACCESS_DENIED`, [ms_cc704588]) if the flags are not set. The evaluated flags are stored at offsets of an address stored in the `gs:8` register. The flags are declared as part of the policy options of the trustlet invoking the IUM system calls. These options are used by the secure kernel to enable or disable secure kernel functionalities for trustlet, such as execution of IUM system calls. The policy options are described in ([Yosif 2017], Section “Trustlet Policy Metadata”). Policy options of trustlets are signed data. A modification of this data results in invalidation of its signature and prevents the execution of the trustlets associated with the modified options.

2.2.5.5 Secure Data Sharing: Mailboxes

The concept of mailboxes enables trustlets to share data with entities running in the normal environment in a secure manner. Figure 33 depicts the workflow of mailbox-based data sharing.

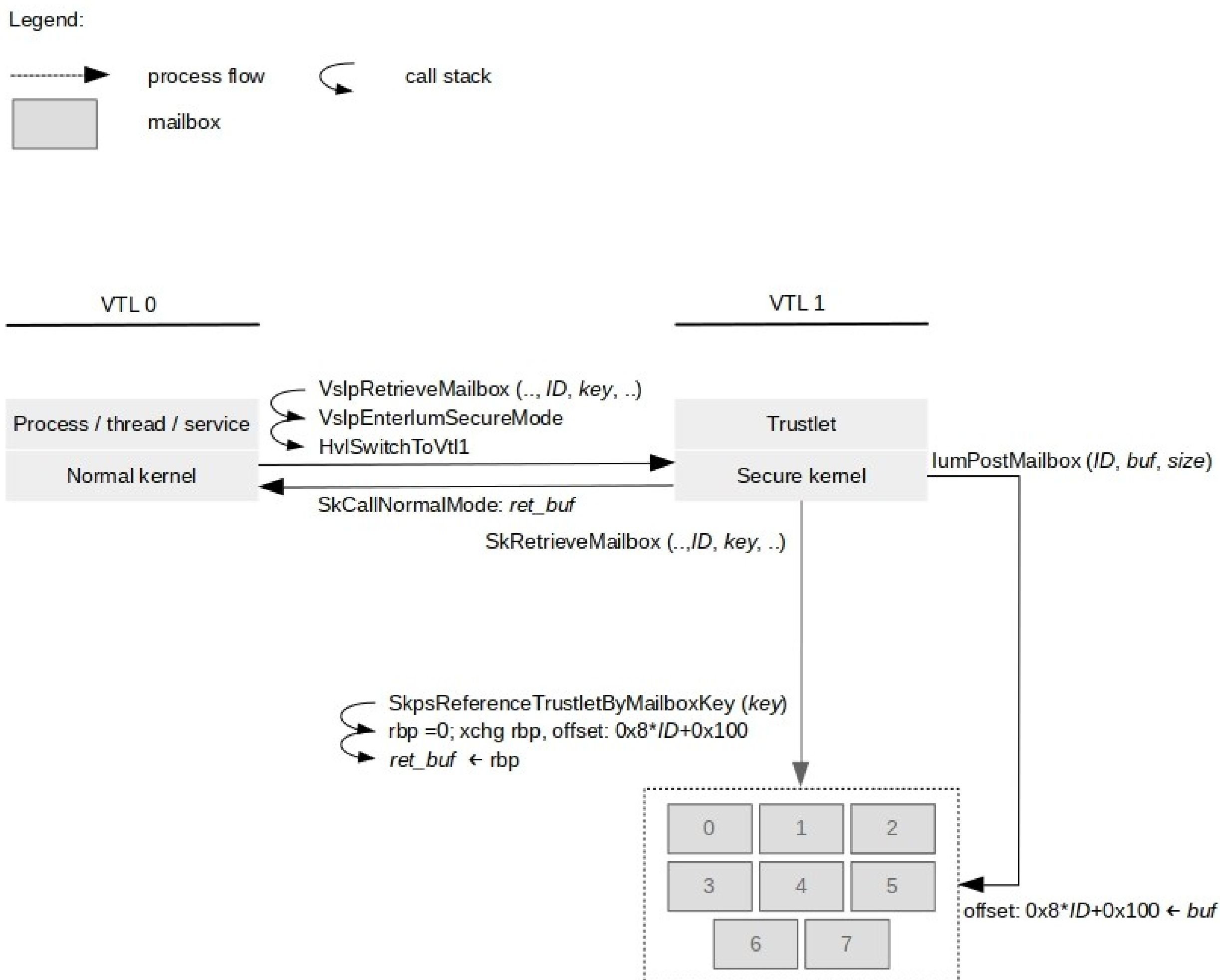

Figure 33: The workflow of mailbox-based data sharing

When a trustlet has data that it needs to share with an entity running in the normal environment, it populates a mailbox with the data. A mailbox is a memory region designated for storing shared data. Each mailbox can be uniquely identified by a mailbox ID. A trustlet populates a mailbox by issuing the `IumPostMailbox` IUM system call (see Section 2.2.1). The first parameter of this function is the ID of the mailbox to be populated (`ID` in Figure 33), the second stores the address of the data buffer where the shared data is stored (`buf` in Figure 33), and the third is the buffer size (`size` in Figure 33).

Each trustlet may have up to eight mailboxes. Possible mailbox IDs are between 0 and 7. The maximum size of each mailbox is 4092 bytes. `IumPostMailbox` evaluates the trustlet-provided mailbox ID and mailbox size against the previously mentioned upper values. `IumPostMailbox` then allocates heap memory. It also copies the data stored at the address that is the second parameter of `IumPostMailbox` at offset `0x4` of the allocated memory. The beginning of the allocated memory stores the size of the copied data. Finally, `IumPostMailbox` loads the address of the heap memory storing the shared data, and the size of the data, to an address referencing the mailbox indexed by the trustlet-provided mailbox ID. This is done by executing an atomic compare-and-exchange operation implemented with the `cmpxchg` instruction [ms_ms683560].

The implementation of cmpxchg in IumPostMailbox indicates that the trustlet-provided data to be shared is stored in a mailbox only if the mailbox is empty; that is, if the address referencing the mailbox is zeroed out. The address referencing the mailbox with ID is at offset 0x8*ID+0x100 of the address stored at gs:8+0x30 (offset: 0x8*ID+0x100 ← buf in Figure 33). gs:8 is the address stored in the gs:8 register at the time of issuing IumPostMailbox.

Once data is stored in a mailbox, entities running in the normal environment can retrieve it. This is done by invoking the VslRetrieveMailbox function implemented in the normal kernel. Parameters of VslRetrieveMailbox include the ID of the mailbox from which data is retrieved (ID in Figure 33) and a mailbox key (key in Figure 33). A mailbox key serves as a password for retrieving data stored in the mailboxes of a given trustlet. VslRetrieveMailbox stores the mailbox ID and the mailbox key in a buffer and issues a VTL call passing the buffer as input data.

VslRetrieveMailbox issues a VTL call by invoking the VslpEnterIumSecureMode and HvlSwitchToVtl1 functions (see Section 2.2.3). VslRetrieveMailbox requests a secure service with SSCN 0x13. This results in invoking the IumInvokeSecureService and SkRetrieveMailbox functions in the context of the secure kernel.

SkRetrieveMailbox first invokes the SkpsReferenceTrustletByMailboxKey function. This function identifies the trustlet whose mailboxes are protected by the mailbox key transferred from the normal environment. SkpsReferenceTrustletByMailboxKey searches through the attributes of running trustlets for this mailbox key. In order to search the attributes of a given trustlet, SkpsReferenceTrustletByMailboxKey invokes the SkFindTrustletAttribute function. This indicates that mailboxes keys are stored as trustlets' attributes. Attributes of a given trustlet store information associated with the trustlet and they are embedded in the executable implementing the trustlet ([Mic 2017], section "Trustlet Attributes", Table 3-5).

If SkpsReferenceTrustletByMailboxKey cannot identify a trustlet, SkRetrieveMailbox returns the error code 0xC0000034 (OBJECT_NAME_NOT_FOUND, [ms_cc704588]). If SkpsReferenceTrustletByMailboxKey identifies a trustlet, SkRetrieveMailbox allocates a buffer (ret_buf in Figure 33) and sets the value of the rbp register to 0 (rbp=0 in Figure 33). It then loads into rbp the address stored at offset 0x8*ID+0x100 of the address stored in the rsi register. As mentioned earlier, this address is presumably the address referencing the mailbox indexed by ID. The loading of the address is implemented using the xchg instruction (xchg rbp, offset: 0x8*ID+0x100 in Figure 33). This instruction exchanges the address stored at the offset 0x8*ID+0x100 of rsi with the value stored in rbp (i.e., 0). This effectively zeroes out the value stored at the offset 0x8*ID+0x100 of rsi and populates rbp. As discussed earlier, this indicates that the mailbox indexed by ID has been retrieved and may be populated again.

SkRetrieveMailbox then populates the newly allocated buffer with the data stored at offset 0x4 of rbp (ret_buf ← rbp in Figure 33). This is the data shared by the trustlet. SkRetrieveMailbox then issues a VTL return by invoking SkCallNormalMode in order to switch to VTL 0. SkRetrieveMailbox passes the buffer populated with shared trustlet data to VTL 0 (SkCallNormalMode: ret_buf in Figure 33). This provides the data shared by a trustlet to the requesting entity that runs in the normal environment.

2.2.5.6 Secure Data Sharing: Secure Storage Blobs

The concept of secure storage blobs enables trustlets to share data between each other in a secure manner. A secure storage blob is a memory region designated for storing shared data. The functionalities of the secure storage blob mechanism are implemented in the IUM system calls IumSecureStoragePut and IumSecureStorageGet. IumSecureStoragePut is used by trustlets for storing data in secure storage blobs. IumSecureStorageGet is used by trustlets for retrieving data stored in secure storage blobs.

The storing data into, and retrieving data from, secure storage blobs is subject to access control. This is based on an authentication value that trustlets accessing a secure storage blob have to provide. This authentication value may be either a collaboration ID or a trustlet instance.

A trustlet sharing data via a secure storage blob may associate a collaboration ID with the blob. This allows multiple trustlets that are in possession of this ID to access the blob. Same as the mailbox key, the collaboration ID of a given trustlet is stored as part of the trustlet's attributes (see Section 2.2.5.5 and [Mic 2017], section "Trustlet Attributes"). Alternatively to collaboration ID, a trustlet sharing data via a storage blob may associate a trustlet instance with the blob. This allows only the specific trustlet that is in possession of this instance to access the blob. A trustlet instance is a form of trustlet identity. It is a 16-byte number generated by the secure kernel and is unique to each instantiated trustlet ([Mic 2017], Section "Trustlet Identity"). When a trustlet is accessing a secure storage blob, if no collaboration ID is provided, the trustlet instance is used for authentication.

Analysing the `IumSecureStorageGet` function allows for better understanding the way in which access to secure storage blobs is secured. Figure 34 depicts a pseudo-code of the implementation of `IumSecureStorageGet`. The second parameter of `IumSecureStorageGet` is the address of a buffer where the data retrieved from a secure storage blob is to be stored (`buf` in Figure 34). The third parameter of `IumSecureStorageGet` is where the size of shared data is to be stored (`size` in Figure 34).

```
SkGetBlob(auth,..., buf, size)
{
    [...]

    foreach el in blob_array
    {
        if (auth == el->auth)
        {
            authenticated = 1;
            break;
        }
    }

    if (authenticated)
    {
        [...]

        memmove(buf, el+0x18, el+0x14);
        size = el+0x14;

        [...]
    }

    [...]
}

SkGetCollaborationId (..., &auth)
{
    [...]

    res = SkFindTrustletAttribute(..., &collabID);

    if (res)
        auth = collabID;
    else
        auth=gs:[8]+0x30+0xC8+0xD8;

    [...]
}

IumSecureStorageGet(..., buf, size)
{
    [...]

    SkGetCollaborationId(..., &auth);

    [...]

    SkGetBlob(auth, ..., buf, size);

    [...]
}
```

Figure 34: Pseudo-code of the implementation of IumSecureStorageGet

`IumSecureStorageGet` first invokes the `SkGetCollaborationId` function. This function attempts to extract the collaboration ID from the attributes of the trustlet retrieving shared data (`collabID` in Figure 34) by invoking `SkFindTrustletAttribute`. If a collaboration ID is not present (`if(res)` and `else` in Figure 34), then the trustlet instance is extracted. At the time of invoking `SkGetCollaborationID`, the trustlet instance is stored at offset `0x1D0` of the address stored in the `gs:[8]` register. `SkGetCollaborationID` then stores the extracted collaboration ID, or the trustlet instance, into a data buffer. This buffer is referred to as the authentication data buffer in this work (`auth` in Figure 34).

Once the authentication data buffer is populated, `IumSecureStorageGet` invokes `SkGetBlob`. This function first iterates through an array that stores pointers to secure storage blobs (`blob_array` in Figure 34). The address of this array is stored in a global variable of the secure kernel. Each element of the array stores a pointer to the authentication value associated with each storage blob, that is, a collaboration ID or a trustlet instance (`el->auth` in Figure 34). It also stores a pointer to the data stored in the blob.

During the iteration of the array of storage blobs, `SkGetBlob` compares the content of the previously populated authentication buffer with the authentication value associated with each storage blob. If a match is found (`authenticated = 1` in Figure 34), `SkGetBlob` copies the data stored in the blob in the buffer for storing shared data (`memmove` and `buf` in Figure 34). It also stores the size of the data in the variable for storing this size (`size` in Figure 34). The shared data is stored at offset `0x18` of the storage blob that `SkGetBlob` has granted access to (`el+0x18` in Figure 34). The size of the shared data is stored at offset `0x14` of this blob (`el+0x14` in Figure 34). The data buffer storing the shared data, and the data size, are then returned to the trustlet invoking the `IumSecureStorageGet` function.

2.3 Threats and Mitigations

This section provides an overview of potential threats against a VSM-enabled Windows environment. It also evaluates the capability of such an environment to mitigate these threats. The threats in scope may be realized during:

- the initialization process of VSM: 'VSM initialization' in Table 7 (see Section 2.1);
- the operation of the VSM-enabled Windows environment, once it has been initialized: 'VSM operation' in Table 7;
- communication and exchange of data ('VSM communication' in Table 5, see Section 2.2).

The threats listed in Table 7 are:

- loading of malicious VSM executables: VSM executables are the executables implementing Hyper-V (`hvix64.exe/hvax64.exe`), the secure kernel (`securekernel.exe`), the core IUM library, and IUM applications (e.g., `lsaIso.exe`). This threat involves loading a VSM executable that has been tampered with. An example threat scenario is loading a tampered `securekernel.exe` such that it leaks contents of memory assigned to IUM applications;
- setting malicious VSM configuration parameters at system start-up: This threat involves unauthorized modification at system start-up of VSM configuration parameters stored in the registry. These parameters are processed during VSM initialization (see Section 2.1). An example threat scenario is a bootkit [bh_boot]configuring VSM such that Credential Guard is disabled;
- unauthorized access to memory allocated to the secure environment: This threat involves reading or manipulating data stored in memory allocated to the secure environment. This includes memory allocated to the secure kernel and trustlets. An example threat is unauthorized access to data managed by the secure kernel;
- unauthorized access to shared trustlet data: This threat involves reading or manipulating shared trustlet data (see Section 2.2.5.5 and Section 2.2.5.6). An example threat scenario is unauthorized access to data marked by a given trustlet as shared only with another specific trustlet;

- exploitation of implementation or design vulnerabilities of the virtualization and secure environment: This threat involves exploiting an implementation or design vulnerability through the interfaces the secure kernel and the hypervisor expose (see Section 2.2). This can be achieved by passing maliciously crafted parameters to the functions implementing these interfaces. An example threat scenario is exploiting a vulnerability in a function implementing a secure service (see Section 2.2.3);
- unauthorized invocation of hypercalls: This threat involves a partition invoking hypercalls in an unauthorized manner. An example threat scenario is a non-root partition invoking hypercalls to access MSRs that are intended to be accessed only by the root partition;
- unauthorized invocation of VTL calls: This threat involves a partition invoking VTL calls in an unauthorized manner. An example threat scenario is a partition invoking VTL calls not originating from the most privileged CPU mode;
- unauthorized invocation of IUM system calls: This threat involves a trustlet invoking IUM system calls in an unauthorized manner. An example threat scenario is a trustlet invoking the `IumSecureStorageGet` IUM system call. This IUM system call is used for obtaining data shared by another trustlet.

In addition to potential threats, Table 7 lists mitigations that a VSM-enabled Windows environment implements as countermeasures against these threats (column 'Mitigation(s)' in Table 7). It also provides references (if any) to sections of this work where the mitigations are discussed in more detail.

	Threat	Mitigation(s)
VSM initialization	Loading of malicious VSM executables	*Integrity verification* (Section 2.1): The integrity of executables implementing the core VSM entities and the VSM features is verified using the Authenticode digital signing technology. The Authenticode signatures used for signing VSM executables have to be issued by Microsoft. The root of trust for the integrity verification of VSM executables is the Windows loader. The integrity of the Windows loader itself is verified by the boot manager or by the UEFI firmware, if Secure Boot is enabled.
	Setting malicious VSM configuration parameters at system start-up	*UEFI-based verification* (Section 2.1): Values of configuration parameters for initializing the core VSM entities, HVCI and Credential Guard, which are stored in the registry, are verified at system start-up against configuration values stored as UEFI variables when Secure Boot is enabled. The configuration parameters stored as *UEFI* variables are defined at the previous system shutdown and reflect the *VSM* configuration at that time (see Section 3.2). Secure Boot is a requirement for HVCI and Credential Guard. Secure Boot is a not a requirement for the initialization of the core VSM entities. In case of a mismatch between values stored in the registry and UEFI variables, the values stored as UEFI variables are used for initialization.
VSM operation	Unauthorized access to memory allocated to the secure environment	*Memory isolation* (Section 1.3): Hyper-V enforces isolation between the memory allocated to the normal and the secure environment. Entities operating in the normal environment cannot directly access memory allocated to the secure environment.

	Threat	Mitigation(s)
	Unauthorized access to shared trustlet data	*Mailboxes* (Section 2.2.5.5): Entities operating in the normal environment can access data of a trustlet that is explicitly designated as shared. Data is shared by storing it in mailboxes. Shared data can be accessed only after a valid mailbox key is provided. *Secure storage blobs* (Section 2.2.5.6): Trustlets can access data of another trustlet that is explicitly designated as shared. Data is shared by storing it in storage blobs. Shared data can be accessed only after a valid collaboration ID or trustlet instance is provided.
	Exploitation of implementation or design vulnerabilities of the virtualization and secure environment	*Minimal exposure* (Section 2.2): The secure kernel implements only security-critical functionalities. Therefore, its codebase is significantly smaller than that of the normal kernel. The secure kernel exposes a minimal interface to trustlets, implementing only security-critical functionalities. All other kernel functionalities are relayed to the normal kernel for execution. *Marshalling and hardening* (Section 2.2.5.2): The secure kernel marshalls and sanitizes data passed to, and returned from, the normal kernel. This is a security measure for checking, controlling, and managing the data coming in, and going out of, the secure kernel. *Secure software development practices* [ms_sdl]: Microsoft develops core system components, such as kernels and the Hyper-V hypervisor ([Kelbley 2010], Section 'Hypervisor Security') by following strict software development practices. These practices are defined as part of the SDL. This reduces the risk of exploitation due to implementation or design flaws. There are no obvious implementation or design flaws in the functionalities analyzed for the purpose of this work.
VSM communication	Unauthorized invocation of hypercalls	*Access control* (Section 2.2.5.3): Hyper-V enforces access control over hypercall execution. For a partition to execute a hypercall protected by access control, it has to possess the required privileges. These privileges are assigned by Hyper-V.
	Unauthorized invocation of VTL calls	*Access control* (Section 2.2.5.1): Hyper-V restricts the issuing of VTL calls. For a VTL call to be issued, among other things, it has to be initiated from the most privileged CPU mode and the input values have to be valid ([Mic 2017], Section 15.6.1.1).
	Unauthorized invocation of IUM system calls	*Access control* (Section 2.2.5.4): The secure kernel implements access control over the execution of IUM system calls.

Table 7: Threats and mitigations

3 Configuration and Logging Capabilities

This section provides an overview of the hardware and software requirements for the core VSM entities, and its features HVCI and Credential Guard, to be initialized and operational (Section 3.1). It also discusses the capabilities implemented in Windows 10 for configuring the VSM features HVCI and Credential Guard (Section 3.2). Finally, it provides an overview of the Windows 10 capabilities for logging VSM-related events (Section 3.3).

3.1 Hardware and Software Requirements

The hardware requirements for the core VSM entities and the VSM features HVCI and Credential Guard apply only to Hyper-V. These are: i) the SLAT CPU features (see Section 1.3.1), ii) a 64-bit CPU with virtualization extensions, with hardware-assisted virtualization and the NX bit set; that is, hardware-enforced DEP enabled; and iii) at least 4 GBs main memory [mic_hvreq]. Users can verify whether a platform fullfills these requirements with the `systeminfo.exe` utility [mic_hvreq].

The Trusted Platform Module (TPM) is not required for the core VSM entities, or HVCI and Credential Guard, to operate. However, it is preferred. If present, the TPM is used for securing Credential Guard secrets [mic_cgreq].

Table 8 lists the software requirements for the core VSM entities and the VSM features HVCI and Credential Guard:

- the column ‘Requirement’ specifies a concrete requirement;
- the column ‘Applies to’ specifies the technological domain, such as core VSM entities, to which the requirement applies;
- the paragraph ‘Description’ describes how the requirement applies to the technological domain from a technical perspective;
- the paragraph ‘Enablement’ provides brief information, and references to detailed information, on how the requirement can be enabled in Windows 10.

<table>
<tr><th>Requirement</th><th>Applies to</th></tr>
<tr><td>Hyper-V</td><td>Core VSM entities</td></tr>
<tr><td colspan="2">Description: Hyper-V implements the concept of VTLs (see Section 1.3.2). In addition, Hyper-V implements the communication interfaces between VTL 0 and VTL 1. This interface is the core mechanism of the VSM-based operation of Windows 10 (see Section 2.2).

Enablement: Hyper-V can be enabled by issuing the <code>Enable-WindowsOptionalFeature</code> PowerShell command or by enabling the <code>Hyper-V</code> Windows feature with the <code>Windows Features</code> utility [mic_hv].</td></tr>
<tr><td>UEFI and Secure Boot
Core VSM entities</td><td>Credential Guard
HVCI</td></tr>
<tr><td colspan="2">Description: HVCI and Credential Guard use variables stored in UEFI context. These variables are used for verifying configuration parameters stored in the system’s registry (see Section 2.1). This activity can be performed only if UEFI is present and Secure Boot is enabled. If Secure Boot is not enabled, HVCI and Credential Guard are not initialized. For HVCI and Credential Guard to operate, the core VSM entities, such as the secure and the normal kernel, have to be operational.

Enablement: For a platform hosting Windows 10 to be UEFI-enabled, the layout of the disk on which Windows is installed must be formatted in the globally unique identifier partition table (GPT) partition style [mic_gpt]. This is typically done by manufacturers. Alternatively, the layout of the installation media used for installing Windows (e.g., an USB drive) has to be formatted in the GPT partition style. Users can</td></tr>
</table>

enable Secure Boot by modifying the UEFI firmware settings.

Table 8: Software requirements for the core VSM entities and the VSM features

3.2 Configuration Capabilities

Users can configure the VSM features HVCI and Credential Guard by configuring group policies or by modifying the system's registry.

Group policy Windows 10 comes with a group policy for configuring HVCI and Credential Guard. This policy can be located with the `Group Policy Object Editor` utility, at the policy path `Computer Configuration` → `Administrative Templates` → `System` → `DeviceGuard` → `Turn On Virtualization Based Security`. This policy needs to be set to `Enabled` so that the VSM features can be configured. Users may configure three policy options. Table 9 presents their names and descriptions.

Policy option	Description
`Select Platform Security Level`	Possible values of this option are: `Secure Boot`; `Secure Boot and DMA Protection`. `Secure Boot` is the default value and it should be enabled on platforms that do not have devices with input-output memory management units (IOMMUs) [ms_vbs]. Secure Boot is a requirement for HVCI and Credential Guard (see Section 3.1). `Secure Boot and DMA Protection` should be enabled on platforms that support direct memory access (DMA). These are platforms that have devices with IOMMUs. This policy option enables mechanisms for mitigating DMA-based attacks and requires hardware support.
`Virtualization Based Protection of Code Integrity`	Possible values of this option are: `Disabled`; `Enabled with UEFI lock`; `Enabled without lock`; and `Not configured`. This policy option allows users to enable or disable HVCI. It also allows users to configure the way in which HVCI can be enabled or disabled. `Enabled without lock` enables HVCI such that it can be disabled by modifying the registry. This includes modifying the registry remotely by using Windows management utilities. `Enabled with UEFI lock` enables HVCI such that it cannot be disabled by modifying the registry. This includes modifying the registry remotely by using Windows management utilities. This option enables the use of an UEFI variable storing a flag. This flag indicates whether HVCI should be initialized at system startup (see Section 2.1). With Secure Boot active, this UEFI variable can be modified only by a physically present user. `Disabled` disables HVCI by modifying the registry. This includes modifying the registry remotely by using Windows management utilities. This option will take effect only if the option `Enabled without lock` is enabled. `Not configured` does not modify the settings that are configured at the point in time when this option is enabled. The settings are those enabled by configuring the previously mentioned policy options.
`Credential Guard Configuration`	Possible values of this option are: `Disabled`; `Enabled with UEFI lock`; `Enabled without lock`; and `Not configured`. This policy option allows users to enable or disable Credential Guard. It also allows users to configure the

Policy option	Description
	way in which Credential Guard can be enabled or disabled. The effects of enabling the values above are conceptually identical to those of enabling the values of the policy option `Virtualization Based Protection of Code Integrity`.

Table 9: Policy options for configuring HVCI and Credential Guard

Registry The policy option values presented in Table 9 are stored in the registry of Windows 10, at the key `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\DeviceGuard`. Figure 35 depicts the layout of this key. Table 10 lists and describes the values that may be stored in each of the registry keys depicted in Figure 35 (column 'Value' in Table 10).

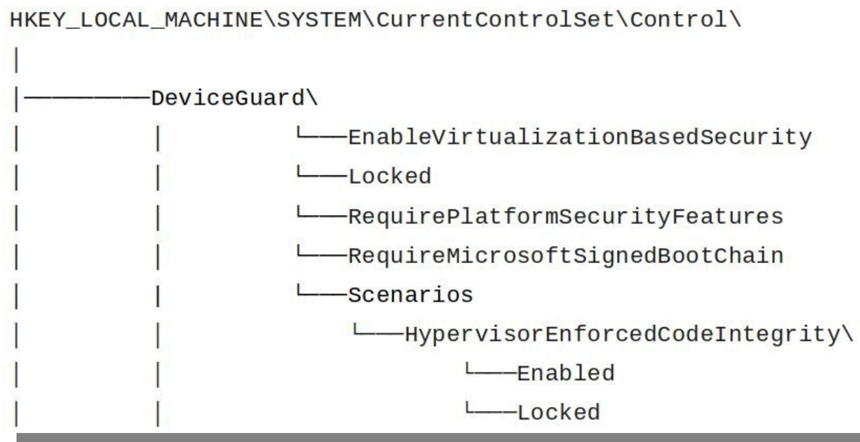

Figure 35: The layout of HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\DeviceGuard

Key	Value
`EnableVirtualizationBasedSecurity`	1: HVCI and Credential Guard can be configured
`Locked`	0: `Enabled without UEFI lock` for HVCI and Credential Guard is enabled (see Table 9, policy option `Credential Guard Configuration`)
	1: `Enabled with UEFI lock` for HVCI and Credential Guard is enabled (see Table 9, policy option `Credential Guard Configuration`)
`RequirePlatformSecurityFeatures`	1: `Secure Boot` is enabled (see Table 9, policy option `Select Platform Security Level`)
	3: `Secure Boot and` **DMA** `Protection` is enabled (see Table 9, policy option `Select Platform Security Level`)

`RequireMicrosoftSignedBootChain`	This registry key may have the value of `0` or `1`. The impact of these values is not documented. The value of this key is evaluated during system initialization (see Section 2.1.1).
`Scenarios\HypervisorEnforcedCodeIntegrity\Enabled`	`0`: `Disabled` is enabled - HVCI is disabled (see Table 9, policy option `Virtualization Based Protection of Code Integrity`)
	`1`: HVCI is enabled (see Table 9, policy option `Virtualization Based Protection of Code Integrity`)
`Scenarios\HypervisorEnforcedCodeIntegrity\Locked`	`0`: `Enabled without UEFI lock` for HVCI is enabled (see Table 9, policy option `Virtualization Based Protection of Code Integrity`)
	`1`: `Enabled with UEFI lock` for HVCI is enabled (see Table 9, policy option `Virtualization Based Protection of Code Integrity`)

Table 10: Values of registry keys for configuring HVCI and Credential Guard

In addition to the registry keys presented in Table 10, the registry keys `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Lsa\LsaCfgFlags` and `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Lsa\RunasPPL` are used for configuring Credential Guard. The values `0`, `1`, and `2` of `LsaCfgFlags` indicate that the option `Disabled`, `Enabled with` *UEFI* `lock`, and `Enabled without` *UEFI* `lock` for Credential Guard are enabled (see Table 9, policy option `Credential Guard Configuration`). If set, `RunasPPL` configures the *LSA process* to be executed under *PPL* protection [ms_runasppl] (see Section 2.1.4).

Windows implements an extensive Windows Management Instrumentation (WMI) interface ([ERNW WP5], Section 3.1.1) for programmatically configuring Hyper-V. A detailed documentation of this interface is available at [mic_hh850319].

3.3 Logging Capabilities

Windows 10 uses the ETW framework for logging VSM-related events ([ERNW WP2], Section 4.1). Table 11 provides the names and the GUIDs of ETW providers logging VSM-related events. The `Microsoft-Windows-IsolatedUserMode` and `Microsoft-Windows-DeviceGuard` providers log exclusively events related to VSM. These are events related to the operation of the IUM, HVCI, and Credential Guard (see Section 1.3.2). The `Microsoft-Windows-Wininit` provider does not log exclusively events related to VSM. However, some events that this provider logs are related to the initialization of the IUM application `lsaIso.exe`.

ETW provider	GUID
`Microsoft-Windows-IsolatedUserMode`	`73a33ab2-1966-4999-8add-868c41415269`
`Microsoft-Windows-Wininit`	`206f6dea-d3c5-4d10-bc72-989f03c8b84b`
`Microsoft-Windows-DeviceGuard`	`f717d024-f5b4-4f03-9ab9-331b2dc38ffb`

Table 11: ETW providers logging VSM-related events

Table 12, Table 13, and Table 14 present the Event IDs, and their descriptions, under which the `Microsoft-Windows-IsolatedUserMode`, `Microsoft-Windows-Wininit`, and `Microsoft-Windows-DeviceGuard` providers may log VSM-related events (column 'Event ID' and 'Event Message' respectively). The `wevtutil` utility ([ERNW WP2], Section 4.3) displays the Event IDs and their descriptions. In Table 12, Table 13, and Table 14, numbers preceeded by the percent sign (%) mark dynamic content generated at run-time. The event descriptions in these tables are as provided by Microsoft.

Event ID	Event Message
1	Secure Trustlet %4 Id %1 and Pid %2 started with status %3.
2	Secure Trustlet Id %1 and Pid %2 stopped with status %3.
3	Secure Kernel started with status %1 and flags %2.
4	Secure Trustlet Id %1 and Pid %2 failed to start with status %3.

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

Event ID	Event Message
13	Credential Guard (LsaIso.exe) was started and will protect LSA credentials.
14	Credential Guard (LsaIso.exe) configuration: %1, %2
15	Credential Guard (LsaIso.exe) is configured but the secure kernel is not running; continuing without Credential Guard.
16	Credential Guard (LsaIso.exe) failed to launch: %1
17	Error reading Credential Guard (LsaIso.exe) UEFI configuration: %1

Table 13: Event IDs generated by the ETW provider Microsoft-Windows-Wininit

Event ID	Event Message
7000	Device Guard successfully processed the Group Policy: Virtualization Based Security = %1, Secure Boot = %2, DMA Protection = %3, Virtualization Based Code Integrity = %4, Credential Guard = %5, Reboot required = %6, Status = %7.
7001	Device Guard failed to process the Group Policy to enable Virtualization Based Security (Status = %1): %2.
7002	Device Guard failed to process the Group Policy to disable Virtualization Based Security (Status = %1): %2.
7010	Device Guard successfully processed the Group Policy: Configurable Code Integrity Policy = %1, Policy file path = %2, Reboot required = %3, Status = %4.
7011	Device Guard failed to process the Group Policy to enable Configurable Code Integrity Policy (Status = %1): %2
7012	Device Guard failed to process the Group Policy to disable Configurable Code Integrity Policy (Status = %1): %2
7013	Device Guard is not available in this edition of Windows

Table 14: Event IDs generated by the ETW provider Microsoft-Windows-DeviceGuard

Hyper-V implements extensive ETW-based logging functionality. Logged events that are related to the operation of Hyper-V can be viewed with the `Event Viewer` utility [msblog_el]. VSM entities, such as

Hyper-V and IUM applications, may implement ETW providers that are not registered under names. These do not necessarily provide logged data in a form as required by standard Windows log viewing utilities, such as `Event Viewer`.

Appendix

Tools

Tool	Availability and description
IDA	*Availability:* https://www.hex-rays.com/products/ida/index.shtml [Retrieved: 14/5/2018] *Description*: A disassembly and debugging framework.
Windows Debugger (windbg)	*Availability:* https://developer.microsoft.com/en-us/windows/hardware/download-windbg [Retrieved: 14/5/2018] *Description*: A debugger for the Windows system.
Group Policy Object Editor	*Availability:* Distributed with Windows 10 *Description*: A tool for configuring group policies.
Event Viewer	*Availability:* Distributed with Windows 10 *Description*: A tool for viewing logged system events.
wevtutil	*Availability:* Distributed with Windows 10 *Description*: A tool for querying running logging mechanisms.

Secure Services

Function	SSCN
DbgkCopyProcessDebugPort	0xB
HvlCollectLivedump	0xE9
HvlInitializeProcessor	0x2
HvlNotifyDebugDeviceAvailable	0xF0
HvlpGetSecurePageList	0x802
HvlPrepareForRootCrashdump	0xEC
HvlPrepareForSecureHibernate	0xEB
HvlpStartSecurePageListIteration	0x800
KeBalanceSetManager	0xD1
KeCopyPrivilegedPage	0xE4
KeRequestTerminationThread	0x8

Function	SSCN
KeReservePrivilegedPages	0xD2
KeSecureProcess	0x6
KeSetPagePrivilege	0xE6/0xE3/0xE5
KeUnsecureProcess	0x1B
MiApplyDynamicRelocations	0xD3
MiFlushEntireTbDueToAttributeChange	0x0
NtDebugActiveProcess	0xB
NtRemoveProcessDebug	0xB
PopAllocateHiberContext	0x1F
PspInitPhase3	0x3
PspUserThreadStartup	0x0
VslAbortLiveDump	0x28
VslCloseSecureHandle	0x1B
VslConfigureDynamicMemory	0x21
VslConnectSwInterrupt	0x22
VslCreateSecureAllocation	0x13
VslCreateSecureImageSection	0x16
VslCreateSecureProcess	0x5
VslCreateSecureThread	0x7
VslEnableOnDemandDebugWithResponse	0x10
VslEndSecurePageIteration	0x801
VslExchangeEntropy	0x1E
VslFastFlushSecureRangeList	0xE1
VslFillSecureAllocation	0x14
VslFinalizeLiveDumpInSk	0x27/0x28
VslFinalizeSecureImageHash	0x17
VslFinishSecureImageValidation	0x18
VslFlushSecureAddressSpace	0xE0

Function	SSCN
VslFreeSecureHibernateResources	0x20
VslGetNestedPageProtectionFlags	0xE7
VslGetOnDemandDebugChallenge	0xF
VslGetSecurePebAddress	0xC0
VslGetSecureTebAddress	0xC
VslGetSetSecureContext	0xE
VslIsTrustletRunning	0x12
VslIumEfiRuntimeService	0xE8
VslIumEtwEnableCallback	0xD4
VslLiveDumpQuerySecondaryDataSize	0x23
VslMakeCodeCatalog	0x15
VslNotifyShutdown	0xEE
VslpAddLiveDumpBufferChunk	0x25
VslpConnectedStandbyPoCallback	0x29
VslpConnectedStandbyWnfCallback	0x29
VslpIumPhase0Initialize	0xD0
VslpIumPhase4Initialize	0x1
VslpKsrEnterIumSecureMode	0xF1
VslPrepareSecureImageRelocations	0x19
VslpSetupLiveDumpBuffer	0x26
VslQuerySecureKernelProfileInformation	0x2A
VslRegisterLogPages	0xEA
VslRegisterSecureSystemProcess	0x4
VslRelocateImage	0x1A
VslReportBugCheckProgress	0xED
VslRetrieveMailbox	0x11
VslRundownSecureProcess	0xA
VslSetupLiveDumpBufferInSk	0x24/0x28

Function	SSCN
VslSlowFlushSecureRangeList	0xE2
VslTerminateSecureThread	0x9
VslTransferSecureImageVersionResource	0x1D
VslValidateDynamicCodePages	0x1C
VslValidateSecureImagePages	0xC1

Reference Documentation

mic_hh850319 https://msdn.microsoft.com/en-us/library/windows/desktop/hh850319(v=vs.85).aspx [25/4/2018]

ms_vbs https://docs.microsoft.com/en-us/windows/security/threat-protection/device-guard/deploy-device-guard-enable-virtualization-based-security [24/4/2018]

mic_gpt https://docs.microsoft.com/en-us/windows-hardware/manufacture/desktop/windows-and-gpt-faq [27/4/2018]

mic_hv https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/quick-start/enable-hyper-v [27/4/2018]

mic_cgreq https://docs.microsoft.com/en-us/windows/security/identity-protection/credential-guard/credential-guard-requirements [27/4/2018]

bh_boot http://www.blackhat.com/presentations/bh-usa-09/KLEISSNER/BHUSA09-Kleissner-StonedBootkit-SLIDES.pdf [17/5/2018]

ms_ms683560 https://msdn.microsoft.com/en-us/library/windows/desktop/ms683560(v=vs.85).aspx [20/4/2018]

Yosif 2017 Yosifovic, Pavel; Ionescu, Alex; Russinovich, Mark E.; Solomon, David A. : Windows Internals, Part 1 and Part 2

ms_lsacfgflag https://docs.microsoft.com/en-us/windows/security/identity-protection/credential-guard/credential-guard-manage [26/4/2018]

ms_runasppl https://docs.microsoft.com/en-us/windows-server/security/credentials-protection-and-management/configuring-additional-lsa-protection [26/4/2018]

mc_bcdedit https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/bcdedit--set [26/4/2018]

ms_cc704588 https://msdn.microsoft.com/en-us/library/cc704588.aspx [19/4/2018]

Russinovich 2012 Russinovich, Mark E.; Solomon, David A.; Ionescu, Alex: Windows Internals, Part 2

ms_aa362656 https://msdn.microsoft.com/en-us/library/windows/desktop/aa362656(v=vs.85).aspx [26/4/2018]

mc_aa362670 https://msdn.microsoft.com/en-us/library/windows/desktop/aa362670(v=vs.85).aspx [25/4/2018]

mic_setvsm https://docs.microsoft.com/en-us/powershell/module/hyper-v/set-vmsecurity?view=win10-ps [25/4/2018]

mic_biom https://docs.microsoft.com/en-us/windows/security/identity-protection/hello-for-business/hello-biometrics-in-enterprise [3/5/2018]

ms_809132 https://msdn.microsoft.com/en-us/library/windows/desktop/mtC(v=vs.85).aspx [3/5/2018]

Mic 2017 Microsoft Corporation: Hypervisor Top Level Functional Specification

ms_vbsi https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs [2/5/2018]

msblog_el https://blogs.technet.microsoft.com/virtualization/2018/01/23/looking-at-the-hyper-v-event-log-january-2018-edition/ [25/4/2018]

ms_sdl https://www.microsoft.com/en-us/sdl [4/5/2018]

Kelbley 2010 Kelbley, John; Sterling, Mike;: Windows Server 2008 R2 Hyper-V: Insiders Guide to Microsoft's Hypervisor

Mic 2017 Microsoft: Hypervisor Top Level Functional Specification

ERNW WP2 ERNW GmbH: SiSyPHuS Win10 (Studie zu Systemaufbau, Protokollierung, Härtung und Sicherheitsfunktionen in Windows 10): Work Package 2

mic_hvreq https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v/system-requirements-for-hyper-v-on-windows [27/4/2018]

ERNW WP5 ERNW GmbH: SiSyPHuS Win10 (Studie zu Systemaufbau, Protokollierung, Härtung und Sicherheitsfunktionen in Windows 10): Work Package 5

Keywords and Abbreviations

Abbreviations ... 57
application programming interface ... 6f., 11f., 18, 28
Boot Configuration Database ... 22, 25, 27
Bundesamt für Sicherheit in der Informationstechnik ... 5, 10
central processing unit ... 5, 7, 10, 12f., 15ff., 38, 44ff.
Current Privilege Level ... 38
data execution prevention ... 12, 46
direct memory access ... 47f., 50
Event Tracing for Windows ... 9, 14f., 49ff.
extended key usage ... 24f., 27f.
Gigabyte ... 7, 12, 46
globally unique identifier ... 9, 14, 49f.
*globally u*nique identifier partition table ... 46
guest physical addresses ... 16f., 31
guest virtual addresses ... 16f.
hypervisor code integrity ... 6f., 9, 11f., 14, 19, 21ff., 44, 46ff.
input-output memory management units ... 47
interprocess communication ... 6, 11, 18
isolated user mode ... 6ff., 11ff., 18f., 21, 24, 28ff., 34, 36, 39ff., 43ff., 49, 51
local security authority ... 18, 27, 49f.
long-term servicing branch ... 5, 10
memory management units ... 16f.
model-specific register ... 29, 31, 44
no execute ... 7, 12, 46
Protected Process Light ... 27, 49
second-level address translation ... 7, 12, 17, 46
Secure Hash Algorithm ... **24**
secure service call number ... 33f., 36, 41, 52
Software Development Lifecycle ... 9, 45
system physical addresses ... 16f.
transaction lookaside buffer ... 17, 33
Trusted Platform Module ... **46**
Unified Extensible Firmware Interface ... 6ff., 11ff., 20ff., 26f., 44, 46ff.
virtual secure mode ... 5ff., 17ff., 26, 28, 31, 33, 43ff., 49f.
*virtual trus*t level ... 5, 7ff., 12ff., 17f., 33f., 36ff., 41, 44ff.
Windows Management Instrumentation ... 49