Master Thesis
Computing Science

Computest & Radboud University Nijmegen

Analysis of Windows Secure Kernel security bugs

Author:
Jonathan Jagt

Company Supervisor:
Daan Keuper - Computest
University Supervisor:
Erik Poll

Second reader:
Xavier de Carné de Carnavalet

June 24, 2025

Abstract

In the last decade, Microsoft has introduced several technologies in Windows to harden the Windows kernel ('NT Kernel') against exploitation. These technologies leverage hardware virtualization to introduce new security boundaries between the NT Kernel and the 'Secure Kernel'. With these new security boundaries enabled, even when the NT Kernel is exploited by an attacker, the impact of exploitation is limited. The Secure Kernel is a Windows operating system component that runs with higher privileges than the NT Kernel and manages the NT Kernel. Communication between the Secure Kernel and NT Kernel uses 'Hypercalls', which are similar to the concept of syscalls. The Secure Kernel therefore becomes a target for attackers aiming to fully compromise the Windows operating system, even when the new security boundaries are enabled.

In this master thesis, we analyze bugs in the Secure Kernel for which a patch is already available. The patched bugs, as well as the functionality where the bugs reside, are almost always undocumented. The goal of this analysis is to hopefully gain new knowledge about these undocumented functionalities and bugs to better comprehend the Secure Kernel. This new knowledge is useful for other security researchers who want to discover new security bugs in the Secure Kernel.

Debugging the Secure Kernel is non-trivial, as it runs outside the context of regular Windows debugging capabilities and there is a lack of public Secure Kernel debugging documentation. We have created and documented our own Secure Kernel setup, which we use for dynamic code analysis when finding an interesting Secure Kernel n-day security bug. From developing the Secure Kernel debugging setup, we strongly recommend using VMWare as the virtualization software for a Secure Kernel debugging setup.

The first step in analyzing a bug in the Secure Kernel is to discover the location of the bug by comparing vulnerable and patched binaries to find changed code related to the bug. The second step is to try to trigger the bug in our Secure Kernel debugging setup. In the third step, if triggering the bug succeeds, we determine if the bug can be used to compromise the Secure Kernel. If that is the case, we try to build a proof-of-concept exploit for the bug to compromise the Secure Kernel. However, in the third step, we encounter various difficulties due to hardening done by Microsoft to the Secure Kernel, which makes it impossible to exploit the Secure Kernel without additional primitives.

As expected, debugging, interacting with, and understanding the Secure Kernel is difficult. We document a Secure Kernel debugging setup, which we use to successfully locate and write a proof-of-concept to trigger the security bug. By using the vulnerable code pattern of the n-day security bug, we also find a zero-day security bug in the Secure Kernel.

Contents

Glossary

Child Partition 'Partition that hosts a guest operating system - All access to physical memory and devices by a child partition is provided via the Virtual Machine Bus (VMBus) or the hypervisor' [16].

Enclave When Virtualization Based Security (VBS) is enabled, Enclaves provide a software-based Trusted Execution Environment available for developer to securely store sensitive data.

EPT Extended Page-Table: Implementation of Intel of Second Level Address Translation. See Section 2.1.3.

EPTP Extended Page Table Pointer: a pointer stored in the Virtual-Machine Control Structure (VMCS) which points to the first table for EPT PML4 address translation.

GPA Guest Physical Address: physical addresses seen by the operating system of the Guest software.

Guest software 'Each virtual machine (VM) is a guest software environment that supports a stack consisting of operating system (OS) and application software' [9].

GVA Guest Virtual Address: virtual addresses used by processes in the Guest software operating system.

HVCI Hypervisor-Enforced Code Integrity: also referred to as 'memory integrity' or 'hypervisor enforced code integrity'. HVCI protects against modifications of the Control Flow Guard bitmap for kernel drivers, as well as code integrity in the NT Kernel by protecting the code that is responsible for validating certificates related to loaded kernel drivers. HVCI was originally released as part of 'Device Guard', but Device Guard is no longer used.

Hypercall 'Interface for communication with the hypervisor - The Hypercall interface accommodates access to the optimizations provided by the hypervisor' [16].

IUM Isolated User Mode: A new mode of execution in user-mode in VTL1 which is used by Trustlets.

KASLR Kernel Address Space Layout Randomization.

KB Knowledge Base: used as the term to refer to an article published by Microsoft for a specific Windows operating system update.

KVM Kernel-based Virtual Machine: a virtualization module in the Linux kernel that allows the kernel to function as a hypervisor.

LSASS Local Security Authority: 'manages the local system policy, user authentication, and auditing while handling sensitive security data such as password hashes and Kerberos keys' [19].

Partition 'Hyper-V supports isolation in terms of a partition. A partition is a logical unit of isolation, supported by the hypervisor, in which operating systems execute' [18].

PDP Page Directory Pointer.

PDT Page Directory Table.

PML4 Page Map Level 4.

PT Page Table.

PTE Page Table Entry.

Root Partition 'Sometimes called parent partition. Manages machine-level functions such as device drivers, power management, and device hot addition/removal. The root (or parent) partition is the only partition that has direct access to physical memory and devices' [16].

RVI Rapid Virtualization Indexing: Implementation of AMD of Second Level Address Translation. See Section 2.1.3.

Secure Kernel Another kernel that runs in parallel with the normal NT Kernel. Due to the lack of loading any third-party modules and separation due to VTL, the secure kernel is considered more secure. The Secure Kernel runs in VTL1.

SKPG Secure Kernel Patch Guard: somtimes also referred to as 'HyperGuard'. Secure Kernel system that periodicly checks if the NT Kernel is not changed unauthorized with malicious code..

SLAT Second Level Address Translation: a hardware virtualization feature that allows to virtualize physical memory of a guest machine. Also known as 'nested-paging'. See Section 2.1.3.

SPA System Physical Address: physical addresses which point to physical memory in the hardware.

SSCN Secure System Call Number: the number used for the Hypercall to the Secure Kernel.

SVA System Virtual Address: virtual addresses seen by the hypervisor process.

TOCTOU Time-Of-Check Time-Of-Use.

Trustlet Trustlets (also known as trusted processes, secure processes, or IUM processes) are programs running as IUM processes in VSM.

VBS Virtualization Based Security: By using hardware virtualization technologies implemented in the CPU, new security contexts are introduced to extend the kernel rings security models with Virtual Trust Levels.

VM Virtual Machine.

VMCS Virtual-Machine Control Structure: VMX transitions and VMX non-root operations are stored in a data structure called a Virtual-Machine Control Structure [9]. See Section 2.1.2.

VMM Virtual-Machine Monitor: 'A VMM acts as a host and has full control of the processor(s) and other platform hardware' [9].

VMX Virtual Mode Extensions: VMX defines processor-level support for Virtual Machines. See Section 2.1.1.

VSM Virtual Secure Mode: A set of hypervisor capabilities that leverages hardware virtualization of the CPU to introduce new memory security features in the NT Kernel [24].

VTL Virtual Trust Level: A hardware virtualization segregation to define boundaries between memory regions used to define the 'normal' world and the 'secure' world.

When this PDF is viewed digitally, it is possible to click in the text on words described in the glossary to jump to the explanation in the glossary.

1 Introduction

This thesis takes a look at patched security vulnerabilities in the Secure Kernel. We started this research without any prior knowledge about Windows, the Windows Kernel or the Secure Kernel. The steps in our research are described below.

1. The first step is to document the existing knowledge about the Secure Kernel. This will give insight in how the Secure Kernel operates and can be used for more vulnerability research.
2. We explain how we use patch diffing to rediscover n-day vulnerabilities in the Secure Kernel.
3. We create and document a Secure Kernel debugging setup as well as a setup to interact with the Secure Kernel. This setup is useful for dynamic code analysis during the Exploitability Assessment and to create a proof of concept exploit during Exploitation.
4. We will do an Exploitability Assessment on certain CVEs that are already patched in the Secure Kernel (see Section 5).

   With this we hope to give insights in exploitability of certain (type of) bugs, by going beyond Microsoft Security Response Center's own assessment given (for example 'Exploitation Less Likely'). For example, can a heap overflow bug be sufficient for exploitation or does it need to be chained with a bug that gives another primitive in order to perform exploitation? This step also helps understanding which parts of the Secure Kernel were vulnerable and thus where vulnerability research already has been done. Furthermore, the practical explanation of performing Secure Kernel exploitation helps other vulnerability researchers starting vulnerability research on the Secure Kernel.
5. If we conclude that we can trigger the bug and gives a useful primitive to control the Secure Kernel, we will write a proof of concept exploit for it (see Section 6).

For the first step, we do a literature study in Section 2 in order to learn more about the Secure Kernel. The literature study starts with the foundation of hardware virtualization (see Section 2.1) and further explains how Hyper-V uses hardware virtualization (see Section 2.2). We then explain in Section 2.3 how the technology of Hyper-V makes the Secure Kernel possible.

The second step is to explain how we rediscover n-day vulnerabilities based on the very limited available information online, which is discussed in Section 3. Most of the time, this information is a security update identifier and a single sentence stating the vulnerability class, for example 'buffer overflow' or 'heap overflow'. Since it is an n-day with an already available fix for it, we should be able to do 'patchdiffing' (comparing assembly between an old and an updated binary to find the differences, see Section 3.1.1) on the DLLs and EXEs of the Secure Kernel and audit only the changed code. This should greatly limit the scope of the security audit. This bindiffing methodology is also portable to other n-day analyses of Windows updates and therefore valuable to document.

In the third step, we create and document a debugging setup as well as an setup to interact with the Secure Kernel so we can trigger bugs and troubleshoot bugs dynamically. In Section 4 we discuss the debugging setup. We explain the setup and corresponding problems we encountered while developing this setup. In Section 4.2.2 we give a practical explanation of a QEMU/KVM Secure Kernel debugging setup used during our research, including the required configurations that need to be done in order to reproduce this

setup. Some alternative setups are also discussed in Section 4.3. We conclude this step with an explanation of the setup we use to interact with the Secure Kernel, which is discussed in Section 4.4.

For step four, in Section 5 we explore the possibilities of exploiting the uncovered bugs found in step two using the setup explained in step three. Not every is exploitable since sometimes you need another bug to create an extra primitive for exploitability (for example, a Secure Kernel address leak to defeat Kernel Address Space Layout Randomization (KASLR)). Understanding the necessary primitives in order to perform exploitation is key to judging exploitability of a certain bug, which we will explore in step four.

If a security bug appears to be exploitable, we will then proceed to step five. In step five (see Section 6), we discuss the Exploitation attack scenario of the exploitable security bug(s). We implement a proof of concept exploit script to attack the Secure Kernel using the security bug(s), giving a practical overview of performing Secure Kernel exploitation based on the knowledge gained in previous steps. This step helps other researchers understanding the implementation of Secure Kernel bugs for which they can use this knowledge to do more vulnerability research and exploitation on other components of the Secure Kernel.

We give suggestions for future research in Section 7 and we then conclude our research in Section 8.

For the list of references, it is important to note that [3] is the only 'official' literature found, which is a master thesis about the "Live forensics of the Windows 10 secure kernel". Other sources are personal blogs or corporate blogs, as well as Microsoft Learn pages. The lack of scientific literature is because there has been no research done on the Secure Kernel by scientific researchers. Therefore, we have to rely on unofficial literature such as blog posts, GitHub repositories, and documentation done by other researchers.

For readers that have no prior knowledge about hardware virtualization, Hyper-V, or the Secure Kernel, Section 2 is the starting point. For readers that do have prior knowledge about hardware virtualization, Hyper-V, and the Secure Kernel, Section 5 will explain the Exploitability Assessment done in this master thesis, which is the practical application of the theory described in Section 2, Section 3, and Section 4.

2 Background

In this chapter, we discuss the background information required for understanding the research done in this thesis. In Section 2.1, we discuss the underlying technologies and concepts used for hardware virtualization. Hardware virtualization is used by Hyper-V to introduce a security boundary between the 'normal world' (where the NT Kernel is executed) and the 'secure world' (where the Secure Kernel is executed). We proceed to discuss Hyper-V in Section 2.2, which explains the concepts 'Partitions' and 'Virtual Trust Levels (VTLs)' used by Hyper-V. We discuess the Secure Kernel in Section 2.3. Furthermore, we explain how the Secure Kernel works together with the NT Kernel (see Section 2.3.1) and how communication is done through Hypercalls between the Secure Kernel and the NT Kernel (see Section 2.3.2). We explain in Section 2.3.3 how the Secure Kernel monitors the NT Kernel for unauthorized modifications of executable code, which is one of the main purposes of the Secure Kernel.

We use the term 'Windows Kernel' to refer to the Windows operating system, which includes the 'NT Kernel' and the 'Secure Kernel'. The NT Kernel is the kernel code that is executed in VTL0 ring 0 (see Section 2.2.2). The Secure Kernel is the kernel code that is executed in VTL1 ring 0. See Figure 4 for a visual overview of the VTL and ring levels.

2.1 Hardware Virtualization

The Windows Kernel utilizes hardware virtualization to strengthen the operating system's security. Starting from Windows 10 and Windows Server 2016, Microsoft introduced Virtual Secure Mode (VSM) which uses hardware virtualization to allow new security boundaries within the operating system software [24]. The purpose of VSM is to limit the impact of malware or exploits even when the NT Kernel is exploited by an attacker since the NT Kernel itself is excluded from the chain of trust. Therefore, hardware virtualization forms the foundation on which new security guarantees are built.

In this section, we explain how hardware virtualization works and is utilized to introduce new security boundaries within the Windows Kernel, which seperates the 'normal world' (where the NT Kernel is executed) and the 'secure world' (where the Secure Kernel is executed). In Section 2.2 we will then further look at how Hyper-V uses hardware virtualization to introduce Partitions and Virtual Trust Levels (VTLs).

2.1.1 Virtual Mode Extensions

Hardware virtualization relies on the CPU supporting Virtual Mode Extensions (VMX), which was introduced by Intel in 2005. VMX defines processor-level support for virtual machines on IA-32 processors. This VMX set is used to control entering and leaving virtualization mode, which establishes a security boundary between the virtual machines and the host system. The operating system can register a Virtual-Machine Monitor (VMM), which is responsible for managing the virtual machines. In Windows, the VMM is named the hypervisor, and for Linux, it is named Kernel-based Virtual Machine (KVM). Each Virtual Machine (VM) that is managed by the VMM is named 'Guest software'. A Guest software supports a stack consisting of an operating system and application software. Guest software is the term used by Intel to refer to a VM, Microsoft refers to a VM as a 'Partition'. The VMM runs at a higher privilege level than the Guest software, which results in the VMM having more control over the system and direct access to the hardware. Each VM operates independently of other VMs and shares the hardware resources (processor(s), memory, storage, I/O, etc.) with other VMs. A VM

is executed with reduced privileges so that the VMM can retain control of the platform resources [9].

Similar to the concept of 'rings' in an operating system, which represent different levels of privileges between programs and the kernel, VMX operations are used to distinguish between the levels of privileges within hardware virtualization. There are two modes of operation (privilege levels):

1. VMX root operation: this is where the VMM will run.
2. VMX non-root operation: this is where the Guest software will run.

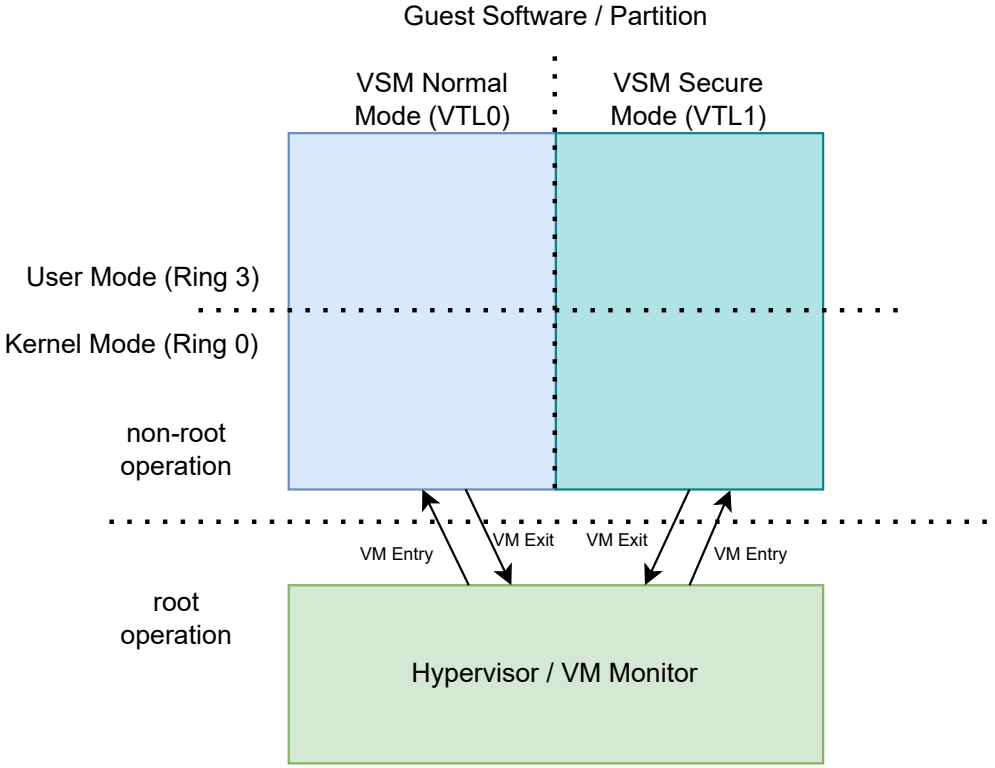


Figure 1: Guest software and Virtual-Machine Monitor overview

Transitioning from root operation to non-root operation is done by 'VM entries', while transitioning from non-root operation to root operation is done by 'VM exits'. These transitions are visualized in Figure 1. Note that the concepts of VSM and VTL are explained in Section 2.2.2. VMX transitions rely on a data structure named "Virtual-Machine Control Structure (VMCS)", which is explained in Section 2.1.2.

The VMM is in control of the processor resources and therefore the VMX non-root operation is restricted in resources and execution. Certain operations and events can cause the VM to exit to the VMM. There is also an explicit instruction set that will always cause a VM exit, known as the 'VMX instructions'. These VMX instructions are used by the Windows Kernel to perform calls to the VMM.

2.1.2 Virtual-Machine Control Structure

Transitions into and out of Virtual Mode Extensions (VMX) non-root operation (VM entries and VM exits, see Figure 1), as well as processor behavior in VMX non-root operation are managed using Virtual-Machine Control Structure (VMCS) data structures [9]. The VMCS serves as an important data structure that manages the processor state during transitions between VMX root and non-root operations. It is used to preserve and to restore the hypervisor's state correctly during these transitions. This state is saved or restored depending on performing a VM Entry or VM Exit. A VMM can use a different VMCS for each virtual processor per VM [9] since it contains the processor state when the VM Exit is executed.

The data of the VMCS is divided into six groups [9]:

1. Guest-state area: on a VM exit the processor state (of the guest) is saved into the guest-state area and loaded again on a VM entry.
2. Host-state area: on a VM exit the processor state (of the host) is loaded from the host-state area and stored again on a VM entry.
3. VM-execution control fields: the VM-execution control fields define the rules and restrictions for processor behavior during VMX non-root operation. These fields also influence the conditions that trigger a VM exit, allowing the VMM to enforce specific policies on the guest's execution.
4. VM-exit control fields: fields related to controlling VM exits.
5. VM-entry control fields: fields related to controlling VM entries.
6. VM-exit information fields: fields that receive the cause and the nature of a VM exit when a VM exit is triggered. The VMM uses this information to determine what should be done after a VM exit has been executed.

2.1.3 Address Translation & Second Level Address Translation

Second Level Address Translation (SLAT) is a hardware-assisted technology that enables efficient mapping of virtualized memory used by Guest software to the physical memory of the host system. To understand how SLAT works, it is helpful to first review the concept of address translation in Windows, as SLAT builds upon similar principles.

2.1.3.1 Address Translation

In Windows, each process operates within its own virtual memory space, allowing processes to use the same virtual addresses without conflict. Address translation is the mechanism that maps these virtual addresses to unique physical memory addresses, ensuring isolation and efficient memory management. Address translation relies on a hierarchical structure of page tables to map virtual addresses to physical addresses. In a 64-bit Windows system, this process involves the following four types of page tables:

1. Page Map Level 4 (PML4) table
2. Page Directory Pointer (PDP) table
3. Page Directory Table (PDT)
4. Page Table (PT)

Every 64-bit virtual address consists of four times 9 bits of offset in the four corresponding page tables and the lower 12 bits are used for the page offset. The upper 16 bits

are always zero. 4-Level Paging starts the page walk at the PML4 table, for which the address is stored in the CR3 CPU register of the process that performs the access. A visual overview of resolving a virtual address to a physical address using the four page tables can be found in Figure 2.

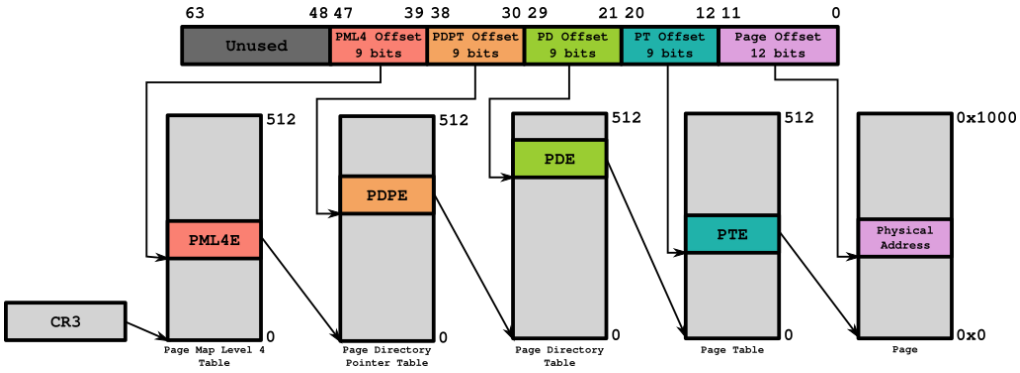


Figure 2: "4 Level Page Mode Overview" from [59]

2.1.3.2 Second Level Address Translation

SLAT can be enabled through the VM-execution control fields (see Section 2.1.2), which introduces an extra layer of paging mechanism. The implementation of SLAT developed by Intel is called Extended Page-Table (EPT). The implementation of SLAT developed by AMD is called Rapid Virtualization Indexing (RVI). On ARM processors, this implementation is known as 'Stage-2 page-tables'. Since the hardware we use for our research contains an Intel CPU, we only consider EPT.

The idea of address translation is also implemented at the level of hardware virtualization. For hardware virtualization, the virtual addresses correspond to the memory addresses of the Guest software and the physical addresses correspond to the physical memory of the hardware. As a result, the 'physical memory' of the Guest software is still virtualized due to SLAT.

The EPT translation mechanism can be either a 4-level EPT or a 5-level EPT [10]. The difference between 4-level and 5-level EPT is that 4-level EPT can use 48-bit guest physical addresses while 5-level EPT can use up to 57 bits, and 4-level EPT has a page-walk length of 4 while 5-level EPT has a page-walk length of 5. Since a 5-level EPT is rarely used, we focus on the 4-level EPT translation mechanism.

The 4-level EPT mechanism has four different address spaces:

1. Guest Virtual Address (GVA): virtual addresses used by processes within the Guest software operating system.
2. Guest Physical Address (GPA): physical addresses seen by the operating system of the Guest software.
3. System Virtual Address (SVA): virtual addresses seen by the hypervisor.
4. System Physical Address (SPA): physical addresses which correspond to the actual physical memory of the hardware.

These address spaces work together to ensure that the hypervisor can manage memory access while maintaining isolation between the guest and host systems.

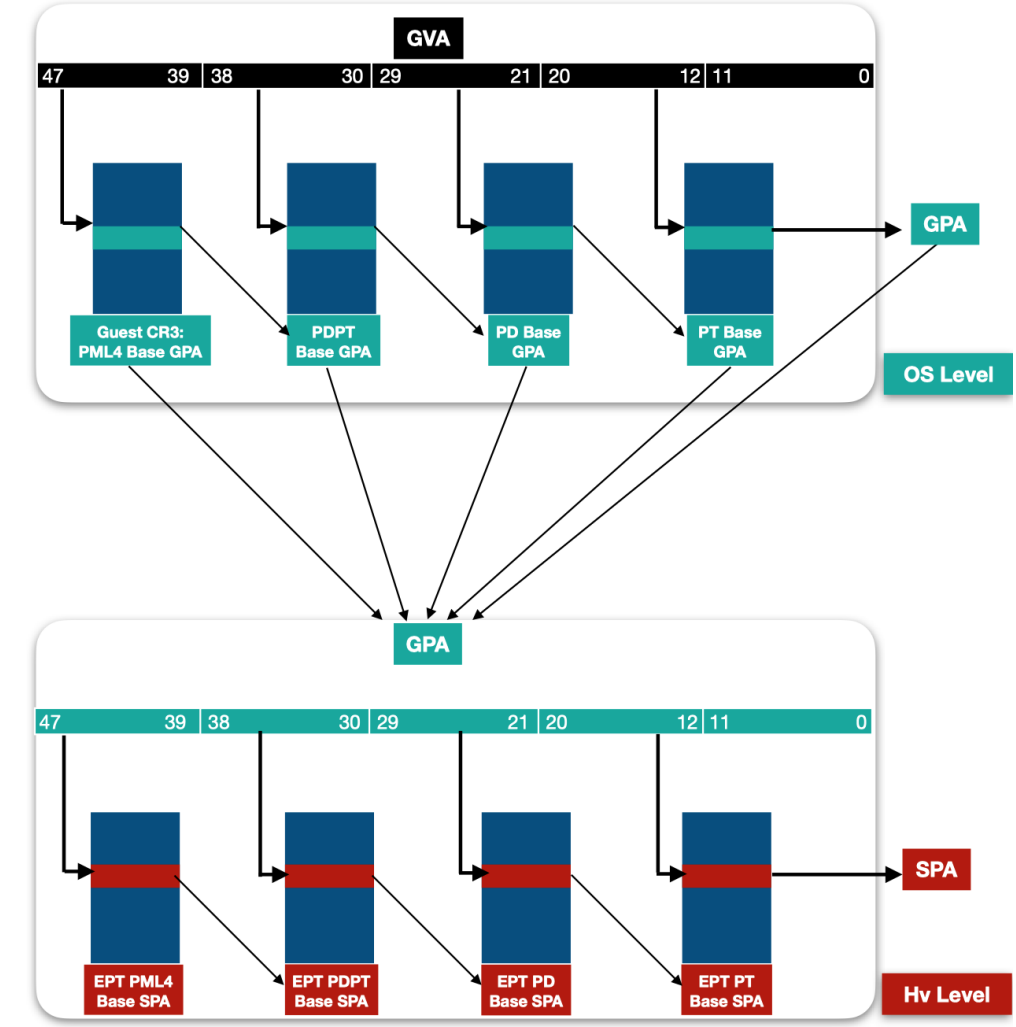


Figure 3: Address Translation from GVA to SPA, from [47]

The SVA is the virtual memory address space as seen by the hypervisor (the VMM running in VMX root mode) which directly runs on the hardware. Therefore SLAT is not used for SVA to SPA and instead a normal page-table walk is used starting from the PML4 table found in the CR3 CPU register. Translation from a GPA to a SPA uses the same mechanism as normal address translation except that the tables used in the address translation are different. The translation from GPA to SPA begins with the EPT PML4 table. Unlike traditional address translation, which uses the CR3 CPU register to reference the page table, EPT relies on a special pointer called the 'Extended Page Table Pointer (EPTP)'. This pointer is stored in the VMCS structure of the virtual processor (see Section 2.1.2) and enables the hypervisor to manage memory translation for virtual machines. Resolving a GVA to a SPA therefore consists of walking 2 levels of page-tables, which is summarized in Figure 3.

Due to the separation of these address spaces, the hypervisor is able to define access rights for each SPA page. This means that the hypervisor is allowed to overrule the

defined memory privileges as defined by the Guest software. For instance, the hypervisor may map a memory page as read-write. If an attacker exploits the NT Kernel to change the page's privileges to read-write-execute, the hypervisor will detect this unauthorized change. When the attacker attempts to execute code on the page, the hypervisor will block the operation, triggering a VM exit and preventing the exploit from succeeding. Furthermore, the EPT is used to ensure that the NT Kernel cannot access restricted pages which belong to the Secure Kernel.

2.2 Hyper-V

Hyper-V is a bare-metal hypervisor developed by Microsoft for the Windows operating system. Hyper-V leverages hardware virtualization technologies (see Section 2.1) to create isolated environments for running virtual machines and improving system security. Hyper-V uses the Virtual-Machine Monitor (VMM) (see Section 1), the Virtual-Machine Control Structure (VMCS) data structures (see Section 2.1.2) as well as Second Level Address Translation (SLAT) (see Section 2.1.3) to create isolated 'Partitions' (see Section 2.2.1). Hyper-V can be used for traditional Virtual Machines (VMs) to run Linux or another Windows OS. In addition to running virtual machines, Hyper-V provides security features within the Windows OS through Virtualization Based Security (VBS). For instance, without VBS, malware with administrative privileges can dump the memory of the `lsass.exe` process to steal sensitive credentials. When VBS is enabled, Hyper-V protects critical processes like `lsass.exe` by moving sensitive information from `lsass.exe` to `LSAISO.exe` (which is a Trustlet running in Virtual Trust Level (VTL)1), preventing unauthorized access and protecting user credentials. To understand how Hyper-V achieves isolation, we explore its core architectural components, starting with the concept of 'Partitions' (Section 2.2.1) and afterwards the concept of VTLs (Section 2.2.2).

2.2.1 Partitions

In Hyper-V, isolation is achieved through 'Partitions', which are logical units created and managed by the hypervisor. Each Partition represents an isolated environment where operating systems can execute independently [20]. Partitions can be seen as the terminology Microsoft uses for Virtual Machines (VMs), which Intel calls Guest software. Partitions do not have direct access to the physical processor or physical memory. Instead, they run in an isolated environment where they have a virtual view of the processors and their memory is virtualized and private for each guest partition. This is made possible due to Virtual Mode Extensions (VMX) (see Section 2.1.1) and Second Level Address Translation (SLAT) (see Section 2.1.3).

Hyper-V defines two types of Partitions:

1. Root Partition (also called 'Parent Partition'): This is the primary partition that manages machine-level functions such as device drivers, power management, and device hot addition/removal. It is the only partition with direct access to physical memory and devices [16].
2. Child Partitions: These are secondary partitions that host guest operating systems. Unlike the Root Partition, Child Partitions do not have direct access to physical memory or devices. Instead, they rely on the Virtual Machine Bus (VMBus) or the hypervisor to access these resources [16].

The Root Partition is responsible for creating and managing Child Partitions, which host operating systems such as Linux or Windows. The Root Partition runs the Windows OS, which can be considered the 'host' OS which manages Hyper-V. The Root Partition

uses Hypercalls (see Section 2.3.2) to create and manage Child Partitions. These Child Partitions operate with lower privileges compared to the Root Partition and rely on the hypervisor to mediate access to physical resources such as memory and devices. By isolating resources and managing access through the hypervisor, Partitions form the foundation of Hyper-V's ability to securely run several operating systems on the same hardware. In the next section, we explore how Hyper-V leverages these partitions to implement security features.

2.2.2 Virtual Trust Levels

Virtual Secure Mode (VSM) is a set of hypervisor capabilities offered to the Root Partition and Child Partitions that leverages hardware virtualization of the CPU (see Section 2.1) which enables the creation and management of new security boundaries within operating system software [24]. VSM is used to introduce a security boundary within a Partition. Windows security features such as Device Guard, Credential Guard, virtual TPMs and 'shielded' VMs are facilitated through VSM. VSM allows operating systems in the Root Partition and Child Partitions to create isolated memory regions for storage and processing of system security assets. This is possible since the hypervisor runs at a higher privilege level than the Partitions. Therefore, the hypervisor has control over the hardware resources and permissions of memory regions, which are used to introduce a new security boundary. Even code running in the highest privileges of the Partition, for example code running in "Ring 0" of the NT Kernel, is unable to access isolated memory regions. This new security boundary, provided by 'Virtual Trust Levels (VTLs)', enables the hypervisor to enforce strict isolation between different privilege levels within a Partition. This ensures that even highly privileged code running in lower levels, such as the NT Kernel in VTL0, cannot access memory or resources allocated to higher levels like VTL1.

VSM achieves isolation by leveraging VTLs, which are managed on both a per-virtual processor and per-partition basis [24]. VTLs are organized hierarchically, with higher levels having more privileges than lower levels. For example, VTL0 is the least privileged level, while VTL1 has greater privileges and can enforce restrictions on VTL0. Although up to 16 levels are supported, VSM uses only two levels: VTL0 and VTL1. The hypervisor enforces memory access protections for each VTL, ensuring that memory regions allocated to higher VTLs cannot be accessed or modified by lower VTLs. These protections are stored in the physical address space of the partition and are managed exclusively by the hypervisor. This mechanism allows secrets to be securely stored in higher VTLs. For example, if a secret is stored in VTL1 and VTL0 is later compromised, the secret remains protected because VTL0 cannot access memory allocated to VTL1.

To enforce isolation provided by VTLs, the hypervisor uses several mechanisms [24]:

1. Memory Access Protections: Each VTL has its own Guest Physical Address (GPA) access protections. Software running in a VTL can only access memory within the VTL itself and follows the configured memory access protections.

2. Virtual Processor State: The processors are virtualized to the VTL and maintained per-VTL-state. Virtual Processor States of lower VTLs cannot access the Virtual Processor State of higher VTLs, but higher VTLs can access the Virtual Processor State of lower VTLs.

3. Interrupts: The interrupt subsystem is also separated, similar to the Virtual Processor State. Lower VTLs can therefore not interfere with processing interrupts of higher VTLs, but higher VTLs can interfere with processing interrupts of lower VTLs.

4. Overlay Pages: For each VTL there is a special 'overlay page' which is used for Hypercalls (see Section 2.3.2) [24].

By leveraging VTLs, Hyper-V introduces a new security boundary that isolates critical components of the operating system. One such component is the Secure Kernel, which operates in the higher-privileged VTL1 environment. The Secure Kernel will be discussed in Section 2.3.

2.3 The Secure Kernel

In Virtual Trust Level (VTL) environments, the normal NT Kernel of the Windows operating system runs in the low-privileged VTL0. The Secure Kernel runs in an isolated environment named 'VTL1' which has higher privileges than VTL0 [71]. The Secure Kernel lacks several components such as the I/O manager and the power manager [3], which minimizes its attack surface and improves security by limiting potential vulnerabilities. Therefore, the Secure Kernel cannot function independently and needs the lower-privileged NT Kernel in order to operate. An overview of the NT Kernel, the Secure Kernel and the corresponding privilege levels can be found in Figure 4.

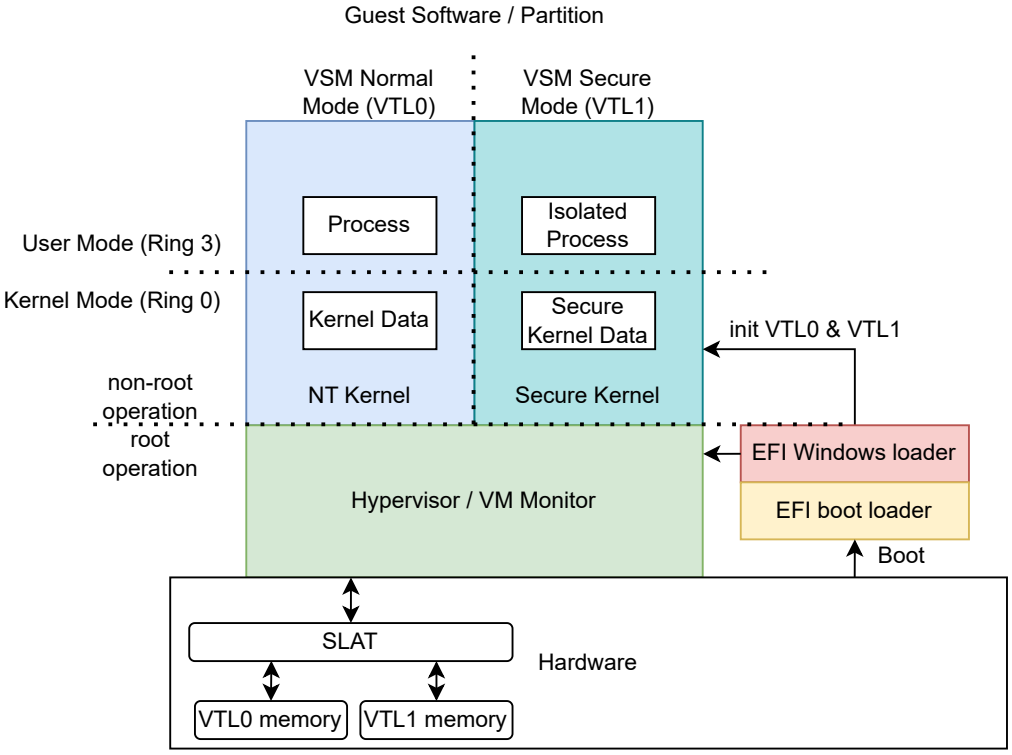


Figure 4: Isolated User Mode (IUM) Architecture

The Secure Kernel consists of the `securekernel.exe` binary which can be found in the Windows operating system at `C:\Windows\System32\securekernel.exe`. The NT Kernel consists of the `ntoskrnl.exe` binary which can be found in the Windows operating system at `C:\Windows\System32\ntoskrnl.exe`. Symbols for the Secure Kernel and NT Kernel binaries can be downloaded from the Microsoft Public Symbol Server. The `securekernel.exe` binary uses a consistent naming convention for its functions and global variables, making it easier to identify their purpose. Table 1 provides an overview

of the most important function prefixes and their corresponding functionalities.

Prefix	Description
Skps	Process support functions.
Skob	Object manager functionality.
Skmm	Memory manager functionality.
Sk, Ski, Skp, Ske	Secure kernel functionality.
Ium, Iump	Isolated user mode functionality.
Nk	Alternate prefix for functionality imported from the normal kernel.
Skpg	Secure Kernel Patch Guard (SKPG) functionality (see Section 2.3.3)

Table 1: "Kernel Function Prefixes and Their Descriptions" from [3]

In addition to its kernel-level functionality, the Secure Kernel also supports specialized user-mode processes called 'Trustlets', which operate within the VTL1 environment to improve security. Trustlets are discussed in Section 2.3.1.

2.3.1 Trustlets

Like the low-privileged NT Kernel, the Secure Kernel is also able to run processes. These processes are executed in User Mode (ring 3) of VTL1 and are called 'Trustlets'. Trustlets are responsible for completing system calls by marshalling them over to the NT Kernel in VTL0 ring 0. At the time of writing, there is no way to create or load third-party Trustlets. Therefore, there are a fixed number of available Trustlets [30]:

1. `LSALSO.exe`: the Trustlet responsible for credential and key guard
2. `vmsp.exe`: the Trustlet responsible for the vTPM
3. `Unknown:` the Trustlet responsible for vTPM key enrollment
4. `Biolso.exe`: the Trustlet responsible for secure biometrics
5. `Fslso.exe`: the Trustlet responsible for secure frame server

As an example, we will look into the `LSALSO.exe` Trustlet. When Virtual Secure Mode (VSM) is enabled, the Local Security Authority (LSASS) process runs together with a Trustlet '`LSAISO.exe`'. The LSASS process manages the local system policy, user authentication, and auditing while handling sensitive security data such as password hashes and Kerberos keys [19]. A well-known technique for credential stealers such as Mimikatz [29] is to dump the memory of LSASS in order to exfiltrate sensitive data. Leveraging VSM, the secrets are not stored anymore in the `LSASS.exe` process but instead in a `LSAISO.exe` (LSA Isolated) Trustlet running in VTL1 ring 3. `LSAISO.exe` communicates with `LSASS.exe` running in VTL0 through an RPC channel. In Figure 5 a visual overview can be found of the processes and privilege levels of LSASS. This protection mechanism, where sensitive data is isolated in VTL1 using VSM, is known as 'Credential Guard'. Credential Guard is only available when hardware supports VBS and when the Windows operating system is configured with a certain Windows Enterprise or Windows Education license [12].

2.3.2 Hyper-V Hypercalls

The Secure Kernel relies on the NT Kernel for certain functionalities it lacks by design. To enable this dependency, a communication channel is required between VTL0 (where the NT Kernel operates) and VTL1 (where the Secure Kernel operates). Communication

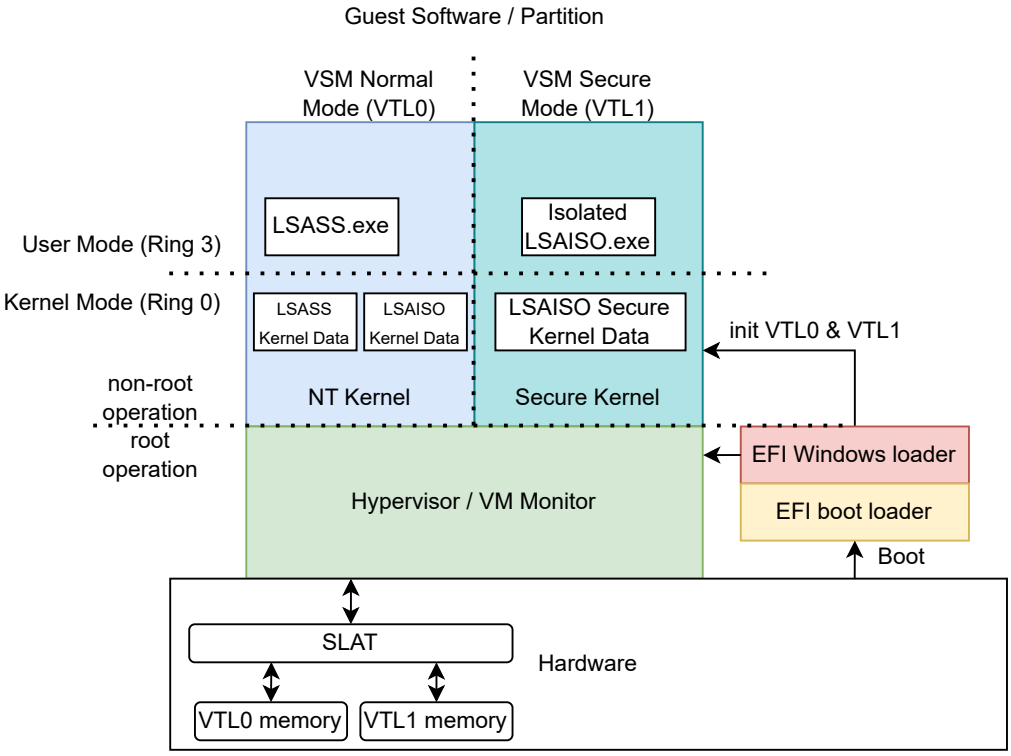


Figure 5: LSASS Trustlet Design

between VTL0 and VTL1 is done using 'Hypercalls' by making use of the VM Entry and VM Exit instructions, as well as the Virtual-Machine Control Structure (VMCS) data structures. Communication to higher privileged VTLs is only allowed from the most privileged guest processor mode (which is ring 0, kernel privileges). Hypercalls function as a communication mechanism, similar to system calls, allowing the NT Kernel in VTL0 to interact with the Secure Kernel in VTL1. Since Hypercalls switch between VTL privileges, it must be performed using Virtual Mode Extensions (VMX) operation instructions (see Section 2.1.1). Specifically, the `VMCALL` instruction is used with the corresponding Hypercall number in register `RCX` [7]. The `VMCALL` instructions are stored on a special 'Hypercall page', which is mapped by VTL1 into the memory space of VTL0. This page acts as a trampoline, enabling transitions to higher-privileged VTLs. Hypercalls use different calling conventions based on the value of the 'Fast' flag and the specific Hypercall being invoked.

When the Fast flag is set to zero, the register mapping is used as described in Table 2. When the Fast flag is set to one, the register mapping is used as described in Table 3. When the Fast flag is set to one, the register contains the direct input parameter value, otherwise a pointer is expected pointing to a buffer storing the input value.

To ensure secure communication, several checks are performed on the supplied Guest Physical Address (GPA) parameters:

1. The GPA pointers must be 8-byte aligned.
2. The input and output parameter list cannot overlap or cross page boundaries.
3. The input and output pages are expected to be GPA pages and not "overlay"

pages, otherwise the input parameter list is undefined.

4. The supplied input GPA is mapped and readable by the calling partition.
5. The supplied output GPA is mapped and writable by the calling partition.

x64	**x86**	**Contents**
RCX	EDX:EAX	Hypercall Input Value
RDX	EBX:ECX	Input Parameters **GPA**
R8	EDI:ESI	Output Parameter **GPA**

Table 2: "Register mapping for Hypercall inputs when the Fast flag is set to zero" from [70]

x64	**x86**	**Contents**
RCX	EDX:EAX	Hypercall Input Value
RDX	EBX:ECX	Input Parameters
R8	EDI:ESI	Output Parameter

Table 3: "Register mapping for Hypercall inputs when the Fast flag is set to one" from [70]

Field	**Bits**	**Information Provided**
Call Code	15-0	Specifies which Hypercall is requested.
Fast	16	Specifies whether the Hypercall uses the register-based calling convention: 0 = memory-based, 1 = register-based.
Variable header size	26-17	The size of a variable header, in QWORDS.
RsvdZ	30-27	Must be zero.
Is Nested	31	Specifies the Hypercall should be handled by the L0 hypervisor in a nested environment.
Rep Count	43-32	Total number of reps (for rep call, must be zero otherwise).
RsvdZ	47-44	Must be zero.
Rep Start Index	59-48	Starting index (for rep call, must be zero otherwise).
RsvdZ	63-60	Must be zero.

Table 4: Hypercall Fields and Their Meaning from [17]

For Hypercalls with call codes above 0x8000, the 'Extended Hypercall Interface' is used [17]. This interface allows larger input parameters (up to 122 bytes) by utilizing 6 XMM registers [48].

Each Hypercall is associated with a 'Secure System Call Number (SSCN)' in the Secure Kernel. The SSCN acts as an identifier for specific functionalities, which are executed through a switch statement in the Secure Kernel. However, the Hypercall Input Value is not simply an ID for the SSCN, but it encodes more information using several flags. There are two different ways of calling a Hypercall:

1. Simple: perform a single operation with a fixed-size set of input and output parameters
2. Rep (repeat): perform multiple operations at once, with a list of fixed-size input and/or output elements [70].

The meaning of all Hypercall fields can be found in Table 4. It is also possible for a Hypercall to have a variable amount of input header data. In such a case, the Hypercall has a fixed-size input header and additional header input that is of variable size.

The SSCN for each Hypercalls is handled by the `securekernel.exe` binary in the `IumInvokeSecureService` function. There are a total of 134 different SSCNs implemented in the `IumInvokeSecureService` function. A detailed list of SSCN values and their corresponding executed functions can be found in Appendix A.

2.3.3 Secure Kernel Patch Guard

As mentioned in Section 2.1, Virtual Secure Mode (VSM) is introduced to limit the impact of malware or exploits even when the NT Kernel is exploited by an attacker. Due to the new security boundaries introduced with Virtual Trust Level (VTL), the Secure Kernel running in VTL1 can manage the NT Kernel running in VTL0, while the NT Kernel cannot manage the Secure Kernel due to the VTLs. Secure Kernel Patch Guard (SKPG), also known as 'HyperGuard', is the practical implementation of monitoring the NT Kernel for unauthorized changes in certain parts of the Windows operating system (for example, executable code). A similar system to this is 'PatchGuard', which is a similar anti-tampering system for the NT Kernel but runs in VTL0 and is thus less effective since it can also be tampered with by attackers exploiting the NT Kernel. The main difference between SKPG and PatchGuard is that SKPG cannot be tampered with by attackers targeting the NT Kernel, since it is running in VTL1, while PatchGuard can be tampered with since it is running in VTL0, which is also where the exploitation by attackers is happening [64].

SKPG is implemented in the Secure Kernel, and functions related to SKPG start with the prefix `Skpg`. During boot initialization, `SkpgInitSystem`, `SkpgAllocateContext`, and `SkpgConnect` together allocate a SKPG context structure in the Secure Kernel ring 0 memory and store information related to the verification procedure of SKPG in the SKPG context structure. The SKPG context structure is an undocumented structure, and thus the purpose of the fields in the SKPG context structure has been discovered through reverse engineering [65]. Important properties of the SKPG context structure are the following [64]:

1. `SkpgPatchGuardCallback`: a callback pointer pointing to the Secure Kernel function `SkpgPatchGuardCallbackRoutine`, called each time SKPG performs checking.
2. `SkpgPatchGuardTimer`: A timer object which will invoke the `SkpgPatchGuard TimerRoutine` callback at random intervals to perform the SKPG checks.
3. Intercept function pointers which will be called if SKPG detects certain operations done by the hypervisor.
4. `TimerRoutine` (offset 0x250): A function pointer which points to the function `SkpgHyperguardTimerRoutine`, which will call the function pointer stored in `Runtime CheckRoutine`.
5. `RuntimeCheckRoutine` (offset 0x268): A function pointer which points to the function `SkpgHyperguardRuntime`, which verifies the SKPG context structure.

Since SKPG is a system that detects and prevents unauthorized changes to certain parts of the Windows operating system, it is important that SKPG itself is also protected from unauthorized changes. For example, if the `SkpgPatchGuardCallback` pointer is modified in memory and updated to point to a function that does nothing, then the SKPG checking mechanism is effectively disabled. The SKPG context structure tries to protect its own integrity by using the fields `TimerRoutine` and `RuntimeCheckRoutine`. A timer is set with a randomized interval to execute the function pointer stored at `TimerRoutine`, which will invoke the Secure Kernel function `SkpgHyperguardTimerRoutine`. This Secure Kernel function will trigger the Secure Kernel function `SkpgHyperguardRuntime`, which performs the actual SKPG context structure verification. The function `Skpg HyperguardRuntime` will also set a new timer with a randomized interval for the next verification check.

The SKPG context structure stores much more information for various SKPG-related functionalities. Most of this information is out of scope for this master thesis. More information about SKPG can be found in [64], [65], and [66].

3 Re-discovering N-day vulnerabilities

In Section 2 we focussed on the background information related to virtualization, Hyper-V and the Secure Kernel. In this section we will focus on the background information related to rediscovering n-day vulnerabilities. The background information in this section is applied in Section 5.

For this master thesis, we will perform analysis on n-day vulnerabilities in the Secure Kernel. These n-day vulnerabilities will be found using a specialized version of 'binary diffing', namely 'patch diffing', which will be discussed in Section 3.1. N-day vulnerabilities are security issues that are already reported and fixed, and for which a patch is available. Most of the time, there is a description online that clarifies the security issue and its impact. In the case of the Windows operating system, this is handled by the Microsoft Security Response Center. A security issue is assigned a CVE and receives a scoring for several categories of impact, such as the 'Attack Complexity' or impact on confidentiality, integrity, or availability. It is also common for the Microsoft Security Response Center to add a 'Weakness' to the CVE, which states if it is a buffer overflow, heap overflow, double free, or other type of memory corruption. Using this information and the patches included in Windows update files, it is possible to re-discover the vulnerability.

Section 3.2 describes how Windows updates can be analyzed for patch diffing.

3.1 Binary Diffing & Patch Diffing

The term 'binary diffing' (or short 'bindiffing') refers to the process of discovering differences between two binary files. The two binary files are usually referred to as primary and secondary. For binary files, it is important that the two binaries use common artifacts to find similarities between them. For example, the two binary files should have strings or functions in common to compute the similarity between them. Furthermore, if a different compiler (version) is used between the primary and secondary binary, then it may be the case that the order of functions or data is changed. The tooling used for computing the difference between the binary should take such differences into account. Therefore, binary diffing is commonly performed on the function level rather than the whole binary. Differences can be computed between the disassembly of two similar functions, but also between the pseudocode retrieved from the decompilation of two functions. The pseudocode is often retrieved from decompiler tools such as IDA Pro, Ghidra, or Binary Ninja. Besides differences between the assembly and pseudocode of functions, there can also be differences between the call graph or cross-references of functions [57].

For binary diffing, there are three tools commonly used:

1. The binary exporter
2. The binary diffing tool
3. The decompiler tool

These three tools are used in each step of the binary diffing tool. Binary diffing is commonly done in three steps:

1. The information about the two binaries is exported to a format that can be processed by the diffing tool. This tool is called the 'Binary Exporter'.
2. The 'binary diffing tool' takes the result of the binary exporter for two binaries and computes the similarity between the two binaries using the given exported

information.

3. When the difference is found by the binary diffing tool, the binary diffing tool may use an external decompiler tool to generate pseudocode of the functions that have changed and show the difference between the old function and the updated function.

Note that the binary diffing tool can also be integrated into a decompiler tool (as a plugin to the decompiler as opposed to an external invocation to the decompiler), for which the binary diffing tool directly reads out the information of the decompiler to compute the differences between two binaries.

Two examples of binary exporters are:

1. BinExport [35]
2. Quokka [58]

The output of BinExport can be read by BinDiff [34]. Quokka is able to export to a file format which can be read again by the Python library integrated into Quokka.

Two examples of tools that are integrated into the decompiler are:

1. Ghidriff: Ghidra Binary Diffing Engine [8]
2. Diaphora: IDA Pro plugin for binary diffing [39]

For the Exploitability Assessment in Section 5, we will make use of Diaphora. The reason for this is that it makes use of the decompilation of IDA Pro which is the reverse engineering software used in this master thesis to decompile the assembly.

3.1.1 Patch Diffing

'Patch diffing' is binary diffing applied to security patches. The goal of patch diffing is to find the patched code which should fix a security bug. By finding this code, it is trivial to compare the code before the patch and after the patch and therefore deduce the security bug.

In order to perform patch diffing, it is best to have the binary just before the patch and just after the patch available. This minimizes the changes between the two binaries and therefore makes it less challenging to compare the binaries. The best case for patch diffing is that the only changed code between the two binaries is related to the security bug and all other code has a 100% similarity. The worst case is when obfuscation is applied to the updated files, which drastically modifies the code between each update making it challenging to perform binary diffing.

For this master thesis, we used patch diffing to security updates of the Secure Kernel. Most of the Secure Kernel code is implemented in a single binary, the `securekernel.exe` binary (see Section 2.3). Microsoft is known for publishing security fixes on 'Patch Tuesday', the second Tuesday of each month. This gives a great opportunity for performing patch diffing, since there is a dedicated update only addressing security bugs.

3.2 Extracting Windows Updates

For each security update that Microsoft publishes, a Knowledge Base (KB) article is created. This article contains information about the update, including the files that are updated and the vulnerabilities that are fixed. The KB article also contains a link to download the update package.

The update package that is downloaded is usually in an MSU (Microsoft Standalone Update) format, which is an archival format. This MSU archive contains multiple CAB (Cabinet) files, which is again an archival format that contains some files such as the manifest, security catalogs, and the new binary updates [46]. For patch diffing, we are interested in the new binary files. However, the bundled binary files are not correctly formatted executables. Instead, they serve as a delta update file for which a special MSDELTA format is used to store updates in. There are three types of delta files, depending on which folder they are stored inside the CAB file [76]:

1. Forward differentials (f): brings the base binary (.1) up to that particular patch level.
2. Reverse differentials (r): reverts the applied patch back to the base binary (.1).
3. Null differentials (n): a completely new file, only compressed; apply to an empty buffer to get the full file.

Using the MSDELTA API, it is possible to read out these delta update files and apply them in the correct order to get a valid executable. This process is in detail described in [76].

However, there is an easier way to obtain a valid updated executable from a Windows update. The Microsoft Symbol Server not only stores symbols for portable executable files, but also the executables themselves. It is possible to retrieve this executable from the Microsoft Symbol Server by having knowledge of the following three pieces of information [28]:

1. The file name
2. The link timestamp / reproducible build identifier
3. The image size

By reading the manifest file from the CAB archives inside the MSU files, and taking advantage of the VirusTotal API [74], it is possible to compute these three pieces of information [42] and therefore create a download link to the executable. This data has been conveniently displayed in the form of a website called 'Winbindex' [43]. Winbindex lists download links to the Microsoft Symbol Server for all the executables for each version inside the Windows operating system. This website is updated automatically to take new updates into account.

4 Debugging Setup

In order to better understand the Secure Kernel and the bugs we found using patch diffing (see Section 3.1.1), we need to use a debugger attached to the Secure Kernel. We explain the steps taken to create our Secure Kernel debugging environment, including the challenges encountered and the modifications made to our setup to overcome these challenges.

First, a setup is described that uses physical hardware (a Windows 11 laptop) to debug the NT Kernel, which is described in 4.1. The second step is to debug the Secure Kernel, which we first attempted by extending the physical debugging setup of Section 4.1 to also include Secure Kernel debugging through the hypervisor, which is described in Section 4.2.1. However, we then concluded in Section 4.2.1 that this setup seems to be impossible in order to correctly debug the Secure Kernel due to physical memory restrictions and we moved to a QEMU/KVM virtualized setup which is described in Section 4.2.2. As an addition to the setup, a small GDB plugin has been developed to perform virtual to physical address translation (as described in Section 2.1.3) which is described in Section 4.2.4.

Section 4.3 shortly mentions alternative setups which are not used in this research, but can be considered in case different hardware or software is available. For example, Section 4.3.1 discusses LiveCloudKd which is a debugging setup that can be used when the host operating system is Windows.

4.1 NT Kernel Debugging Setup

This section describes the physical setup used for NT Kernel debugging. The purpose of this section is to make it easier to reproduce this research or perform future research on the NT Kernel. The setup is visualized in Figure 6. The initial setup time without prior experience of attaching a debugger to a bare-metal Windows laptop was about five hours including troubleshooting. The Windows laptop is the debuggee, the machine that is being debugged.

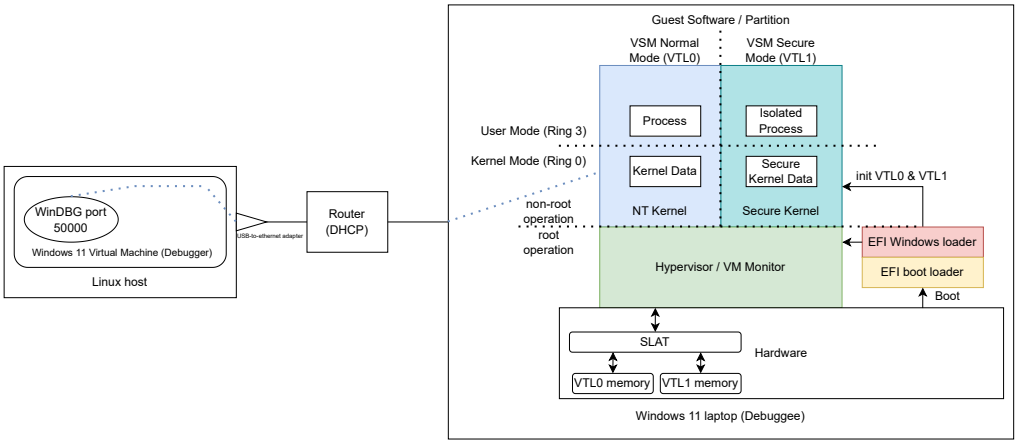


Figure 6: Visualization of the NT Kernel Debugging Setup

The outcome of this section is to have a debugger running on a machine attached to another laptop in order to debug the NT Kernel on the other laptop (the laptop is the target for debugging, the 'debuggee'). Each NT Kernel can be configured to support debugging capabilities through the Windows debugger 'WinDBG'. This is useful

for the development of kernel modules, troubleshooting of bugs and inspecting certain undocumented components of the NT Kernel.

The following hardware is used for the kernel debugging setup:

1. A Linux laptop with Ubuntu 22.04 LTS installed
2. 12th Gen Intel(R) Core(TM) i7-12700H CPU
3. Two ethernet cables
4. USB-to-ethernet adapter
5. A Windows laptop with Windows 11 Version 22H2 for x64-based Systems installed

Note that there are special hardware requirements for the ethernet port used by the debuggee. The debugger can be used over serial or ethernet, but ethernet has significantly better performance than serial. Therefore, we used debugging over an ethernet network cable. For this to work, the target device must have a supported network interface card (NIC). The supported NICs for Windows 11 can be found at the corresponding support page of Microsoft[1]. The supported NICs for Windows 10 can be found at the corresponding support page of Microsoft[2]. Note that generally this means that a device should have a hardware ethernet port. We do not know if a USB-to-ethernet adapter is also supported during debugging since the debuggee laptop already has a physical ethernet port which is supported and therefore we recommend to use a debuggee laptop with a physical ethernet port.

We use bare-metal hardware instead of a virtual machine for the debuggee since the Secure Kernel uses hardware virtualization which we prefer not to virtualize. Furthermore, nested virtualization should be done in case of virtualization which may introduce new issues. For the debugger we use a VM with Windows 11 installed. The debugger VM can also be a bare-metal Windows 11 device, but since the kernel debugging setup was already installed on the Windows 11 virtual machine we used that instead.

Using QEMU/KVM, we configured a USB-passthrough to allow the USB-to-ethernet adapter to be passed through to the debugger VM since we need to have an ethernet adapter available within the debugger VM and our Linux laptop does not have an extra physical ethernet port. Both the ethernet cable from the USB-to-ethernet adapter as well as the Windows debuggee laptop are connected to a router which performs DHCP. Then, we perform the steps for setting up 'KDNET' (Windows Kernel netwerk debugging) automatically[3]. It is important that the debugger is already listening for an incoming connection before the debuggee is rebooted.

We were unable to configure a direct ethernet connection between the debugger and debuggee. It appears that when booting with kernel debugging activated, the static IP assignment used for direct ethernet connection (without DHCP) is not well supported. The Windows Kernel debugger explicitly expects DHCP instead of a static IP assignment. Note that there is an option to turn off DHCP for the debugger, but it is unclear which IP will then be used. Therefore, we decided to make use of DHCP so it is clear which IP addresses will be used.

[1] `https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/supported-ethernet-nics-for-network-kernel-debugging-in-windows-11`

[2] `https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/supported-ethernet-nics-for-network-kernel-debugging-in-windows-10`

[3] `https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/setting-up-a-network-debugging-connection-automatically`

4.2 Secure Kernel Debugging Setup

After we have enabled Windows Kernel debugging following the steps from Section 4.1, we want to extend the debugging capabilities to the Secure Kernel. Therefore, in this section we are going to extend the setup to also include Secure Kernel debugging next to Windows Kernel debugging. In Section 4.2.1 we describe an attempt at setting up Secure Kernel debugging using physical hardware. However, at the end of this section, we conclude that it is not possible given the (non-specialized) hardware used. In Section 4.2.2 we describe the process of setting up Secure Kernel debugging using a virtualized setup with QEMU/KVM. The total research time into setting up a working Secure Kernel debugging environment (including time spent on failed research) was about 1.5 weeks.

4.2.1 Physical Setup for Secure Kernel Debugging

This section describes an attempt at setting up Secure Kernel debugging using physical hardware as the debuggee. At the end of this section, we conclude that given the hardware used it is not possible to configure Secure Kernel debugging. Therefore, the setup described in Section 4.2.2 contains information about the QEMU/KVM setup used in this research for Secure Kernel debugging. When referring to WinDBG in this section, if not mentioned explicitly, we refer to the modern UI variant of WinDBG.

Before configuring the debugger to attach to the Secure Kernel, it is important to verify that Virtualization Based Security (VBS) is enabled. We suggest to consult the Microsoft documentation on how to verify that VBS is correctly configured with 'memory integrity' enabled[4].

If you have a working Windows Kernel debugging setup, it is trivial to extend this to the Secure Kernel. On the debuggee the following command can be executed with the `-hk` parameter to both debug the Windows Kernel as well as the Secure Kernel:

```
C:\kdnet>.\kdnet.exe 192.168.88.113 50000 -hk
Enabling network debugging on Intel(R) Ethernet Connection
    (13) I219-LM.

To debug the hypervisor, run the following command on your
    debugger host machine.
windbg -k net:port=50001,key=2dkau18n5sifv.2bu5rgzwhbodh.
    f42r22gp42vi.36riwnl4lk2z5

To debug this machine, run the following command on your
    debugger host machine.
windbg -k net:port=50000,key=1j8spslp8z1cw.2rbwea1rwodzr.
    t6k42u0r38ur.35acc5uvglsm3

Then reboot this machine by running shutdown -r -t 0 from
    this command prompt.
```

Before rebooting the debuggee, it is important to have two WinDBG sessions active at the debugger so that both a connection for the hypervisor debugger as well as the kernel debugger can be created. See Figure 7 for a visualization of this setup.

[4] https://learn.microsoft.com/en-us/windows/security/hardware-security/enable-virtualization-based-protection-of-code-integrity?tabs=security

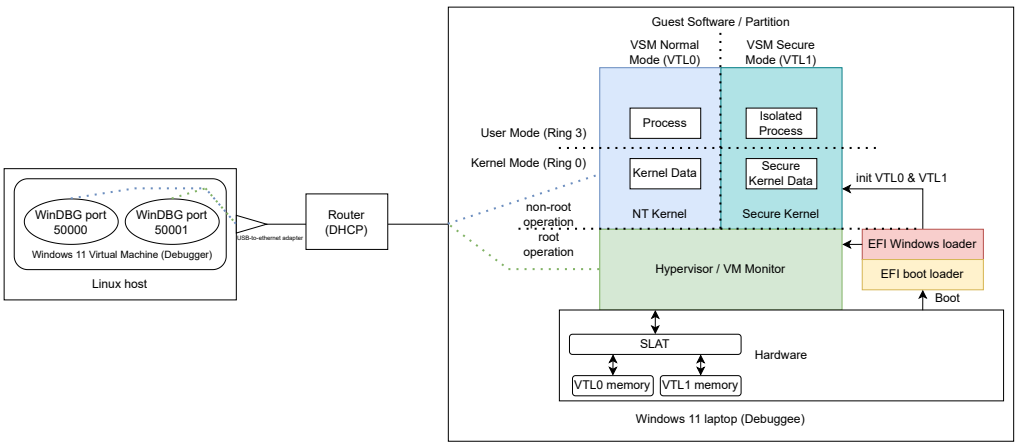


Figure 7: Visualization of the Kernel and Hypervisor Debugging Setup

Using this setup it is possible to both debug the hypervisor binary `hvix64.exe` as well as the NT Kernel. However, the goal for this master thesis is to debug the `securekernel.exe`. According to a blog post written by Quarkslab [56], it is possible to retrieve the `securekernel.exe` base address within VTL1 by setting a breakpoint in the hypervisor at `HvCallVtlReturn` and reading the corresponding VMCS structure (see Section 2.1.2) to extract an address within `securekernel.exe`. This address can then be converted from a virtual address to a physical address (see Section 2.1.3) and by backward scanning the memory for a PE executable header, the `securekernel.exe` base address can be found.

The Virtual-Machine Control Structure (VMCS) structure is located through a `vmptrld` instruction, which belongs to the VMX instruction set (see Section 2.1.1). Since the `hvix64.exe` binary does not include symbols, reverse engineering is required in order to locate the `HvCallVtlReturn` function. The example assembly shown in the Quarkslab blog post can be found in Figure 8. The Windows version that is used on the debuggee laptop during our research is different from the Windows version used by Quarkslab and therefore the offsets have been changed. By configuring IDA to use 'text view' for the assembly and performing a full-text search on the instruction 'vmptrld', converting back to the 'graph view' and pattern matching the surrounding code for each vmptrld instruction, we have identified the `HvCallVtlReturn` function. In Figure 9 the assembly can be found that is used by the `hvix64.exe` binary used by the Windows version installed on the debuggee laptop.

Using the command `?hv` in the WinDBG port 50001 session (the hypervisor debugger), the base address of the hypervisor can be read. After setting and reaching a breakpoint on the `vmptrld [rcx+188h]` instruction, the physical address of the VMCS struct can be read. However, now we encounter a problem: there is no way through both WinDBG sessions to read memory through a physical address. The hypervisor debugger is using the isolation of VTL1 and the kernel debugger is using the isolation of VTL0 and both VTL contexts are using their own virtual address space. And since the debuggers are debugging inside the VTL context, the physical memory is not reachable to both debuggers. Theoretically, the Secure Kernel should be somewhere in the memory space of the hypervisor, but the corresponding virtual address is unknown and difficult to extract. From this, we conclude that (without specialized hardware, see Section 4.3) it is not possible to use physical hardware as the debuggee for Secure Kernel debugging (see also Section 8.2).

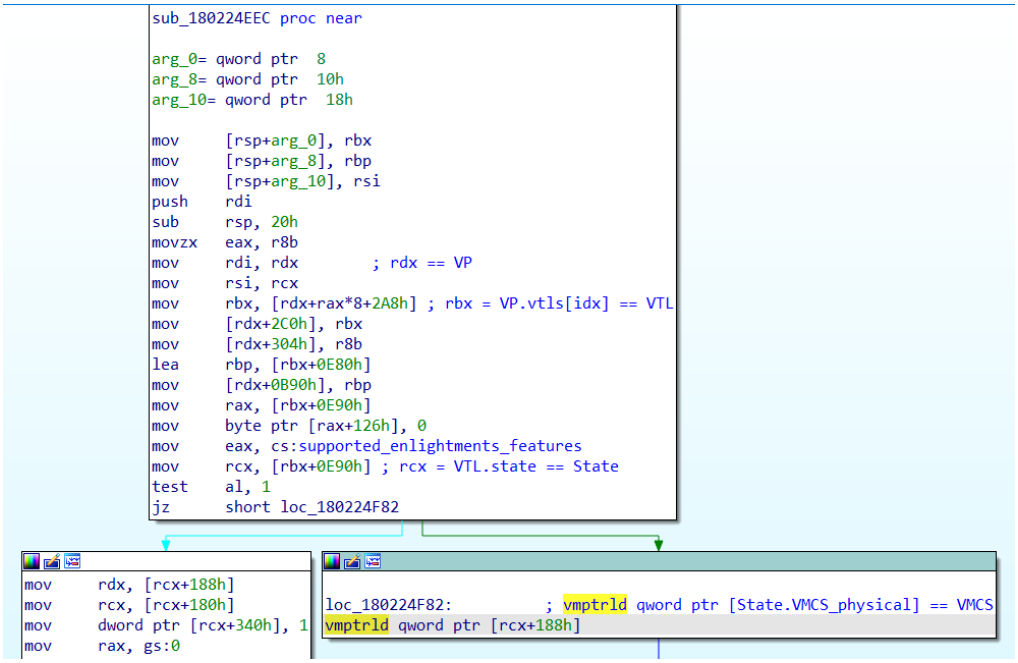


Figure 8: Start of the `HvCallVtlReturn` function in the `hvix64.exe` binary used by Quarkslab, from [56]

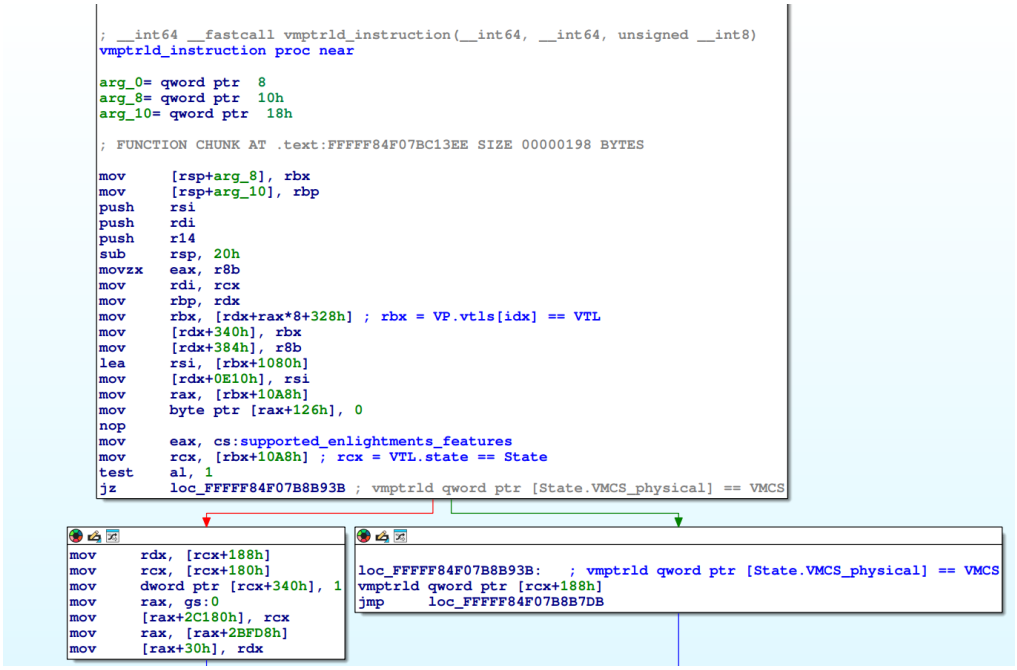


Figure 9: Start of the `HvCallVtlReturn` function in the `hvix64.exe` binary used by the debuggee laptop

4.2.2 QEMU/KVM Virtualized Setup for Secure Kernel Debugging

As concluded in Section 4.2.1, we require a debugging setup where it is possible to read/write physical memory and insert breakpoints on physical addresses. Therefore, we decided to switch to a virtualized setup since a virtualized setup allows us to read the physical memory of a Virtual Machine (VM).

By comparing setups used by other researchers, the common difficulty that is encountered is retrieving the virtual address of the Secure Kernel. Quarkslab solved this by analyzing the VMCS struct (see Section 2.1.2) in the VTL1 context by examining context switches in the `hvix64.exe` binary [56]. This approach is challenging and prone to errors since the `hvix64.exe` binary does not have downloadable symbols and therefore may include different offsets and/or instructions for each Windows installation. Camille Mougey proposes different solutions for retrieving the virtual address of the Secure Kernel [49]:

1. The first option is to inspect the memory and looking for VMCS structures.
2. The second option is to inspect the memory and looking for the Secure Kernel, then by using the pagination tables the associated address can be rebuild.
3. The third option is to break early during the boot sequence, in the function `hvloader` the Secure Kernel will report back its address.
4. The fourth option is to break in Hyper-V and looking for the VTL transition.

Camille Mougey then continues and implements a fifth option: scanning physical memory directly for the `securekernel.exe` binary, patching the `IumInvokeSecureService` function to an infinite loop, continuing execution flow and manually breaking after a few seconds and examining if the infinite loop is reached. If the infinite loop is reached, then through the information from the registers it is possible to retrieve the virtual address of the Secure Kernel. Yarden Shafir also mentions the complexity of the setup used by Quarkslab and shows a Windows-based setup using 'LiveCloudKd' [69] to perform Secure Kernel debugging [67] (see also Section 4.3).

We propose a different solution to solve the problem of retrieving the Secure Kernel virtual (and physical) address. Adrien Chevalier mentions during the boot process of Windows the EFI Windows loader 'winload.efi' will load the Secure Kernel and NT Kernel into memory [6]. Therefore, during the execution of winload.efi the physical and/or virtual address of the Secure Kernel must be known. The entry point address of the Secure Kernel is stored in the `OslpVsmSystemStartup` symbol in winload.efi. Examining the usage of this symbol in winload.efi reveals interesting behavior:

```
.text:0000000001D099BC loc_1D099BC: ; CODE XREF:
   OslpVsmLoadModulesAndCrashDumpKeys+197
.text:0000000001D099BC mov     rdx, [rbp+40h+arg_8]
.text:0000000001D099C0 lea     rcx, aSecurekernelVi ; "
   SecureKernel virtual image base = 0x%p "...
.text:0000000001D099C7 mov     r9, cs:OslpVsmSystemStartup
.text:0000000001D099CE mov     r8d, [rdx+40h]
.text:0000000001D099D2 mov     rdx, [rdx+30h]
.text:0000000001D099D6 call    BlStatusPrint
.text:0000000001D099DB lea     r8,
   OslVsmHvCrashDumpEncryptionKey
```

Decompilation in IDA does not provide code for this functionality and therefore can only be viewed through the disassembly. The EFI Windows loader implements functionality

to print the virtual image base of the Secure Kernel to a connected debugger console. We therefore use this interesting behavior to retrieve the virtual address of the Secure Kernel rather than the VMCS structure as is done by Quarkslab [56]. The corresponding physical address can be retrieved by performing a page table walk of the Secure Kernel virtual address (see Section 2.1.3). However, through the use of hardware-assisted breakpoints by using the `hbreak` command in GDB, it is possible to insert breakpoints into the Secure Kernel without the usage of physical addresses. Physical addresses are still useful for performing read or write operations on the Secure Kernel loaded in memory.

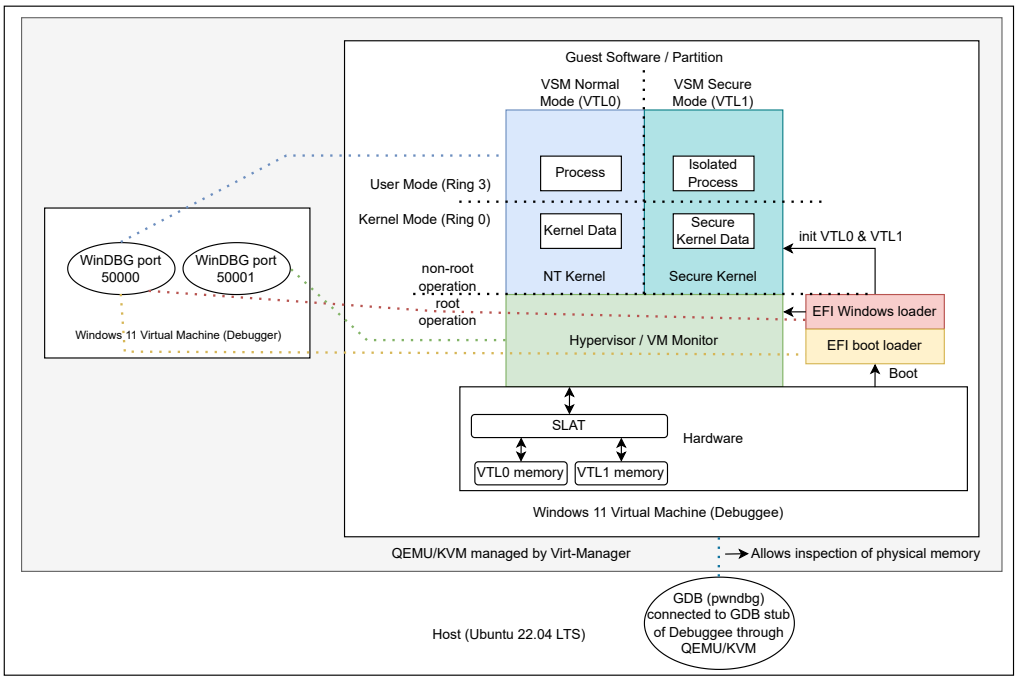


Figure 10: Visualization of the Virtualized EFI Bootloader, EFI Windows loader, Kernel and Hypervisor Debugging Setup

While attaching a debugger only to the EFI Windows loader and having a GDB session through QEMU/KVM is sufficient to perform Secure Kernel debugging, the setup that is described in this section will also include boot manager, hypervisor, and NT Kernel debugging capabilities to allow flexibility for debugging other parts of the Windows operating system. An overview of the setup used can be found in Figure 10. Since this research is performed on a machine with a host operating system of Ubuntu 22.04 LTS, the virtualization will be provided through QEMU/KVM. There is also an alternative setup possible in case the host operating system is not Linux but Windows, see Section 4.3. Virt-Manager [44] is used as a frontend for configuring QEMU/KVM since it allows for easier configuration of VMs. Virt-Manager uses a graphical frontend to manage an XML configuration for each VM.

The setup we use will consists of two VMs:

1. A Windows debugger VM running two WinDBG sessions
2. A Windows debuggee VM (the target VM we will debug)

The two VMs are configured to be within the same virtual network. This allows communication from the debugger VM to the debuggee VM. This can be configured in

Virt-Manager through the virtual networks configuration within the connection details popup.

The debuggee VM needs to be configured in a specific way to allow kernel debugging through WinDBG as well as GDB debugging through QEMU/KVM. First, the debuggee VM needs to have Virtualization Based Security (VBS) enabled since otherwise the Secure Kernel is not loaded. Since we now use a virtualized setup, this requires nested virtualization. The following Virt-Manager CPU configuration allows for nested virtualization within QEMU/KVM:

```
<cpu mode="custom" match="exact" check="partial">
   <model fallback="allow">Skylake-Client-noTSX-IBRS</model>
   <feature policy="disable" name="hypervisor"/>
   <feature policy="require" name="vmx"/>
   <feature policy="disable" name="mpx"/>
</cpu>
```

When the debuggee VM successfully boots into Windows, the 'memory integrity' option can be enabled through Windows Defender to enable VBS (which requires a reboot to activate).

Second, we need to allow for WinDBG debugging (kernel, hypervisor, EFI boot loader, and EFI Windows loader) in the debuggee VM. As mentioned in Section 4.1, in order to allow for debugging the machine needs to support specific network interface cards (NICs). However, now that we have a virtualized setup, a NIC cannot be chosen anymore. By default, Virt-Manager allows configuring an 'e1000e' or 'virtio' NIC for the VM. In our case, we will make use of the 'e1000e' NIC, but this is not sufficient without extra configuration. The following features need to be configured for the VM in Virt-Manager to allow debugging capabilities with WinDBG, where special attention is required for the `<vendor_id state="on" value="KVMKVMKVM"/>` value [75]:

```
<features>
    <acpi/>
    <apic/>
    <hyperv mode="custom">
        <relaxed state="on"/>
        <vapic state="on"/>
        <spinlocks state="on" retries="8191"/>
        <vendor_id state="on" value="KVMKVMKVM"/>
    </hyperv>
    <kvm>
        <hidden state="on"/>
    </kvm>
    <vmport state="off"/>
    <smm state="on"/>
    <ioapic driver="kvm"/>
</features>
```

Using this configuration, it is possible to configure debugging through `kdnet.exe` with the following command where `-bhkw` enables kernel, hypervisor, EFI boot loader, and EFI Windows loader debugging ('192.168.122.224' is the IP of the debugger VM):

```
C:\kdnet>.\kdnet.exe 192.168.122.224 50000 -bhkw
Enabling network debugging on Intel(R) Ethernet Connection
   (13) I219-LM.
```

```
To debug the hypervisor, run the following command on your
    debugger host machine.
windbg -k net:port=50001,key=2bh3bggmptiqs.2vs6a8tvtgh9r.1
    gabysttkv5pv.1eexgc247jajw

To debug this machine, run the following command on your
    debugger host machine.
windbg -k net:port=50000,key=37420v9g7zd6g.hiyg5wx3cfuz.3
    jir69zr7wx2l.3u4tnuf6bgpau

Then reboot this machine by running shutdown -r -t 0 from
    this command prompt.
```

Using the instructions given by the output of the `kdnet.exe` command, it is possible to configure the two WinDBG sessions in the debugger VM. It is important to enable 'Break on connection' in WinDBG to allow debugging capabilities during boot. Note that in case no connection is received in the WinDBG sessions after rebooting the debuggee VM, there may be an error in the network configuration that blocks the two VMs from communicating with each other.

Third, special configuration is required to enable the GDB stub provided by QEMU/KVM to enable GDB debugging capabilities for the debuggee VM. For QEMU, the command line flag to enable the GDB stub is `-s` which allows GDB to connect to `127.0.0.1:1234` on the host machine. However, configuration is done through Virt-Manager and therefore the XML configuration needs to be modified in the following way. The first line of the XML configuration needs to be updated to:

```
<domain xmlns:qemu="http://libvirt.org/schemas/domain/qemu
    /1.0" type="kvm">
```

This is necessary since otherwise the schema does not accept the XML configuring the command line flag for enabling the GDB stub. To enable the GDB stub, inside the `<domain>...</domain>` the following configuration needs to be added:

```
<qemu:commandline>
    <qemu:arg value="-s"/>
</qemu:commandline>
```

Note that this configuration may be automatically moved to the bottom of the XML configuration by Virt-Manager. This configuration directly adds the required `-s` to the QEMU process which enables the GDB stub on `127.0.0.1:1234`. When the debuggee VM is now booted, it is possible to connect at any time with the QEMU/KVM GDB stub. In this research, pwndbg [54] is used as an extension on GDB to allow rich output and ease of debugging. In order to inspect physical memory in QEMU/KVM through the GDB stub, it is important to run the following command to enable physical memory mode: `maintenance packet Qqemu.PhyMemMode:1`. Disabling this mode is necessary as soon as a virtual address needs to be resolved, which can be done using `maintenance packet Qqemu.PhyMemMode:0` [55].

Fourth, it is preferred to configure the debuggee VM to run on a single vCPU. This improves stability during debugging through QEMU/KVM since context switches between vCPUs are not performed anymore. This will solve problems where memory appears to be unreadable while a hardware-assisted breakpoint is still reached.

4.2.2.1 QEMU/KVM Secure Kernel debugging workflow

Below the workflow for the QEMU/KVM debugging setup is described, assuming that the configuration steps in Section 4.2.2 are successfully performed.

1. Boot the debugger VM and enable the two WinDBG sessions according to the output of `kdnet.exe` on the debuggee VM. Note that 'Break on connection' needs to be enabled.
2. Boot the debuggee VM.
3. Connect from the host to the QEMU/KVM GDB stub using `target remote 127.0.0.1:1234` in GDB, then continue execution with `c`.
4. The WinDBG session on port 50000 will receive a connection for the EFI boot loader (`bootmgfw`), which can be continued using `g`.
5. The boot loader will close and a new connection on the WinDBG session on port 50000 will be received for the EFI Windows loader (`winload`). Add a breakpoint after the instruction that calls `BlStatusPrint` to print the Secure Kernel virtual image base. The address for this breakpoint can be computed using the 'Primary image base' as printed by WinDBG when the connection is received from the EFI Windows loader plus the offset to the instruction after the call to `BlStatusPrint`. In our case this is the following WinDBG command: `bp 0000000001D099DB`. Note that this address is persistent across reboots.
6. Continue execution with the command `g`. In the output console, the Secure Kernel virtual image base will be printed and the breakpoint will be hit. In this example, the Secure Kernel virtual image base that is printed is 0xFFFFF8055F27E000.
7. Compute the virtual address of the location where a breakpoint in the Secure Kernel needs to be set. In this example, we will insert a breakpoint at the start of the function `IumInvokeSecureService` which is in the `securekernel.exe` binary of our debuggee VM at location 0xFFFFF8046A598600.
8. Press ctrl+c in the GDB terminal on the host to start debugging through QEMU/KVM. We want to insert a hardware-assisted breakpoint at the location of `IumInvokeSecureService` which can be done with the following command: `hbreak *(0xFFFFF8055F288600)`.
9. Continue execution in GDB with the command `c`.
10. Continue execution in the WinDBG port 50000 session with the command `g`.
11. A new connection will be received in the WinDBG port 50000 session which is from the EFI Windows loader again (`winload`), which can be skipped with the command `g`.
12. A connection is now received in the WinDBG port 50001 session from the hypervisor, which can be skipped with the command `g`.
13. Enter the command `g` again to skip over an extra interrupt breakpoint to continue the execution flow of the hypervisor.
14. At this point, the breakpoint for `IumInvokeSecureService` is hit and execution is paused in the GDB session. In order to continue execution, the breakpoint can be temporarily disabled using `disable 1` where '1' is the breakpoint number. Then execution can be continued with the command `c`.

15. A connection is now received in the WinDBG port 50000 session for the NT Kernel, which can be skipped twice with the command `g`.

16. The debuggee VM now fully boots into Windows. At this point, it is possible to debug the hypervisor, NT Kernel or enable the hardware-assisted breakpoint in GDB again using ctrl+c and the command `enable 1`. An example of successful output can be found in Figure 11.

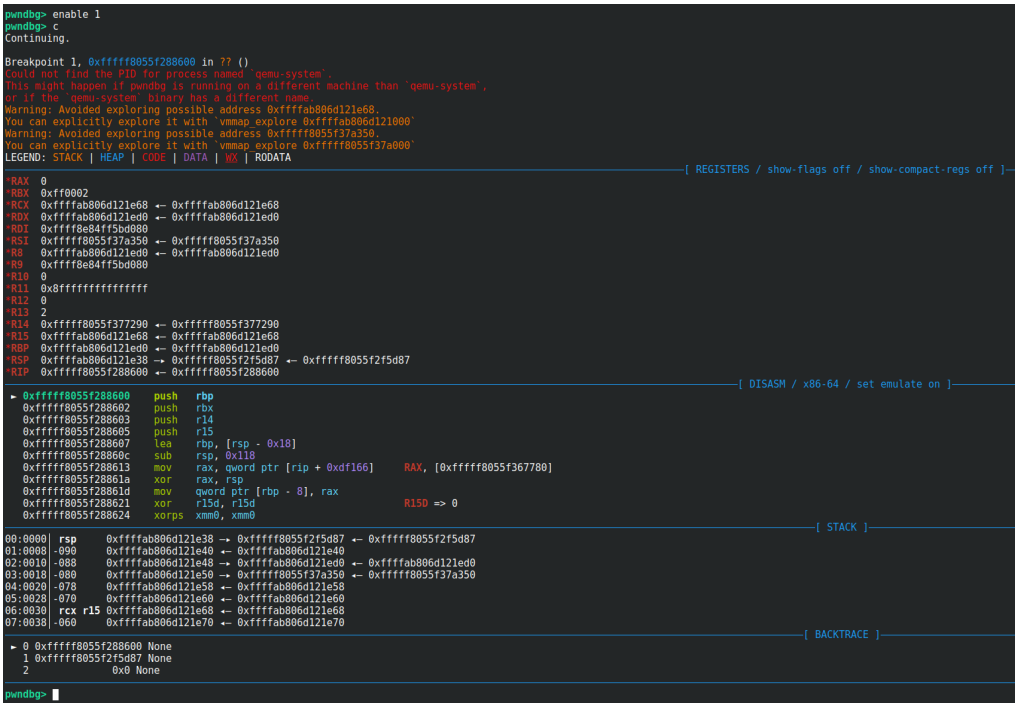


Figure 11: Example output of successfully reaching a breakpoint in the `IumInvokeSecureService` function of the Secure Kernel

Using this workflow, it is possible to debug the following components:

- Hypervisor (WinDBG)
- NT Kernel (WinDBG)
- EFI boot loader (WinDBG)
- EFI Windows loader (WinDBG)
- Secure Kernel (GDB)

The GDB debugger can always be used for debugging since it allows debugging all code running inside the VM, but preference should be given to WinDBG when possible since it better integrates into the Windows ecosystem by for example being able to automatically use the Microsoft Public Symbol Server.

See Appendix B for the complete Virt-Manager XML configuration of the debuggee VM. Note, however, that the relevant parts of the XML configuration are described in this section and Appendix B should only be used for troubleshooting purposes.

4.2.3 Stability & Performance of the QEMU/KVM Virtualized Setup for Secure Kernel Debugging

The QEMU/KVM setup described in Section 4.2.2 is usable for debugging the Secure Kernel, as well as the NT Kernel, hypervisor, EFI boot loader, and EFI Windows loader. This makes it a diverse setup to use for all sorts of research into the Windows operating system. However, the setup is not perfect. Like with alternative setups (which we will discuss in Section 4.3), this setup also suffers from its own stability and performance problems. We noticed these problems during the setup of QEMU/KVM (Section 4.2.2) as well as during the Exploitability Assessment (Section 5). While stability improved by only assigning one vCPU core to the debuggee VM, some problems remain. There are two specific problems related to stability:

1. Stepping through the instructions does not work.
2. Some hardware breakpoints are incompatible with each other.

The first problem is about reaching a breakpoint and wanting to step through the instructions one by one. This would normally be done with commands like `ni` and `si`. These commands do not work in our setup, however. When executing such single-step commands, code execution will change to a completely different context. Then, even after executing the 'continue' command, the debuggee VM is frozen and needs a hard reboot. This problem is, however, trivial to bypass, since we can just insert a lot of hardware breakpoints to stop execution at several places where we want to inspect the context. Since reaching a breakpoint is working as expected, we can use that as a replacement for single-stepping.

However, we encountered a second unexpected issue: some hardware breakpoints are 'incompatible' with each other. With 'incompatible', we mean that first defining a breakpoint on some location and then defining a second breakpoint on a different location would break the first breakpoint and block the continuation of code execution. This is a very mysterious bug for which we were unable to pinpoint the problem. We also discussed this with our supervisor and got the feedback that debuggers are more like 'hit or miss': it sometimes works and sometimes it does not. Probably this problem is due to a bug somewhere in the GDB stub of QEMU/KVM. Debugging a debugger is not something we want to invest time in, so we just tried to bypass the problem. GDB does report an error for the broken breakpoint, so we delete this breakpoint in order to continue execution. In case it is not preferable to delete a breakpoint, then it is possible to:

1. Delete the second breakpoint so the first breakpoint will start working again,
2. continue execution until the first breakpoint,
3. delete the first breakpoint,
4. insert the second breakpoint, and
5. continue execution until the second breakpoint.

This method is cumbersome, but it will allow the usage of any breakpoint location.

Furthermore, there are also some performance problems with the QEMU/KVM setup described in Section 4.2.2. Since we have to make use of Virtualization Based Security (VBS) in order to load the Secure Kernel, it is required to allow nested virtualization through Virtual Mode Extensions (VMX). QEMU/KVM only allows this when the CPU is configured to 'Skylake-Client-noTSX-IBRS' during our testing. This, however, results in poor performance of the whole debuggee VM. The Windows UI is slow to respond,

and it takes about ten seconds to copy a file of only a few kilobytes. Furthermore, a reboot cycle takes about five minutes during our testing. While this performance is still in a barely usable state, it makes the Exploitability Assessment of Section 5, as well as the development of the setup to interact with the Secure Kernel as described in Section 4.4, take more time than necessary.

It is unfortunate that our QEMU/KVM setup described in Section 4.2.2 also has stability and performance issues. It may be the case that a more up-to-date version of QEMU/KVM would resolve one or more problems. The QEMU/KVM version used in this research is '6.2.0 (Debian 1:6.2+dfsg-2ubuntu6.25)', which is quite old due to Ubuntu 22.04 LTS, since the latest QEMU/KVM version is 9.2 at the time of writing.

4.2.4 GDB plugin for resolving virtual addresses to physical addresses

As mentioned in Section 4.2.2, it may be useful to be able to resolve virtual addresses to physical addresses to perform read/write operations on the memory without depending on any process context. For example, when entering in GDB ctrl+c at any time during post-boot execution of the debuggee VM, the execution context is different from the Secure Kernel execution context and therefore the virtual addresses of the Secure Kernel cannot be resolved anymore. This can be resolved by either hitting a breakpoint in the Secure Kernel so the execution context is correctly loaded again, or using a physical address. We have developed a GDB plugin[5] to resolve virtual addresses to physical addresses by implementing the page table walk as described in Section 2.1.3. When the script is loaded into GDB with the `source` command, the command `ptwalk` can be used to resolve a virtual address to a physical address. Note that during resolving the virtual address, the execution context must be correct for the virtual address which means that when examining the virtual address, the output should be successfully returned. An example output for the `ptwalk` command can be found below:

```
pwndbg> ptwalk 0xfffff8055f288600
sending: Qqemu.PhyMemMode:1
received: "OK"
CR3 = 0x4400000
Virtual Address = 0xfffff8055f288600
PML4 index = 0x1f0
PDPT index = 0x15
PD index   = 0xf9
PT index   = 0x88
Page offset = 0x600
PML4 Entry @ 0x4400f80 = 0x4403063
PDPT Entry @ 0x44030a8 = 0x4402063
PD Entry @ 0x44027c8 = 0x4401063
PT Entry @ 0x4401440 = 0x1669021
Physical Address = 0x1669600
sending: Qqemu.PhyMemMode:0
received: "OK"
pwndbg>
```

Note that the resulting physical address `0x1669600` can only be read when enabling physical memory mode in the GDB stub of QEMU/KVM. The GDB plugin automatically enables and disables physical memory mode in order to not interfere with the

[5] `https://github.com/JJ-8/ptwalk`

debugging session. Physical memory mode can be enabled using `maintenance packet Qqemu.PhyMemMode:1` and disabled with `maintenance packet Qqemu.PhyMemMode:0` [55].

4.2.5 VMWare Virtualized Setup for Secure Kernel Debugging

As mentioned in Section 4.2.3, the stability and performance of the Secure Kernel debugging setup using QEMU/KVM as a virtualization backend is not perfect. During our research in Section 5.1, we made use of the Secure Kernel debugging setup with QEMU/KVM to evaluate the Exploitability Assessment of CVE-2024-43528. Since CVE-2024-43528 was difficult to understand using manual code analysis, we heavily relied on dynamic code analysis. During the dynamic code analysis, we had to insert a lot of hardware breakpoints to debug the Secure Kernel as well as frequently rebooting the Virtual Machine (VM) to reload the custom kernel module we developed in Section 4.4.2. This is a slow process, since a reboot cycle takes about five minutes when using QEMU/KVM as the virtualization backend. Furthermore, we were unable to speed up this process since snapshots are not possible with our QEMU/KVM Secure Kernel debugging setup (see Section 4.2.3). Therefore, we decided to invest time in switching to VMWare as the virtualization backend. We have chosen VMWare since it is also used by Quarkslab to successfully debug the Secure Kernel [56], so we know it is possible to have a working Secure Kernel debugging setup using VMWare.

The setup used by Quarkslab makes use of nested virtualization to debug the Secure Kernel. This is because they are specifically interested in debugging the Virtual TPM used by Hyper-V when running a Windows VM. However, we are interested in debugging the Secure Kernel and therefore we do not require nested virtualization. Furthermore, we have already identified a better method compared to Quarkslab's method for identifying the base address of the Secure Kernel (see Section 4.2.2). We will make use of this improved method for our VMWare Secure Kernel debugging setup.

4.2.5.1 VMWare debugger VM and debuggee VM setup

We decided to start from scratch with a newly installed Windows VM since we thought it would take more time to migrate our current debuggee VM from QEMU/KVM to VMWare. The installation of VMWare itself was difficult, because the Broadcom website appeared to be bugged, which made it impossible to download VMWare. Furthermore, at the time of writing, the software update domain `https://softwareupdate.vmware.com/` appeared to be offline, which made it impossible to download VMWare. After a lot of online searching, we found an installer stored on the Wayback Machine for VMWare 17.6.3[6] which we used for installing VMWare. After installing VMWare, we had to select a Windows version to install in the newly created VM. At the time of writing, Windows 24H2 was the most recent Windows version and also the Windows version available for download from Microsoft. However, before starting to install 24H2, we found out that there are some issues with up-to-date Windows 11 24H2 installs combined with Virtualization Based Security (VBS) within VMWare [38]. Therefore, we decided to install Windows 11 23H2 in VMWare to prevent any compatibility issues with VBS.

Furthermore, we installed the VMWare guest tools to have automatic screen resizing and shared clipboard available. Since, at the time of writing, the software update domain was unavailable, the installation could not be done through VMWare itself. However,

[6] `https://web.archive.org/web/20250304135300/https://softwareupdate.vmware.com/cds/vmw-desktop/ws/17.6.3/24583834/linux/core/VMware-Workstation-17.6.3-24583834.x86_64.bundle.tar`

the exe-file can be downloaded[7] and executed within the VM to install the VMWare guest tools successfully. A reboot is required after successful installation.

After the installation of Windows 23H2 inside VMWare, we made a short attempt to connect the VMWare VM to the QEMU/KVM debugger VM by bridging the VMWare network with the QEMU/KVM network. This would reduce the amount of installation time since we already have a fully working debugger VM inside QEMU/KVM. However, correctly bridging the two networks appeared to be non-trivial since the debugger VM and debuggee VM should be in the exact same subnet to have direct IP communication without Network Address Translation. Therefore, we decided to clone the debuggee VM to use it as the debugger VM within VMWare. Since, with this setup, both VMs are within VMWare, they can communicate directly with each other through the VMWare network interface 'vmnet8'.

As with the QEMU/KVM setup described in Section 4.2.2, we want to have the following three debugging sessions:

1. WinDBG session on port 50000 for bootloader, EFI loader, and NT Kernel debugging
2. WinDBG session on port 50001 for hypervisor debugging
3. GDB session for Secure Kernel debugging

The two WinDBG sessions will run in the debugger VM inside VMWare, and the GDB session will connect to the GDB stub of VMWare.

In order to configure bootloader, EFI loader, NT Kernel, and hypervisor debugging, we have to set up `kdnet.exe` on the debuggee VM. `kdnet.exe` can only be configured when secure boot is disabled, and since VBS requires secure boot to be enabled in VMWare, this should be set up before enabling VBS. First, we enabled bootloader, EFI loader, NT Kernel, and hypervisor debugging on the debuggee VM similar to our debugging setup with QEMU/KVM. However, due to a bootloader bug within VMWare, WinDBG is unable to start a debugging session:

```
Microsoft (R) Windows Debugger Version 10.0.27793.1000 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.

Using NET for debugging
Opened WinSock 2.0
Waiting to reconnect...
Connected to target 192.168.225.128 on port 50000 on local
   IP 192.168.225.129.
You can get the target MAC address by running .kdtargetmac
   command.
BD: Boot Debugger Initialized
Connected to Windows Boot Debugger 22621 x64 target at (Mon
   Apr 14 21:09:05.933 2025 (UTC + 2:00)), ptr64 TRUE
Kernel Debugger connection established.  (Initial Breakpoint
    requested)

************* Path validation summary **************
Response                         Time (ms)     Location
Deferred                                       srv*
```

[7] https://packages.vmware.com/tools/releases/latest/windows/x64/VMware-tools-12.5.1-24649672-x64.exe

```
Symbol search path is: srv*
Executable search path is:
ReadVirtual() failed in GetXStateConfiguration() first read
    attempt (error == 0.)
CS descriptor lookup failed
Windows Boot Debugger Kernel Version 22621 UP Free x64
Primary image base = 0x00000000`10000000 Loaded module list
    = 0x00000000`1017fa20
System Uptime: not available
Unable to get program counter
0018:8375 0000            add     byte ptr [bx+si],al
```

Through a WinDBG GitHub issue[8], we found out that this was caused by enabling bootloader debugging. Since bootloader debugging is not necessary for debugging the Secure Kernel, we have disabled this. Therefore, we have used the following command to configure `kdnet.exe`:

```
C:\kdnet>.\kdnet.exe 192.168.225.129 50000 -hkw
Enabling network debugging on Intel(R) 82574L Gigabit
    Network Connection.

To debug the hypervisor, run the following command on your
    debugger host machine.
windbg -k net:port=50001,key=xkii0m5mi03o.4db352olw3ar.18
    i07r0o54kmy.3f5kd18vdpu7y

To debug this machine, run the following command on your
    debugger host machine.
windbg -k net:port=50000,key=308fcwww5xs7m.zutachl9fi94.1
    tyd2is749c5g.1gy97dvbsyxep

Then reboot this machine by running shutdown -r -t 0 from
    this command prompt.
```

Note that 192.168.225.129 is the IP of the debugger VM within the VMWare internal network.

Furthermore, we have to enable VBS support in VMWare. VMWare has a dedicated option for this, which can be found in the interface through 'Edit virtual machine settings' → tab 'options' → 'Enable Virtualization Based Security (VBS) support'. This will also enable secure boot automatically. Lastly, in the debuggee VM itself, we can execute the following command to enforce that the Secure Kernel is loaded during startup without enforcing memory integrity:

```
reg add "HKLM\SYSTEM\CurrentControlSet\Control\DeviceGuard"
    /v "EnableVirtualizationBasedSecurity" /t REG_DWORD /d 1
    /f
```

Below, the command and output can be found to check if VBS is correctly configured after rebooting the debuggee VM.

```
PS C:\Windows\system32> Get-CimInstance -ClassName Win32_
    DeviceGuard -Namespace root\Microsoft\Windows\DeviceGuard
```

[8] https://github.com/microsoft/WinDbg-Feedback/issues/75

```
AvailableSecurityProperties : {1, 3, 4, 5...}
   CodeIntegrityPolicyEnforcementStatus : 2
InstanceIdentifier : 4ff40742-2649-41b8-bdd1-e80fad1cce80
   RequiredSecurityProperties : {0}
SecurityFeaturesEnabled : {0} SecurityServicesConfigured :
   {2}
SecurityServicesRunning : {2}
   UsermodeCodeIntegrityPolicyEnforcementStatus : 1
Version : 1.0 VirtualizationBasedSecurityStatus : 2
VirtualMachineIsolation : False
   VirtualMachineIsolationProperties : {0}
PSComputerName :
```

In order to configure the GDB stub of VMWare, we had to make the following additions to the VMWare '.vmx' configuration file of the debuggee VM.

```
debugStub.listen.guest64 = "TRUE"
debugStub.hideBreakpoints= "TRUE"
debugStub.port.guest64 = "1234"
```

Note that there is also the option `monitor.debugOnStartGuest32 = "TRUE"`, but this option should *not* be configured since it prevents the WinDBG session in the debugger VM from correctly connecting to the debuggee VM. In order to connect with GDB, the following two commands can be used:

```
set remotetimeout 100
target remote :1234
```

The GDB extension pwndbg [54] is functional with VMWare and recommended to be used during debugging. The `set remotetimeout` command is required for VMWare since sometimes connecting triggers a timeout within GDB, which causes the connection to fail. When a breakpoint is hit in GDB and execution is halted, VMWare will show overlay UI on top of the VM screen output, which can be seen in Figure 12.

Clicking somewhere on this overlay UI will continue execution of the debuggee VM. However, it is strongly recommended to use the GDB command `continue` to continue execution. Furthermore, we encountered a bug with VMWare related to the overlay UI: when continuing execution, the overlay UI is not removed. The VM execution continues, but the display output is not restored. In order to bypass this bug, the VMWare window should be closed, but the debuggee VM should still run in the background (this option is given as a prompt when closing VMWare with an active VM open). When reopening VMWare, the overlay UI is removed, and input can be given again to the VM. It is recommended to have the debuggee VM and debugger VM in separate windows so only one of the two windows has to be closed and reopened.

4.2.5.2 VMWare Secure Kernel debugging workflow

Below the workflow for the VMWare debugging setup is described, assuming that the configuration steps in Section 4.2.5.1 are successfully performed.

1. Boot the debugger VM and enable the two WinDBG sessions according to the output of `kdnet.exe` on the debuggee VM. Note that 'Break on connection' needs to be enabled.
2. Boot the debuggee VM.

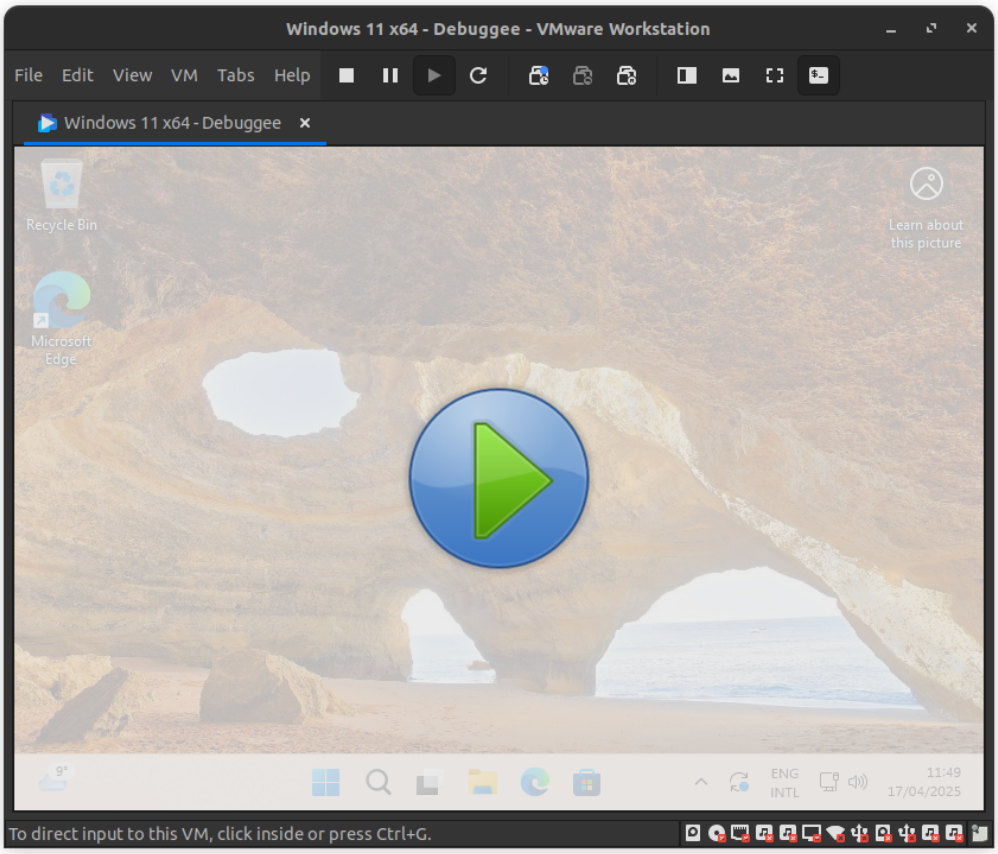


Figure 12: UI shown when GDB breakpoint is triggered

3. Ignore the warning from VMWare about enabling promiscuous mode on adapter 'Ethernet0'.
4. The WinDBG session on port 50000 in the debugger VM will receive a connection for the EFI Windows loader (`winload`), which can be continued using `g`.
5. The Secure Kernel virtual image base address will now be printed and can be used to rebase the `securekernel.exe` binary.
6. Connect with pwndbg to the GDB stub of VMWare using `target remote :1234`.
7. Insert a hardware breakpoint in the Secure Kernel with the command `hbreak *0x<address>` in GDB.
8. Continue execution in GDB with `c`.
9. Close the VMWare window and choose 'Run in Background' since it is now unresponsive.
10. Open the debugger VM and debuggee VM windows in VMWare again.
11. Continue execution in the WinDBG port 50000 session with the command `g` to continue the second break in the EFI Windows loader.
12. A connection is now received in the WinDBG port 50001 session from the hypervisor, which can be skipped with the command `g`.

13. Enter the command `g` again to skip over an extra interrupt breakpoint to continue the execution flow of the hypervisor.
14. A connection is now received in the WinDBG port 50000 session for the NT Kernel, which can be skipped twice with the command `g`.

Depending on where the hardware breakpoint was set in GDB, the debuggee VM will halt execution, show the UI overlay of VMWare, and the GDB session will become interactive to start debugging. Note that each time a hardware breakpoint is hit in the debuggee VM and execution is continued in GDB, the VMWare window must be restarted in order to remove the overlay UI and make the screen responsive again.

Secure Kernel memory is only available when a hardware breakpoint is hit in the Secure Kernel and not when execution is halted outside of the Secure Kernel context. This is likely to be related to CPU context switches. A good strategy for having Secure Kernel memory easily available for inspection is to insert a hardware breakpoint at the start of `IumInvokeSecureService`, since this function will be executed about every second.

VM snapshots through VMWare are working normally during debugging with WinDBG and GDB. When a VM state is reverted to a previous snapshot, the GDB connection will stop. However, reconnecting with the VMWare GDB stub allows for further debugging. The WinDBG session is not interrupted when taking a snapshot or reverting to a previous snapshot.

4.3 Alternative Secure Kernel debugging setups

In Section 4.2 we explained the Secure Kernel debugging setup used for this research. However, there are alternative methods that can be considered for debugging the Secure Kernel. There are two different alternative setups possible:

1. With the SourcePoint JTAG debugger and an AAEON UP Xtreme i11 (Tiger Lake) / AAEON UP Xtreme i12 (Alder Lake) physical target, it is possible to debug the Secure Kernel directly from the CPU [62] [63]. This setup does make it possible to debug the Secure Kernel with physical hardware. However, this setup has not been used since it requires specialized hardware that we do not currently have access to.
2. In this research we first make use of QEMU/KVM as the virtualization software as described in Section 4.2.2, but this is not strictly required. It is also possible to use different virtualization software. For example, Quarkslab makes use of VMWare for debugging the Secure Kernel which is a platform-independent virtualization software [56] which we also decided to use as described in Section 4.2.5. The setup from Quarkslab is different from our setup since it uses nested virtualization which was required for their research. Another possibility is to use Hyper-V as the virtualization backend, which is made possible through LiveCloudKd. We further elaborate on LiveCloudKd in Section 4.3.1 since we have tested this setup as a potential suitable debugging setup solution.

4.3.1 LiveCloudKd

In case the host operating system is Windows, one can consider using LiveCloudKd for debugging. LiveCloudKd makes use of Hyper-V as the virtualization backend and provides a solution for Secure Kernel debugging. For experimentation purposes, the setup with LiveCloudKd has been reproduced on the Windows laptop that was initially planned to be used as the physical laptop. LiveCloudKd can be used in two ways:

1. Nested virtualization
2. Specific Windows host operating system version

Since we expected the nested virtualization method to be difficult to set up and we questioned the speed performance of the setup, we decided to use the specific host operating system version. LiveCloudKd requires Windows Server 2019 (with August 2020 updates: '`en_windows_server_2019_updated_aug_2020_x64_dvd_f4bab427.iso`') or Windows 10 20H1 (19041) [26] as the host OS [32]. However, it has also been reported that LiveCloudKd works on a 64-bit Windows 11 23H2 host operating system [67]. Since the installation of Windows Server 2019 failed on our laptop, we used the Windows 10 20H1 (19041) version.

LiveCloudKd requires having an arbitrary Windows guest OS installed, virtualized by Hyper-V. We were only successful in installing LiveCloudKd in the classic WinDBG environment, not the modern UI variant. When launched after installation, LiveCloudKd will launch classic WinDBG and open a terminal printing the Secure Kernel base address as well as the NT Kernel base address. Using the Secure Kernel address it is possible to insert a breakpoint at, for example, `IumInvokeSecureService`. WinDBG can then be used as the debugger for debugging the Secure Kernel.

The setup used by LiveCloudKd is similar to the setup used in Section 4.2.2. The difference is the virtualization software used, but both setups directly inspect the VM memory and insert breakpoints in the execution flow of the VM.

The reason that we decided to develop the virtualized setup as described in Section 4.2.2 instead of using LiveCloudKd is due to the lack of stability of LiveCloudKd. We were not able to consistently successfully start a Secure Kernel debugging session since sometimes it will fail without any clear reason.

It is interesting to note that LiveCloudKd appears to use an undocumented way of recovering the Secure Kernel virtual base address. See Section 7.2 for more information and future research ideas about this methodology.

4.4 Sending custom Hypercalls requests to the Secure Kernel

Using the QEMU/KVM setup from Section 4.2.2, we can debug the Secure Kernel using GDB. This is necessary for the dynamic analyses done in Section 5. There is only one step remaining for the setup in order to do dynamic analyses: we need to be able to interact with the Secure Kernel itself. This means that we need to be able to send Hypercalls (see Section 2.3.2) to the Secure Kernel. The Hypercalls we send should contain a Secure System Call Number (SSCN) and parameters customized by us, so we can try to trigger the vulnerability with a specific payload. In this section, we discuss how we approached this problem, what we have tried, and what we developed in order to make this possible.

First, in Section 4.4.1 we briefly discuss what KernelForge [51] is, what problem it tries to solve, and what we have tried to get this working. In the end, we didn't make use of KernelForge since it did not work. Therefore, in Section 4.4.2 we discuss our approach to developing a kernel driver to interact with the Secure Kernel. In order to develop a kernel driver, we had to make some small modifications to our setup to allow debugging of the kernel driver which we discuss in Section 4.4.2.1. After this setup has been done, we continued with developing a new approach to interacting with the Secure Kernel through the NT Kernel, which we explain in Section 4.4.2.2.

4.4.1 KernelForge

Since more vulnerability research has been done on the Secure Kernel, we assumed there was already an existing project that allows interaction with the Secure Kernel or a documented methodology. This assumption was correct, since we have found the KernelForge project [51] which is made as a library to easily interact with the NT Kernel and also the Secure Kernel even when security features of Virtualization Based Security (VBS) (see Section 2.2) are enabled. In this section, we explain the concept of KernelForge and what we have tried to make it work. However, in the end, we decided not to use KernelForge due to it being broken and unable to fix the problems. Instead, we decided to write our own custom kernel driver, which is explained in Section 4.4.2.

Due to VBS, interacting with the NT Kernel and Secure Kernel has become more challenging for rootkits and kernel exploits. Since the Secure Kernel can control the NT Kernel, it allows for very strong protections against unauthorized code execution. One of the security features that VBS provides is Hypervisor-Enforced Code Integrity (HVCI). HVCI is used for two security features [15]:

1. HVCI protects modifications of the Control Flow Guard (CFG) bitmap for kernel drivers.
2. HVCI protects modifications of the kernel mode code that is responsible for verifying certificates of loaded kernel drivers.

The first security feature is not relevant for loading kernel drivers, since it is purely made for exploitation protection for the NT Kernel. Yet this is still relevant when you want to call arbitrary kernel functions through an exploitation primitive, since CFG will prevent this. The second security feature is relevant for loading custom (unsigned) kernel drivers and for preventing exploitation when the NT Kernel is compromised.

KernelForge uses a common technique to bypass HVCI, which consists of using threads, an arbitrary kernel read/write memory primitive, and an ROP-chain to execute any kernel function. This technique is exposed to an (exploit) developer as a Windows C-library that can be used for easy interaction with the NT Kernel when a kernel memory read/write primitive is given. Direct interaction with the NT Kernel by calling exported and non-exported functions implies that interaction with the Secure Kernel is possible since the NT Kernel has the possibility for communication with the Secure Kernel through Hypercalls. In Figure 13, an overview is given of how KernelForge is used to interact with the Secure Kernel. Our goal is to interact with the Secure Kernel on a Virtual Machine (VM) we control. Therefore, we do not need an exploit for the kernel read/write primitive and we can just load a kernel driver ourselves. KernelForge supports this through the `WinIo.sys` kernel driver, which provides full physical memory access and should work even when HVCI is enabled. This allows us to use the KernelForge library to automatically call exported and non-exported functions in the NT Kernel from a user-space client using the primitive given by the `WinIo.sys` driver. Therefore, the `WinIo.sys` kernel driver is only loaded once and then from user-mode ring 3 the communication to the Secure Kernel is done by using the KernelForge library to call NT Kernel functions.

KernelForge claims that the `WinIo.sys` kernel driver works with HVCI enabled. This is, however, not true anymore. When enabling the 'memory integrity' functionality in the Windows Defender settings, it will automatically enable the 'Microsoft Vulnerable Driver Blocklist' which cannot be disabled. This blocklist will prevent the `WinIo.sys` kernel driver from loading and will delete the kernel driver file instead since Microsoft is aware of it granting exploitation primitives. See Figure 14 for the corresponding popup.

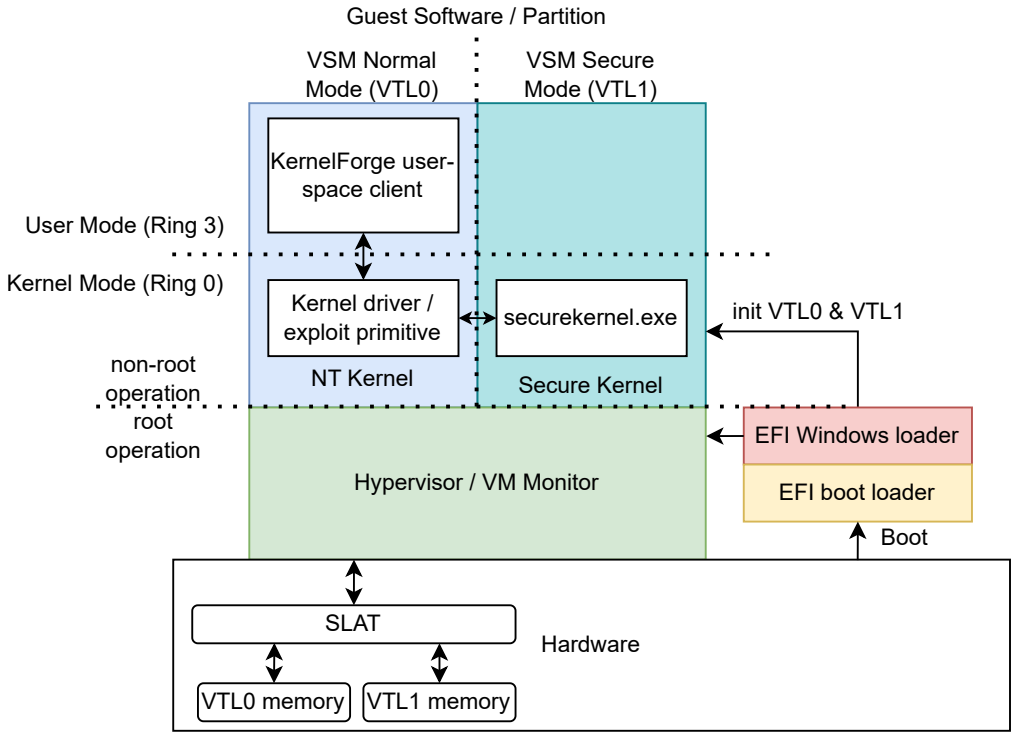


Figure 13: Overview of the setup of KernelForge

However, through the Windows registry, it is possible to enable VBS without enabling HVCI. This will prevent the Microsoft Vulnerable Driver Blocklist setting from being automatically enabled too.

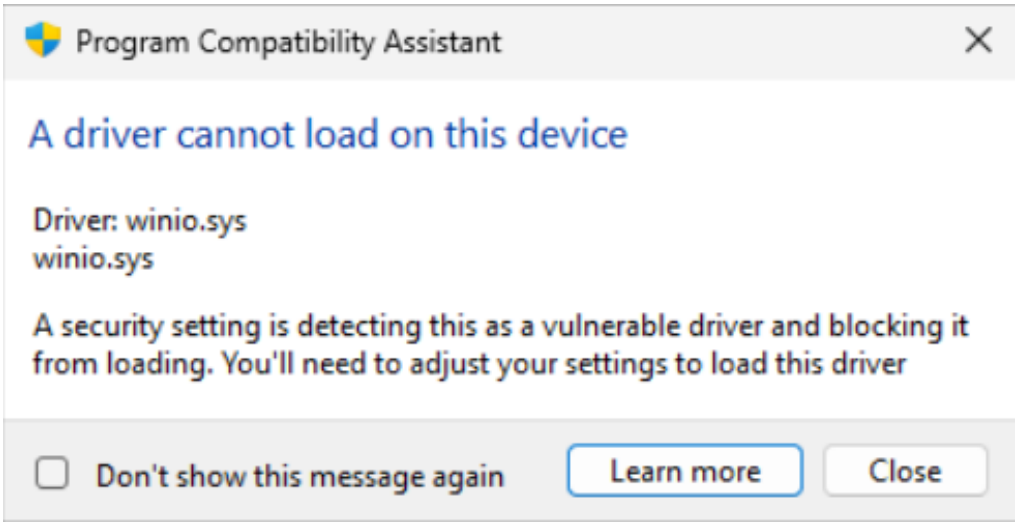


Figure 14: Error displayed when 'Microsoft Vulnerable Driver Blocklist' is enabled

After copying the `WinIo.sys` kernel driver to `C:\Windows\System32\drivers\WinIo.sys` and rebooting the VM, the driver is automatically enabled and running. In order to

test if KernelForge is working correctly, a `kforge_example.exe` executable is included in the GitHub repository. We have executed this binary and it prints that the `WinIo.sys` driver is correctly loaded, but it fails directly afterward. The `WinIo.sys` kernel driver only allows for physical memory access. However, KernelForge needs to have a virtual address. Therefore, an attempt is made to do address translation through the PML4 page map (see Section 2.1.3) but it fails to discover the PML4 page table address. The exact location where this fails is in the `DriverInitPageTableBase` function inside the for-loop of the `kforge_driver/kforge_driver.cpp` file. We tried to debug the issue and see if there is something wrong with the logic of the function, but it appears that the `WinIo.sys` kernel driver is not returning the expected values. We do not know why this is the case, but it may be related to running this in a VM under QEMU/KVM with certain hardware virtualization configurations missing. Since we think that fixing KernelForge would take a significant amount of time, we decided to switch plans and write our own custom kernel driver to interact with the NT Kernel and therefore the Secure Kernel (see Section 4.4.2).

4.4.2 Custom kernel driver

After a failed attempt at using KernelForge as the setup for interacting with the NT Kernel (and therefore Secure Kernel) (see Section 4.4.1), we decided to develop a custom Windows kernel driver ourselves to interact with the NT Kernel (and therefore Secure Kernel). This development has been done without any prior experience in developing Windows kernel drivers or Windows development in general. Before we developed the custom kernel driver, we had to make some changes to the setup described in Section 4.2.2 in order to be able to debug the custom kernel driver, which is described in Section 4.4.2.1. Section 4.4.2.2 describes the method we used to send Hypercalls to the Secure Kernel through the NT Kernel. Since we have no prior experience with developing a custom kernel driver, we spent about one week on the setup and development, including troubleshooting and debugging.

4.4.2.1 Custom kernel driver development setup

Since we have no prior experience in writing a custom Windows kernel driver, we first focused on getting a Windows kernel driver setup working so we can catch and debug potential bugs. We therefore made some changes to the QEMU/KVM setup as described in Section 4.2.2 to allow developing the custom kernel driver in the debugger VM and running the custom kernel driver in the debuggee VM. We found that the most helpful resources for setting up and developing a custom kernel driver come from the game cheating and modding community besides the Microsoft Learn online resources. The setup we used for developing the custom kernel driver is described in detail below. An overview of the setup is given in Figure 15. We temporarily increased the number of virtual CPU cores to 4 in order to speed up the setup process.

The first step for the custom kernel driver setup is to install the necessary software and create an empty Kernel Mode Driver (KMDF) project in Visual Studio. We then use 'Hello World' example kernel driver code in order to create a valid custom kernel driver. The installation and coding are done by following the 'Write a Hello World Windows Driver' tutorial from Microsoft Learn [23]. We encountered one issue in Visual Studio where it refuses to sign the compiled custom kernel driver, due to an 'access denied' error for the operation. Therefore, we had to disable signing in order to successfully compile the custom kernel driver.

The deployment of the custom kernel driver to the debuggee VM was challenging. We followed the Microsoft Learn article about 'provisioning a computer for driver deployment

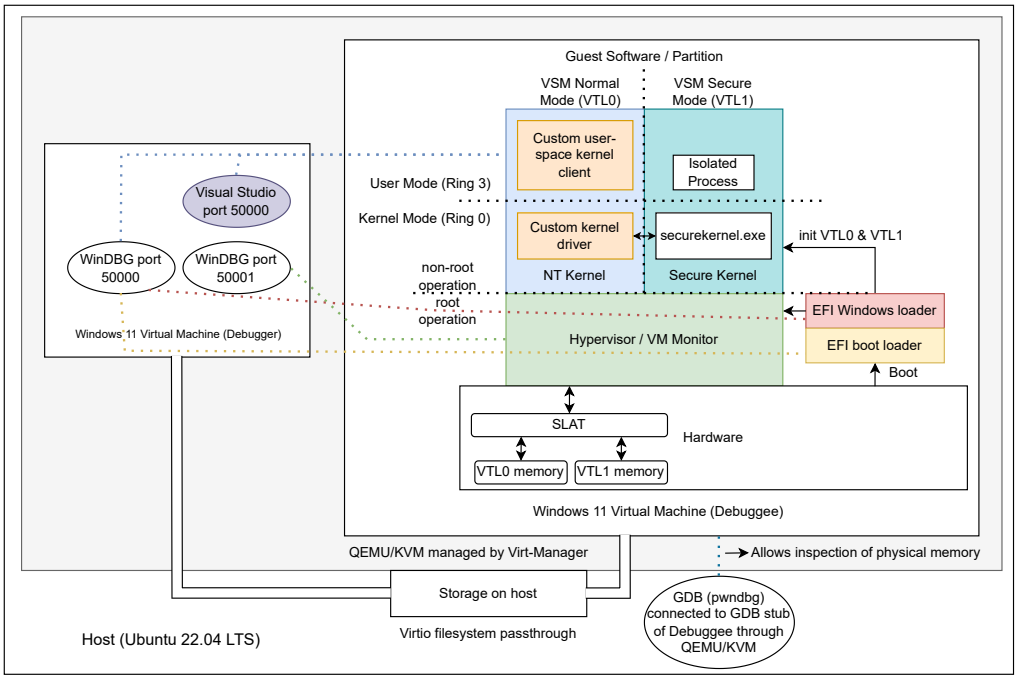


Figure 15: Overview of the development setup for the custom kernel driver

and testing' [22], which explains how to change some configurations within Windows and execute an MSI. The MSI is the setup installer for enabling custom kernel driver testing, which is copied from the debugger VM to the debuggee VM. Since copying files from the debugger VM to the debuggee VM will be very common, we made use of a shared folder between the debugger VM and debuggee VM using virtio [53]. The MSI will set up a new Windows user account and reboot the VM several times to change configurations. The last two installation steps failed without any clear reason. Since no information is given for troubleshooting, we decided to continue the provisioning. The last step of the provisioning is to connect Visual Studio to the debuggee through the kernel debugger VM. Since in Section 4.2.2 we already configured kernel debugging on the debuggee, this was trivial to configure.

Next, the solution needs to be deployed to the debuggee VM through Visual Studio running in the debugger VM. During this deployment, the spawning and closing `cmd.exe` terminals give the impression that something is failing. However, we continued the process of installing the driver as stated in the Microsoft Learn tutorial [23]. The built artifacts of the custom kernel driver are copied to the debuggee VM together with the `devcon.exe` utility. The custom kernel driver is then installed with the command `devcon.exe install SecureKernelCommunicator.inf root\SecureKernelCommunicator` which will show a popup as shown in Figure 16. After accepting the installation of the custom kernel driver, the custom kernel driver is loaded and executed. The custom kernel driver can be removed with the command `devcon.exe remove SecureKernelCommunicator.inf root\SecureKernelCommunicator` which requires a reboot of the debuggee VM.

The last step of the custom kernel driver development setup is to be able to debug the custom kernel driver. First, we looked into how we can view the output of print statements located in the custom kernel driver. A common technique [45] is to use the DebugView utility from the Sysinternals toolset [13] to listen to kernel driver logging.

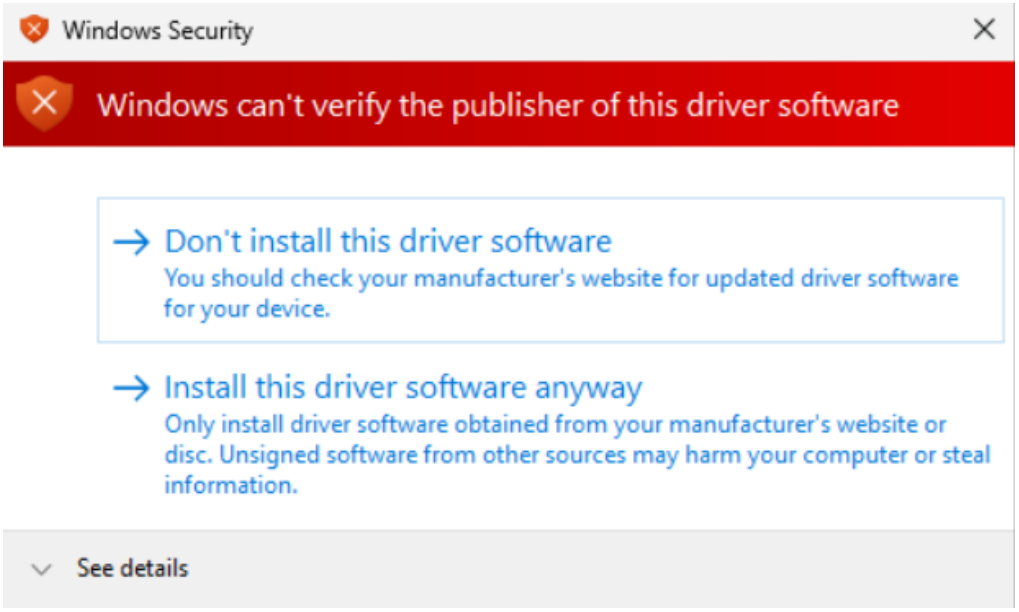


Figure 16: Dialog shown before installing the custom kernel driver

However, DebugView does not display any output and crashes without an error when trying to listen to kernel driver logging. Therefore, we cannot use DebugView. We replaced DebugView with WinDBG, since WinDBG can display kernel log messages if configured correctly. We use the QEMU/KVM setup as described in Section 4.2.2 to have a WinDBG session connected to the NT Kernel. Then we pause the VM execution and run the command `ed nt!Kd_Default_Mask 8` to change the logging verbosity of the NT Kernel in order to see the print statements from the custom kernel driver.

Since we have the NT Kernel running in a WinDBG session, any critical error (for example, a dereference error) will cause the NT Kernel to stop and display a message in WinDBG that the kernel has crashed. We can then analyze the error with the command `!analyze -v` in order to see where in the code the error occurred. This will help us identify and fix any issues in the custom kernel driver. With this setup, we are now able to develop, deploy, and debug our custom kernel driver to interact with the NT Kernel and Secure Kernel.

4.4.2.2 Sending Hypercalls to the Secure Kernel

As described in Section 4.4.2.1, we now have a development setup available to develop a custom kernel driver. Before we decided to fully write our own custom kernel driver, we made an attempt at reproducing earlier research on writing a Hyper-V 'bridge' for fuzzing purposes [37]. This blog post provides a lot of explanation and example code snippets to invoke a Hypercall. However, it does not provide a repository with the complete source code and all the required configurations. In the end, we were unable to reproduce the setup and code from the blog post and decided to write our own custom kernel driver.

Our approach is to communicate with the Secure Kernel through the NT Kernel, rather than calling the Secure Kernel directly. This mimics the method used by the NT Kernel itself for Secure Kernel communication. Our custom kernel driver implements the following:

1. Resolve the NT Kernel base address when loading the custom kernel driver.
2. Accept a Secure System Call Number (SSCN) and a custom payload from user-space through a Device Input and Output Control (IOCTL).
3. Call the NT Kernel function `VslpEnterIumSecureMode` with the given SSCN and payload to do the Hypercall.

In order to call the function `VslpEnterIumSecureMode`, we first need to know the NT Kernel base address since the function `VslpEnterIumSecureMode` is implemented in the NT Kernel. We found in a forum post on 'Unknown Cheats' a code snippet that resolves the NT Kernel base address based on the context of a kernel driver object [31]. Since we want to implement this in a custom kernel driver, this is a perfect match for our use case. During testing, we found that this code very reliably resolves the NT Kernel base address. However, when loading the custom kernel driver after boot, sometimes it may cause a crash and the debuggee VM will stop responding. In such a case, a reboot is necessary in order to try to successfully load the NT Kernel base address again. This crash is caused due to an invalid name string compared to the string 'ntoskrnl.exe', which causes dereference issues. Since we experienced this crash only when loading the custom kernel driver after boot (while normally it would load during boot), we did not put any effort into fixing this bug.

Second, we need to accept user-space input from a client binary that communicates with the custom kernel driver. This communication is done through the normal APIs used to communicate between user-space and kernel drivers, which are the 'Device Input and Output Control' APIs. We have implemented a handler using this API which reads the SSCN and a pointer to the extra data corresponding to the SSCN (the 'payload') which are both sent to the kernel driver. Besides this handler, we have also implemented logic to close and unload the driver. This is boilerplate code for a Windows kernel driver and has been implemented based on Windows kernel driver development tutorials [72] [5].

Last, using the resolved NT Kernel base address, we need to call the NT Kernel function `VslpEnterIumSecureMode`. `VslpEnterIumSecureMode` is responsible for sending Hypercalls to the Secure Kernel (see Section 2.3.2). We want to call the function according to the following signature: `VslpEnterIumSecureMode(2, sscn, 0, payload );`. Since the `VslpEnterIumSecureMode` function is a non-exported function, we can only call it by computing the exact position of the function within the `ntoskrnl.exe` binary. Therefore, we need to check in a disassembler at which offset the function `VslpEnterIumSecureMode` is located. This is trivial since Microsoft provides debugging symbols for the `ntoskrnl.exe`, which can be automatically discovered when the binary is imported into IDA. Note that due to the hardcoded offset of the `VslpEnterIumSecureMode` function in the custom kernel driver, the custom kernel driver will only work on systems where the correct version of the `ntoskrnl.exe` binary is running. The custom kernel driver needs to be recompiled with an updated offset to support different systems with a different `ntoskrnl.exe` binary running.

We now have a working setup for sending Hypercalls to the Secure Kernel through our custom kernel driver. During Section 5, we can further extend the custom kernel driver in order to interact with logic corresponding to the CVE being assessed, if necessary.

5 Exploitability Assessment

In this section, we assess the exploitability of security bugs in the Secure Kernel. We analyze Secure Kernel patches and to try to determine if a security bug is exploitable when it is not patched. Furthermore, in case we think that a bug is exploitable, we write code to trigger the security bug which works as a proof-of-concept trigger script on which we can build an exploit. In Section 6 we will build further on the proof-of-concept trigger script to try to create a full exploit for the security bug. However, the scope of this section is just the patch analysis and proof-of-concept trigger script.

We use patch diffing (see Section 3.1.1) to discover the security bugs related to CVEs. Furthermore, we also use dynamic code analysis in order to better understand the security bug and to have a proof-of-concept of triggering the security bug. We make use of Diaphora [39] as the binary diffing tool (see Section 3.1). The settings used for Diaphora are the default export settings, so without any modifications. See Figure 17 for the exact configuration.

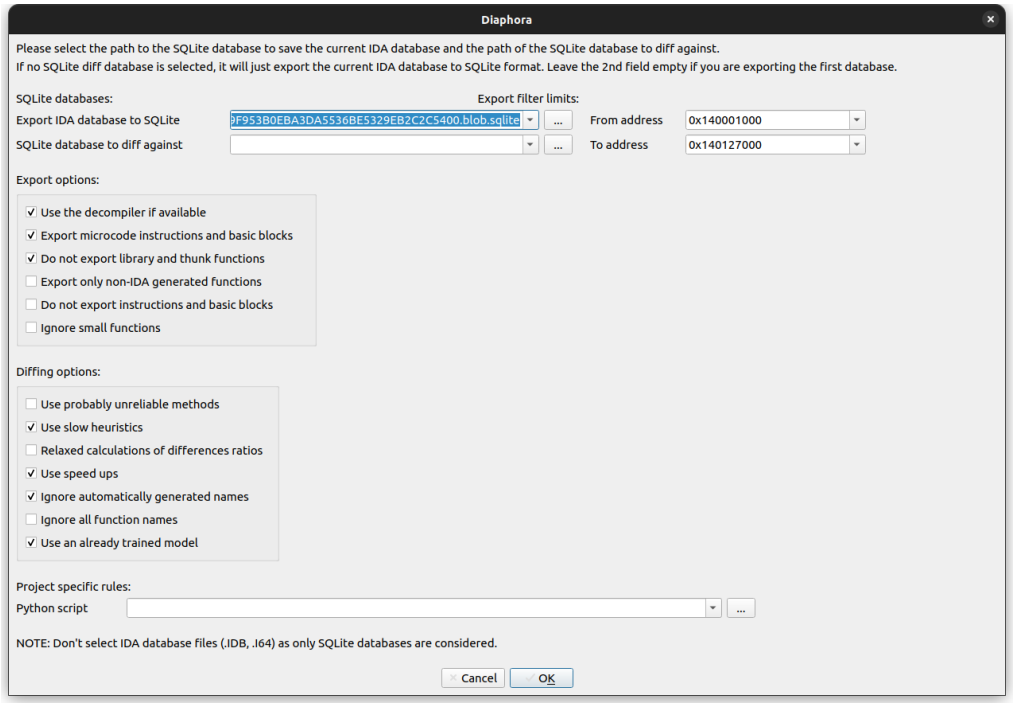


Figure 17: Settings used for patch diffing with Diaphora

In Section 5.1, we assess CVE-2024-43528 [4]. CVE-2024-43528 has been chosen since it is a recently patched bug in the Secure Kernel for which there is a Windows update available. Furthermore, Microsoft Security Response Centre classified this bug as a 'Heap-based Buffer Overflow' which may be a powerful enough primitive to exploit. However, in the end it turned out that it was not possible to trigger the bug related to CVE-2024-43528 due to unsupported functionalities when using the virtualized setup. In Section 5.1.3 we discuss our findings about mistakes with patch assignment of Microsoft for CVE-2024-43528. We then decide in Section 5.2 to stop looking at specific CVEs and analyze all Secure Kernel patches done for Windows 11 Version 22H2 for x64-based Systems. Through that we did find an n-day vulnerability, for which we developed a custom Enclave (see Section 5.2.2) to interact with the corresponding Secure Kernel

API (see Section 5.2.3). We then developed a proof-of-concept trigger script in Section 5.2.4 to trigger the bug. By analyzing this n-day vulnerability, we also found a zero-day vulnerability similar to the n-day vulnerability we discovered in the Secure Kernel, which is discussed in Section 5.3.

We make use of the setup as described in Section 4 for dynamic code analysis. However, the analysis of every CVE will first be done by using manual code analysis since we first need to identify the code that is responsible for the security bug by using patch diffing.

5.1 Exploitability Assessment of CVE-2024-43528

The goal of this section is to assess the exploitability of CVE-2024-43528. CVE-2024-43528 is a Heap-based Buffer Overflow vulnerability in the Secure Kernel. This vulnerability was patched in the October 8th, 2024 Windows update [4]. According to the published report of Microsoft Security Response Center, the Exploit Code maturity is 'Unproven' which means that there is no public exploit available or an exploit is theoretical. In order to analyze if this bug is exploitable or not, we use patch diffing (see Section 3.1.1) to uncover the bug.

5.1.1 Manual Code Analysis for Exploitability Assessment of CVE-2024-43528

As described in Section 4.1, we analyze the binaries of Windows 11 Version 22H2 for x64-based Systems. The corresponding KB-number for this patch is KB5046633. Using this KB-number, we can find the patched `securekernel.exe` version and the previous version on Winbindex [43]. For convenience, we name the patched version `post-securekernel.exe` and the unpatched version `pre-securekernel.exe`. We then use patch diffing on the two binaries as described in Section 3.1.1 and we analyze the results manually.

The results of the patch diffing yield exactly one changed function within the `post-securekernel.exe` binary: the `SkmmRegisterFailureLog` function has been changed. According to Appendix A, `SkmmRegisterFailureLog` corresponds to Secure System Call Number (SSCN) 252. The results can be found in Figure 18 and Figure 19 where red indicates the removed code from `pre-securekernel.exe`, green indicates the new code from `post-securekernel.exe` and yellow indicates irrelevant changes between the pseudocode of the two binaries.

```
__int16 v57; // [rsp+28h] [rbp-8h]
__int64 v58; // [rsp+28h] [rbp-8h]
char v60; // [rsp+80h] [rbp+50h]
unsigned __int64 v61; // [rsp+88h] [rbp+58h] BYREF

v2 = a2;
v4 = 0LL;
if ( !(unsigned int)SkmiClaimPhysicalPage(a2, 32770) )
  return 3221225496LL;
LOBYTE(v6) = SkAcquireSpinLockExclusive((__int64)&SkmiFailureLogLock);
v60 = v6;
LOBYTE(v7) = 2;
if ( SkmiFailureLog )
{
  v8 = -1073741800;
  goto LABEL_70;
}
v57 = a1;                                   // THIS IS REMOVED
v9 = a1 & 0xFFF;
```

```
__int64 v57; // [rsp+28h] [rbp-8h]

char v59; // [rsp+80h] [rbp+50h]
unsigned __int64 v60; // [rsp+88h] [rbp+58h] BYREF

v2 = a2;
v4 = 0LL;
if ( !(unsigned int)SkmiClaimPhysicalPage(a2, 32770) )
  return 3221225496LL;
LOBYTE(v6) = SkAcquireSpinLockExclusive((__int64)&SkmiFailureLogLock);
v59 = v6;
LOBYTE(v7) = 2;
if ( SkmiFailureLog )
{
  v8 = -1073741800;
  goto LABEL_70;
}

v9 = a1 & 0xFFF;
```

Figure 18: First difference after patching CVE-2024-43528

In short, the following changes can be seen in the pseudocode:

1. `v57` has been removed and replaced by `v9`. Note that `v57` refers to `a1` directly while `v9` refers to `a1 & 0xFFF`.

```
99    *(_QWORD *)v13 = v15;
100   v19 = v9 + (v13 << 25 >> 16);
101   v20 = *(_DWORD *)(v19 + 40);
102   if ( *(_DWORD *)(v19 + 44) || (v20 & 0xFFF) != 0 ||
      (v21 = v20 >> 12, ((v57 + 48) & 0xFFF) + 8 * v21 > 0x1000) )// VULN IS HERE
103   {
104     v8 = -1073741811;
105   }
106   else
107   {
108     if ( v21 > 2 )
109       v21 = 2;
```

```
97    *(_QWORD *)v13 = v15;
98    v19 = v9 + (v13 << 25 >> 16);
99    v20 = *(_DWORD *)(v19 + 40);
100   if ( *(_DWORD *)(v19 + 44)
101     || (v20 & 0xFFF) != 0
102     || (v21 = v20 >> 12, (unsigned __int64)(8 * v21) + v9 + 48 > 0x1000) )
103   {
104     v8 = -1073741811;
105   }
106   else
107   {
108     if ( v21 > 2 )
109       v21 = 2;
```

Figure 19: Second difference after patching CVE-2024-43528

2. `v57` is of size 32-bit while `v9` is of size 64-bit.

Furthermore, we have observed the following two interesting changes:

1. Due to the removal of the `v57` variable, the if-statement is updated to use the `v9` variable. Earlier in the code flow, `v9 + 48` is checked to be less than or equal to 4096 (which is `(a1 & 0xFFF)+ 48`). In the code of `post-securekernel.exe`, this check has already occurred while in `pre-securekernel.exe` this check has not been done on `v57 + 48` (which is `a1 + 48`).
2. The assembly has been changed to compute the values for comparison in the if-statement in a different way. This change impacts the referencing of the `r14` register as well as the `rax` register.

Below the assembly instructions can be found for the if-statement as can be seen in Figure 19 for the `pre-securekernel.exe` binary.

```
    add     r13, rbx
    mov     r14d, [r13+28h]
    cmp     [r13+2Ch], edi
    jnz     loc_14004478C
loc_14004446a:
    test    r14d, 0FFFh
    jnz     loc_14004478C
loc_140044477:
    mov     rax, [rbp+var_8]
    add     eax, 30h ; '0'
    shr     r14d, 0Ch
    and     eax, 0FFFh
    lea     eax, [rax+r14*8]
    cmp     eax, 1000h
```

Here `var_8` refers to the removed `v57` variable in the pseudocode. Below the assembly instructions can be found for the if-statement as seen in Figure 19 for the `post-securekernel.exe` binary.

```
    add     r13, rbx
    mov     r14d, [r13+28h]
    cmp     [r13+2Ch], edi
    jnz     loc_140044784
loc_140044462:
    test    r14d, 0FFFh
    jnz     loc_140044784
```

```
loc_14004446f:
    shr     r14d, 0Ch
    lea     rax, [rbx+30h]
    lea     ecx, ds:0[r14*8]
    add     rax, rcx
    cmp     rax, 1000h
```

The major difference between these two snippets of assembly is the initialization of the last `lea` instruction of the `pre-securekernel.exe`. The `lea` instruction - Load effective address - loads the address of the memory location and stores the value in the first operand. This is useful for obtaining a pointer into a memory region or to perform simple arithmetic operations. The contents of the memory location are not loaded. In `pre-securekernel.exe` the `lea` instruction is done using `eax, [rax+r14*8]` for which the lower 32 bits of the `rax` register will be overwritten with the loaded value and the upper 32 bits will be set to zero. The loading is done using two 64-bit registers: `rax` and `r14`. Just before the loading, the `and eax, 0FFFh` instruction makes sure that the `rax` register is within the range 0-0xfff. In `post-securekernel.exe` the instruction `eax, [rax+r14*8]` is divided into two `lea` instructions:

1. `lea rax, [rbx+30h]`
2. `lea ecx, ds:0[r14*8]`

The addition is therefore not done anymore within the `lea` instruction itself, but explicitly done afterwards using `add rax, rcx`. There is one last important observation in the difference between the two binaries: in `post-securekernel.exe` the `v9` variable is checked within the if-statement. This check is not done in the `pre-securekernel .exe` code. The variable `v19` relies on the value of `v9` and is later used as an offset in a heap buffer. The offset is the difference computed between `v19` and the result of `SkmiAllocateSystemPtes`.

We assume the vulnerability lies in the `r14` register. This is a reasonable assumption since after the if-statement succeeds, code will be executed that performs memory reads and writes in a do-while loop. This do-while loop is based on the value of `r14`. However, the value of `r14` is checked afterward using the following assembly code for both binaries:

```
    ja      loc_14004478C
    cmp     r14d, esi
    lea     rcx, SkmiNonPagedPtes ; global variable
    mov     edx, esi
    cmova   r14d, esi
```

Here `esi` is initialized to the constant 2. This code checks if the `r14d` register has a bigger value than 2, and if so sets the value of `r14d` to 2. This check prevents the `r14` register from being a different value than 0, 1, or 2. Therefore, the assumption that the vulnerability lies in the `r14` register does not hold, since the do-while loop can only be executed with the specific values 0, 1, or 2 and therefore cannot perform unexpected memory manipulations through the value of `r14`.

At this point, we concluded that understanding the vulnerability through manual code analysis seems non-trivial. We suspect that this vulnerability has been found using a fuzzer or other automation instead of manual code review. Therefore, we decided to introduce dynamic code analysis using the setup described in Section 4.2. Using a debugger, it may become easier to spot the vulnerability by inspecting concrete memory values to better understand the purpose of the variables.

5.1.2 Dynamic Code Analysis for Exploitability Assessment of CVE-2024-43528

As concluded in Section 5.1.1, through only manual code analysis of the code related to the vulnerability, understanding the bug seems non-trivial. Therefore, we decided to make use of dynamic code analysis to try to inspect the state of the registers and memory in order to understand the bug. We make use of the QEMU/KVM setup as described in Section 4.2.2 for debugging purposes and we use the setup described in Section 4.4 to interact with the Secure Kernel. In the case of CVE-2024-43528, we interact with Secure System Call Number (SSCN) 252. In order to understand what the expected inputs are, we will first investigate the normal use-case of SSCN 252.

```
__int64 VslRegisterLogPages()
{
  int v0; // ebx
  __int64 result; // rax
  unsigned __int8 CurrentIrql; // bl
  _DWORD *SchedulerAssist; // r9
  int v4; // eax
  unsigned int v5; // esi
  unsigned __int8 v6; // cl
  struct _KPRCB *CurrentPrcb; // r9
  _DWORD *v8; // r8
  int v9; // eax
  bool v10; // zf
  _QWORD v11[10]; // [rsp+30h] [rbp-D8h] BYREF
  _QWORD v12[14]; // [rsp+80h] [rbp-88h] BYREF

  v0 = PspIumLogBuffer;
  memset(v12, 0, 0x68uLL);
  memset(v11, 0, 0x48uLL);
  if ( !(unsigned __int8)HvlQueryVsmConnection(0LL) )
    return 3221225629LL;
  result = VslpLockPagesForTransfer((unsigned int)v11, v0, 0x2000, 1, 2);
  if ( (int)result >= 0 )
  {
    CurrentIrql = KeGetCurrentIrql();
    __writecr8(2uLL);
    if ( KiIrqlFlags && (KiIrqlFlags & 1) != 0 && CurrentIrql <= 0xFu )
    {
      SchedulerAssist = KeGetCurrentPrcb()->SchedulerAssist;
      v4 = 4;
      if ( CurrentIrql != 2 )
        v4 = (-1LL << (CurrentIrql + 1)) & 4;
      SchedulerAssist[5] |= v4;
    }
    v12[1] = v11[0];
    v12[2] = v11[7];
    v5 = VslpEnterIumSecureMode(2u, 252, 0, (__int64)v12);
    if ( KiIrqlFlags )
    {
      v6 = KeGetCurrentIrql();
      if ( (KiIrqlFlags & 1) != 0 && v6 <= 0xFu && CurrentIrql <= 0xFu && v6 >= 2u )
      {
        CurrentPrcb = KeGetCurrentPrcb();
        v8 = CurrentPrcb->SchedulerAssist;
        v9 = ~(unsigned __int16)(-1LL << (CurrentIrql + 1));
        v10 = (v9 & v8[5]) == 0;
        v8[5] &= v9;
        if ( v10 )
          KiRemoveSystemWorkPriorityKick(CurrentPrcb);
      }
    }
    __writecr8(CurrentIrql);
    VslpUnlockPagesForTransfer(v11);
    return v5;
  }
  return result;
}
```

Figure 20: Decompilation of the '`VslRegisterLogPages`' function in the `ntoskrnl.exe` binary, calling SSCN 252

Interaction with the Secure Kernel is done through the NT Kernel. Therefore, we

decided to inspect the `ntoskrnl.exe` which implements the NT Kernel. The SSCN calls from VTL0 to VTL1 are done through the '`VslpEnterIumSecureMode`' function which takes as the second argument the SSCN [7]. By inspecting every cross-reference to the `VslpEnterIumSecureMode` function in `ntoskrnl.exe`, we have found a single call to `VslpEnterIumSecureMode` which uses SSCN 252. In Figure 20, the decompilation can be found of the '`VslRegisterLogPages`' function implemented in the `ntoskrnl.exe` binary, which implements the logic for the normal use-case of SSCN 252.

As can be seen in Figure 20, the SSCN 252 expects two parameters which are stored in the memory of the variable `v12`. The two parameters are coming from the function `VslpLockPagesForTransfer`. There appears to be no public documentation about the purpose of the function `VslpLockPagesForTransfer`.

By following the steps as described in Section 4.2.2, we have set a breakpoint at the start of the function `SkmmRegisterFailureLog` in the Secure Kernel. We have observed that during normal boot and execution of the Windows operating system, this function is only called once during boot. Therefore, this means that the SSCN call from Figure 20 is only called once during normal boot and execution of the Windows operating system. We have modified the user-mode client binary to replicate the behavior of the SSCN call, but since we do not know the values in `v12`, we use dummy values instead.

By inserting several hardware breakpoints in the `SkmmRegisterFailureLog` function, we can trace the execution flow of our own Hypercall with SSCN 252 to the Secure Kernel. We observed that the vulnerable code mentioned in Section 5.1.1 is not directly reachable by sending a Hypercall with SSCN 252. After the first SSCN 252 call, the global variable `SkmiFailureLog` is initialized. When `SkmiFailureLog` is already initialized, the code cannot be executed again due to a check at the start of the `SkmmRegisterFailureLog` function. Therefore, we have to find a way to set `SkmiFailureLog` to zero in order to be able to reach the vulnerable code again. By inspecting the cross-references to `SkmiFailureLog`, we have found that in the function `SkmmFreeFailureLog` the global variable `SkmiFailureLog` is set to zero. The `SkmmFreeFailureLog` function is only called once, which is in the `SkPrepareForHibernate` function. The `SkPrepareForHibernate` function is again called from the `IumInvokeSecureService` function through SSCN 259.

Since we want to replicate the call to `SkPrepareForHibernate` with SSCN 259 in order to set `SkmiFailureLog` to zero, we need to know which parameter(s) the Hypercall expects. We assumed that this Hypercall is coming from the NT Kernel again, and therefore we searched for all functions with 'hibernate' in the name. Of the results, the function `HvlPrepareForSecureHfunction is called only once fromibernate` stood out. The decompilation of this function can be seen in Figure 21. Inspecting this function shows that it indeed does a call to the NT Kernel function `VslpEnterIumSecure Mode` with SSCN 259. The `HvlPrepareForSecureHibernate` function is called only once from the `PopSaveHiberContext` function, which calls `HvlPrepareForSecureHibernate` with four 128-bit globals combined into a single parameter.

We need to reproduce the logic of `PopSaveHiberContext` in order to successfully call the `HvlPrepareForSecureHibernate` function. The struct that is required as the first parameter to call `HvlPrepareForSecureHibernate` consists of an array of several 128-bit global values within the NT Kernel. Since implementing this in the user-space binary will become complex due to the specific requirements to call the `HvlPrepareForSecure Hibernate` function, we implemented this logic in the custom kernel driver itself. By using the NT Kernel base address, we can resolve the locations of the global variables and reconstruct the array.

After implementing this and loading the updated kernel driver into the debuggee VM,

```
__int64 __fastcall HvlPrepareForSecureHibernate(_OWORD *a1)
{
  _OWORD *v2; // rax
  __int128 v3; // xmm1
  __int128 v4; // xmm0
  __int128 v5; // xmm1
  __int128 v6; // xmm0
  __int128 v8; // [rsp+20h] [rbp-A8h] BYREF
  __int64 v9; // [rsp+30h] [rbp-98h]
  unsigned __int64 v10; // [rsp+38h] [rbp-90h]
  _QWORD v11[14]; // [rsp+40h] [rbp-88h] BYREF

  v9 = 0LL;
  LODWORD(v10) = 0;
  v8 = 0LL;
  v2 = (HvlpAcquireHypercallPage)(&v8, 1LL, 0LL, 88LL);
  v3 = a1[1];
  *v2 = *a1;
  v4 = a1[2];
  v2[1] = v3;
  v5 = a1[3];
  v2[2] = v4;
  v6 = a1[4];
  v2[3] = v5;
  v2[4] = v6;
  memset(v11, 0, 0x68uLL);
  v11[1] = v10 >> 12;
  LODWORD(a1) = VslpEnterIumSecureMode(2u, 259, 0, v11);
  HvlpReleaseHypercallPage(&v8);
  return a1;
}
```

Figure 21: Decompilation of the 'HvlPrepareForSecureHibernate' function in the ntoskrnl.exe binary, calling SSCN 259

we have inserted several hardware breakpoints to check if the SkmmFreeFailureLog function in the Secure Kernel is executed. This appeared to be not the case. However, the SkPrepareForHibernate function is called. By further inserting hardware breakpoints and debugging the execution flow, we have found that a constraint is not met and therefore the SkmmFreeFailureLog function is not executed. In Figure 22, the decompilation of the SkPrepareForHibernate function can be found. As can be seen in the decompilation, there are two checks done before the SkmmFreeFailureLog function is called: SkpValidateHiberCrashCaller(1) and SkeEnterRestrictedMode(). Through debugging, we found that the SkpValidateHiberCrashCaller(1) function returns zero, which prevents code execution from reaching the SkmmFreeFailureLog function.

In Figure 23, the decompilation of the SkpValidateHiberCrashCaller function can be found. Our goal is to return this function the value 1 in order to continue execution in the SkPrepareForHibernate function. Since SkpValidateHiberCrashCaller is called with value 1, the nested if-statement will determine the outcome of this function. By inserting hardware breakpoints on the comparison instructions, we found that the comparison KeGetCurrentIrql()<= 2u is satisfied. By searching for functions containing 'Irql' in the name, we found that there exists a SkeRaiseIrql function in the Secure Kernel and a KzRaiseIrql function in the NT Kernel. The KzRaiseIrql function has documentation

```
1  __int64 __fastcall SkPrepareForHibernate(__int64 a1)
2  {
3    __int64 result; // rax
4    unsigned __int8 CurrentIrql; // di
5    __int64 v3; // rcx
6    int v4; // ebx
7    signed __int32 v5[8]; // [rsp+0h] [rbp-89h] BYREF
8    __int64 v6; // [rsp+30h] [rbp-59h] BYREF
9    _QWORD v7[10]; // [rsp+40h] [rbp-49h] BYREF
10   _BYTE v8[24]; // [rsp+90h] [rbp+7h] BYREF
11   __int128 *v9; // [rsp+A8h] [rbp+1Fh]
12   __int64 v10; // [rsp+B0h] [rbp+27h]
13
14   v6 = a1;
15   memset_0(v8, 0, 0x40uLL);
16   if ( !(unsigned int)SkpValidateHiberCrashCaller(1) )
17     return 0xC0000022LL;
18   result = SkeEnterRestrictedMode();
19   if ( (int)result >= 0 )
20   {
21     memset_0(&xmmword_FFFFF8027C053418, 0, 0x40uLL);
22     SkmmFreeFailureLog();
```

Figure 22: Decompilation of the '`SkPrepareForHibernate`' function in the Secure Kernel

```
1  __int64 __fastcall SkpValidateHiberCrashCaller(int a1)
2  {
3    struct _EXCEPTION_REGISTRATION_RECORD *ExceptionList; // rax
4
5    ExceptionList = KeGetPcr()->NtTib.ExceptionList;
6    if ( a1 )
7    {
8      if ( WORD2(ExceptionList[10].Handler)
9        || KeGetCurrentIrql() <= 2u
10       || *((_DWORD *)KeGetPcr()->NtTib.StackBase + 42)
11       || (ShvlpFlags & 1) == 0
12       || IumHiberCrashContext != 2 )
13     {
14       return 0LL;
15     }
16   }
17   else if ( ExceptionList != (struct _EXCEPTION_REGISTRATION_RECORD *)qword_FFFFF8027C053508
18         || IumHiberCrashContext != 3 )
19   {
20     return 0LL;
21   }
22   return 1LL;
23 }
```

Figure 23: Decompilation of the '`SkpValidateHiberCrashCaller`' function in the Secure Kernel

available on Microsoft Learn[9] which states that the function increases the Interrupt Request Level (IRQL) for the current processor. According to Visual Studio, however, this function is not available even after importing the required `wdm.h` header file. Instead, the auto-complete suggested a different function named similarly: `KfRaiseIrql`. We boxed the call to `HvlPrepareForSecureHibernate` with a call to `KIRQL CurrentIrql = KfRaiseIrql(3);` before and a call to `KeLowerIrql(CurrentIrql);` after. After updating the custom kernel driver in the debuggee VM with the new version, we indeed do not satisfy the `KeGetCurrentIrql()<= 2u` comparison anymore.

[9] `https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-kzraiseirql`

```
__int64 __fastcall SkAllocateHibernateResources(_QWORD *a1, unsigned __int64 *a2)
{
  __int64 result; // rax
  unsigned int v5; // r10d
  unsigned __int64 v6; // rcx
  __int64 v7; // rax
  unsigned __int64 v8; // [rsp+60h] [rbp+18h]

  result = SkCheckHibernationSupport();
  if ( (int)result >= 0 )
  {
    result = SkpSetHiberCrashState(2, 0LL);
    if ( (int)result >= 0 )
    {
      if ( qword_FFFFF8027C053458
        && (int)SkPreparePageEncryption(3LL, 0LL, qword_FFFFF8027C053458, 0LL, 0LL, 0LL, 0LL) < 0 )
      {
        SkpSetHiberCrashState(1, 0LL);
      }
      else
      {
        v6 = SkpResumeInformation;
        *a1 = SkpResumeInformationPageCount;
        v7 = *(_QWORD *)(((v6 >> 9) & 0x7FFFFFFF8LL) - 0x98000000000LL);
        HIDWORD(v8) = HIDWORD(v7) & 0xFFFFF;
        LODWORD(v8) = (v6 & 0xFFF) + (v7 & 0xFFFFF000);
        *a2 = v8 >> 12;
        return 0;
      }
      return v5;
    }
  }
  return result;
}
```

Figure 24: Decompilation of the '`SkAllocateHibernateResources`' function in the Secure Kernel

However, through further debugging the execution flow in the `SkpValidateHiberCrashCaller` function, we found that the last check `IumHiberCrashContext != 2` is satisfied. By inspecting cross-references to the global variable `IumHiberCrashContext`, we found that a write to this variable is done in the `SkpSetHiberCrashState` Secure Kernel function. There is only one call to `SkpSetHiberCrashState` that sets the global variable `IumHiberCrashContext` to 2, which is done in the `SkAllocateHibernateResources` Secure Kernel function. See Figure 24 for the decompilation of the `SkAllocateHibernateResources` function. The `SkAllocateHibernateResources` function is only called from the `IumInvokeSecureService` function with SSCN 36. By checking all calls to `VslpEnterIumSecureMode` in the NT Kernel, we found that the `VslAllocateSecureHibernateResources` NT Kernel function uses SSCN 36 to call `SkAllocateHibernateResources` in the Secure Kernel. The `VslAllocateSecureHibernateResources` requires a 'MemoryMap' struct as the only parameter to the function. Since we do not know how the MemoryMap structure looks, we inspected the only cross-reference to this function, which is in the `PopAllocateHiberContext` NT Kernel function. The `PopAllocateHiberContext` function does a lot of operations on the MemoryMap and calls in the end the `VslAllocateSecureHibernateResources` function. Since reimplementing this functionality in our custom kernel driver becomes very difficult, we instead stopped at this point and reflected on our current progress for investigating CVE-2024-43528.

We timeboxed the dynamic code analysis of CVE-2024-43528 to one week. We did this to prevent going too deep into the bug and spending too much time on something that may be too complex to exploit or too difficult to reproduce. Furthermore, there are more CVEs in the Secure Kernel that can be analyzed and therefore we did not want to spend too much time on a single CVE. After analyzing the `PopAllocateHiberContext` function, we reached our time limit and therefore we will now reflect on the current progress and the conclusions we can already make.

We already concluded that the logic related to the bug in `SkmmRegisterFailureLog`

can be triggered only once, during the boot sequence of Windows. Afterwards, the functionality is locked until a global variable is set to zero. Therefore, it is not trivial to reach the functionality of CVE-2024-43528.

Furthermore, we investigated how we can set the global variable to zero. This appeared to be a difficult task, since a lot of constraints must be met before this is possible. In order to set these constraints, functionality related to 'hibernation' must be used. However, we realized that our debuggee VM does not support Windows Hibernation due to the virtualized setup. The Windows Hibernation functionality is probably related to all these functionalities we found in the NT Kernel and Secure Kernel due to the naming of the functions and global variables. Windows Hibernation is not supported because, according to Windows, the firmware does not support hibernation. By searching online, we came across posts of people that also experience the issue that hibernation is not supported in QEMU/KVM VMs [40]. The proposed solution for this is to disable Hyper-V. This is however not a solution for us, since we require the Virtual Secure Mode (VSM) Hyper-V component to be enabled in order to allow the Secure Kernel to be loaded during boot.

From this, we concluded that it is not worth looking anymore at this CVE, since it appears that it is not possible to trigger the bug with our current setup.

5.1.3 Patch Assignment Issues for CVE-2024-43528 / KB5046633

In Section 5.1.2 we have concluded that we were not able to trigger the bug in our QEMU/KVM virtualized setup as described in Section 4.2.2. Therefore, we are going to look into another CVE. We made an overview of the last eight CVEs in the Secure Kernel to better understand which patches can be analyzed, which can be found in Table 5. After creating this overview, we noticed something very peculiar: many CVEs were assigned the same KB-number '5046633'. This includes the patch for CVE-2024-43528 which we analyzed in Section 5.1. However, in the patch examined in Section 5.1 we have seen that there are only a few lines of code changed. This seems to be a mistake on the part of Microsoft, since it looks impossible that there are five different CVEs in the code of Figure 18 and Figure 19.

CVE	**CWE**	**KB**
CVE-2025-21325	CWE-732: Incorrect Permission Assignment for Critical Resource	5050021
CVE-2024-43646	CWE-822: Untrusted Pointer Dereference	5046633
CVE-2024-43640	CWE-415: Double Free	5046633
CVE-2024-43631	CWE-822: Untrusted Pointer Dereference	5046633
CVE-2024-43528	CWE-122: Heap-based Buffer Overflow	5046633
CVE-2024-43516	CWE-822: Untrusted Pointer Dereference	5046633
CVE-2024-38142	CWE-122: Heap-based Buffer Overflow	5041585
CVE-2024-21302	CWE-284: Improper Access Control	5041585

Table 5: List of eight Secure Kernel CVEs with corresponding details for Windows 11 Version 22H2 for x64-based Systems

Therefore, we decided to look into the patches released for different Windows versions. Up until now, we have only looked at binaries related to Windows 11 Version 22H2 for x64-based Systems since that is the operating system version installed on the debuggee VM. We first tried to look into the binaries of Windows 10 Version 22H2 for x64-based Systems which correspond to KB-number 5046613. This KB-number can be found and downloaded on Winbindex. However, the corresponding release contains another

knowledge base update. We used Diaphora to compare the binary with the previous version available on Winbindex (KB5044273) to patchdiff the binaries (see Section 3.1.1). The results can be seen in Figure 25.

Line	Address	Name	Address 2	Name 2	Ratio	BBlocks 1	BBlocks 2	Description
00008	1400241d0	NtQueryInformationToken	1400241d0	NtQueryInformationToken	0.9974561	28	28	Potential size check added
00033	14009ceb0	RtlCreateUserStack	14009ceb0	RtlCreateUserStack	0.9947501	31	31	Potential size check added
00000	140001360	ShvlSendSyntheticClusterIpiEx	140001360	ShvlSendSyntheticClusterIpiEx	0.9900000	12	12	Pattern 'mem'
00001	1400048ec	SkmiCompleteEnclaveContainer...	1400048ec	SkmiCompleteEnclaveContainer...	0.9900000	66	66	Potential size check added
00002	140006b7c	SkmmAccessFault	140006b7c	SkmmAccessFault	0.9900000	94	94	Potential size check added
00003	140010fcc	SkmiCopyKernelImagePage	140010fcc	SkmiCopyKernelImagePage	0.9900000	36	36	Potential size check added
00004	140011dd0	SkmiRelocateImagePage	140011dd0	SkmiRelocateImagePage	0.9900000	36	36	Potential size check added
00006	140023508	SkpHandleWppEvent	140023508	SkpHandleWppEvent	0.9900000	49	49	Potential size check added
00009	1400242fc	NtTraceControl	1400242fc	NtTraceControl	0.9900000	21	21	Pattern 'ProbeForWrite'
00010	140024844	SkmmQueryVirtualMemory	140024844	SkmmQueryVirtualMemory	0.9900000	113	113	Potential size check added
00011	140024e74	SkSyscall	140024e74	SkSyscall	0.9900000	30	30	Pattern 'mem'
00012	14002c640	SkpsInitializeProcess	14002c640	SkpsInitializeProcess	0.9900000	41	41	Pattern 'mem'
00014	14003e51c	SkmmRegisterFailureLog	14003e51c	SkmmRegisterFailureLog	0.9900000	62	64	Potential size check added
00016	1400416d0	_output_l	1400416d0	_output_l	0.9900000	187	186	Pattern 'mem'
00017	140051058	SkpspDispatchBasicEnclaveCall	140051058	SkpspDispatchBasicEnclaveCall	0.9900000	68	64	Pattern 'ProbeForWrite'
00022	14007d2d0	SkmiObtainStraddleData	14007d2d0	SkmiObtainStraddleData	0.9900000	14	14	Pattern 'mem'
00023	14007dd5c	SkmiApplyHotPatch	14007dd5c	SkmiApplyHotPatch	0.9900000	81	81	Pattern 'mem'
00025	140082d10	SkmiSetImageTracePoint	140082d10	SkmiSetImageTracePoint	0.9900000	96	96	Pattern 'mem'
00026	140086430	SkpgHotpatchLoadImage	140086430	SkpgHotpatchLoadImage	0.9900000	49	49	Pattern 'mem'
00028	14008cb08	NtQueryValueKey	14008cb08	NtQueryValueKey	0.9900000	11	11	Potential size check added
00029	14008dc48	vDbgPrintExWithPrefixInternal	14008dc48	vDbgPrintExWithPrefixInternal	0.9900000	18	18	Pattern 'mem'
00031	140099b50	SkProvisionDumpKeys	140099b50	SkProvisionDumpKeys	0.9900000	17	17	Pattern 'system'
00005	14002107c	NtQueryInformationProcess	14002107c	NtQueryInformationProcess	0.9871630	51	50	Potential size check added
00018	14005d988	NtSystemDebugControl	14005d988	NtSystemDebugControl	0.9867100	43	42	Potential size check added
00020	140064c40	KiMcheckAbort	140064c40	KiMcheckAbort	0.9863590	13	10	Potential size check added
00007	140023cdc	NtQuerySystemInformation	140023cdc	NtQuerySystemInformation	0.9861765	119	119	Pattern 'mem'
00032	140099ce8	SkiRaiseException	140099ce8	SkiRaiseException	0.9855385	25	23	Potential size check added
00027	14008b644	SkpHandleEtwEvent	14008b644	SkpHandleEtwEvent	0.9846405	53	52	Potential size check added
00019	14005ec04	SkpLoadDumpKeys	14005ec08	SkpLoadDumpKeys	0.9834671	20	20	Potential size check added
00021	14006e430	SkhalpPciAccessDeviceInternal	14006e430	SkhalpPciAccessDeviceInternal	0.9651964	26	26	Pattern 'mem'
00030	140096ea8	SkDecryptTrustletBoundData	140096ea8	SkDecryptTrustletBoundData	0.9623873	50	50	Pattern 'mem'
00024	140080de8	SkmiRevertHotPatch	140080de8	SkmiRevertHotPatch	0.9508558	20	20	Pattern 'mem'
00013	14002cd64	SkpspCreateSecureProcessPara...	14002cd64	SkpspCreateSecureProcessPara...	0.9344196	9	9	Pattern 'mem'
00015	140041390	_vsnprintf_l	140041390	_vsnprintf_l	0.8125000	11	10	Pattern 'mem'

Figure 25: Difference between update KB5046613 and KB5044273 of `securekernel.exe`

There are many differences between these two versions, which is in contrast to the binaries for Windows 11 Version 22H2 for x64-based Systems as discussed in Section 5.1.1. One of the updated functions is 'SkmmRegisterFailureLog' for which the same patch has been applied as analyzed in Section 5.1.1.

This difference has piqued our interest and therefore we decided to patch diff more binary updates related to CVE-2024-43528 for different Windows versions. We have also looked into the update for Windows 11 21H2 for x64-based Systems which corresponds to KB5044280. This update is also listed on Winbindex (as the last update available), but it contains an unknown file version. The release date of the update corresponds to patch Tuesday in October 2024, which is the correct date for CVE-2024-43528. However, downloading the previous version listed on Winbindex (which is KB5041592 and corresponds to patch Tuesday in September 2024), the download results in exactly the same binary as the update for KB5044280. Therefore these updates are identical and no difference can be computed. Since extracting Windows updates manually as described in Section 3.2 is very time-consuming, we decided to download the update released on patch Tuesday in August 2024, which is update KB5036894.

The result of patch diffing for CVE-2024-43528 between KB5036894 and KB5044280 in Windows 11 21H2 for x64-based Systems revealed only one changed function, which is 'IumInvokeSecureService'. This is not in line with the update we examined for Windows 10 Version 22H2 for x64-based Systems and Windows 11 Version 22H2 for x64-based Systems. We decided to continue patch diffing on the last ten updates for Windows 11 Version 22H2 for x64-based Systems to see if we can rediscover the IumInvokeSecureService patch. We found that this patch does exist for the update of KB5037853 to KB5041585. This is an update that was released on patch Tuesday in August 2024. This means that there are three mistakes made for Windows 11 21H2 for x64-based Systems:

1. The update for IumInvokeSecureService (patch between KB5036894 to KB5044280) was released one month later compared to Windows 11 Version 22H2 for x64-based

Systems.

2. The knowledge base reference in CVE-2024-43528 is incorrect.
3. CVE-2024-43528 has never been patched for Windows 11 21H2 for x64-based Systems.

That the patch has never been released for Windows 11 21H2 for x64-based Systems can be explained due to the end-of-life status of Windows 11 21H1. Windows 11 21H1 reached EOL in October 2023 and most likely Microsoft stopped updating in October 2024. However, this does not explain the incorrect knowledge base reference for CVE-2024-43528 as well as the one-month delayed patch.

In Table 6, we made a similar overview as in Table 5 for the eight CVEs, but now for Windows 11 Version 24H2 for x64-based Systems (instead of 22H2).

CVE	**CWE**	**KB**
CVE-2025-21325	CWE-732: Incorrect Permission Assignment for Critical Resource	5050009
CVE-2024-43646	CWE-822: Untrusted Pointer Dereference	5046617 / 5046696
CVE-2024-43640	CWE-415: Double Free	N/A
CVE-2024-43631	CWE-822: Untrusted Pointer Dereference	5046617 / 5046696
CVE-2024-43528	CWE-122: Heap-based Buffer Overflow	5047621 / 5046696
CVE-2024-43516	CWE-822: Untrusted Pointer Dereference	5047621 / 5046696
CVE-2024-38142	CWE-122: Heap-based Buffer Overflow	5041571
CVE-2024-21302	CWE-284: Improper Access Control	5041571

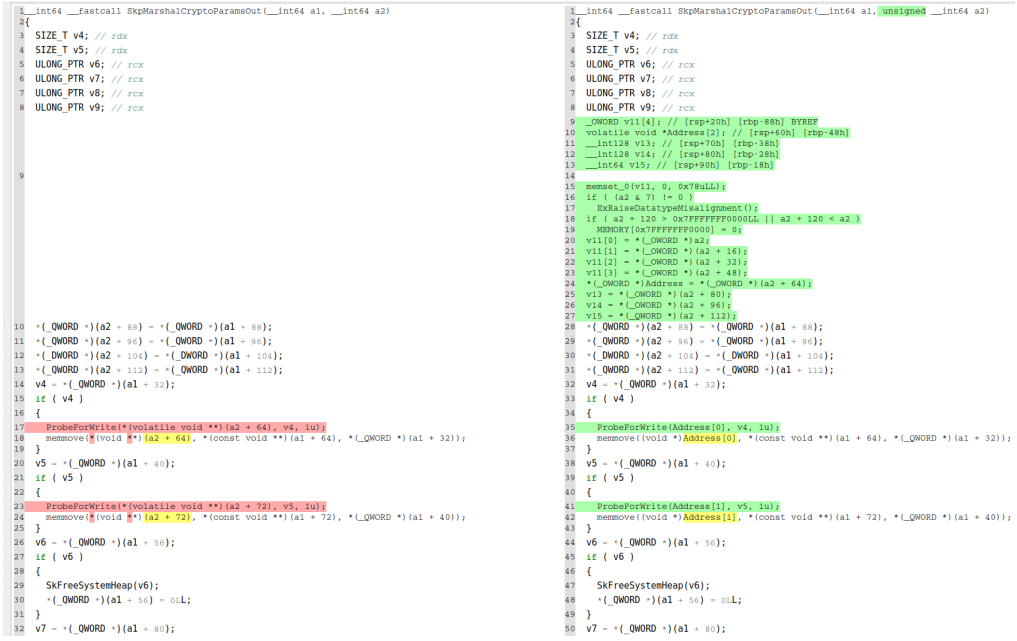
Table 6: List of eight Secure Kernel CVEs with corresponding details for Windows 11 Version 24H2 for x64-based Systems

The double KB-number assignment is due to the 'Security Update' and 'Security Hotpatch Update' available on this platform, which receive a separate KB-number assignment. As can be seen in Table 6, only a maximum of two CVEs share the same KB-number. Comparing this to Table 25, we conclude that it is a mistake that the update for Windows 11 Version 22H2 for x64-based Systems has the same KB-number for five different CVEs (see also Section 8.6).

5.2 Exploitability Assessment of Secure Kernel patches

In Section 5.1.3, we concluded that KB-numbers are an unreliable way to identify security patches for specific CVEs. Therefore, we decided to stop using this method. We think a broader search through several patches combined with manual code analysis will also work for identifying a security bug, which may be linked to a CVE and KB-number. We decided to download the oldest available `securekernel.exe` binary (available since September 2022) and the latest available `securekernel.exe` binary (available since April 2025, at the time of writing) for Windows 11 Version 22H2 for x64-based Systems from Winbindex [43] and perform patch diffing between these two binaries. This will yield all applied patches for Windows 11 Version 22H2 for x64-based Systems between the (at the time of writing) current release and the first release, which would certainly contain security patches. We then review every security patch and determine if the patch would be a security patch or a regular bug fix. In case we identify a security patch, we can use this security patch for the Exploitability Assessment.

Using this method, we identified a security patch in the update from KB5041585 to KB5044285. There is no CVE linked to this update. In Figure 26, the pseudocode



```
__int64 __fastcall SkpMarshalCryptoParamsOut(__int64 a1, __int64 a2)
{
  SIZE_T v4; // rdx
  SIZE_T v5; // rdx
  ULONG_PTR v6; // rcx
  ULONG_PTR v7; // rcx
  ULONG_PTR v8; // rcx
  ULONG_PTR v9; // rcx

  *(_QWORD *)(a2 + 88) = *(_QWORD *)(a1 + 88);
  *(_QWORD *)(a2 + 96) = *(_QWORD *)(a1 + 96);
  *(_DWORD *)(a2 + 104) = *(_DWORD *)(a1 + 104);
  *(_QWORD *)(a2 + 112) = *(_QWORD *)(a1 + 112);
  v4 = *(_QWORD *)(a1 + 32);
  if ( v4 )
  {
    ProbeForWrite(*(volatile void **)(a2 + 64), v4, 1u);
    memmove(*(void **)(a2 + 64), *(const void **)(a1 + 64), *(_QWORD *)(a1 + 32));
  }
  v5 = *(_QWORD *)(a1 + 40);
  if ( v5 )
  {
    ProbeForWrite(*(volatile void **)(a2 + 72), v5, 1u);
    memmove(*(void **)(a2 + 72), *(const void **)(a1 + 72), *(_QWORD *)(a1 + 40));
  }
  v6 = *(_QWORD *)(a1 + 56);
  if ( v6 )
  {
    SkFreeSystemHeap(v6);
    *(_QWORD *)(a1 + 56) = 0LL;
  }
  v7 = *(_QWORD *)(a1 + 80);
```

```
__int64 __fastcall SkpMarshalCryptoParamsOut(__int64 a1, unsigned __int64 a2)
{
  SIZE_T v4; // rdx
  SIZE_T v5; // rdx
  ULONG_PTR v6; // rcx
  ULONG_PTR v7; // rcx
  ULONG_PTR v8; // rcx
  ULONG_PTR v9; // rcx
  _OWORD v11[4]; // [rsp+20h] [rbp-88h] BYREF
  volatile void *Address[2]; // [rsp+60h] [rbp-48h]
  __int128 v13; // [rsp+70h] [rbp-38h]
  __int128 v14; // [rsp+80h] [rbp-28h]
  __int64 v15; // [rsp+90h] [rbp-18h]

  memset_0(v11, 0, 0x78uLL);
  if ( (a2 & 7) != 0 )
    ExRaiseDatatypeMisalignment();
  if ( a2 + 120 > 0x7FFFFFFF0000LL || a2 + 120 < a2 )
    MEMORY[0x7FFFFFFF0000] = 0;
  v11[0] = *(_OWORD *)a2;
  v11[1] = *(_OWORD *)(a2 + 16);
  v11[2] = *(_OWORD *)(a2 + 32);
  v11[3] = *(_OWORD *)(a2 + 48);
  *(_OWORD *)Address = *(_OWORD *)(a2 + 64);
  v13 = *(_OWORD *)(a2 + 80);
  v14 = *(_OWORD *)(a2 + 96);
  v15 = *(_QWORD *)(a2 + 112);
  *(_QWORD *)(a2 + 88) = *(_QWORD *)(a1 + 88);
  *(_QWORD *)(a2 + 96) = *(_QWORD *)(a1 + 96);
  *(_DWORD *)(a2 + 104) = *(_DWORD *)(a1 + 104);
  *(_QWORD *)(a2 + 112) = *(_QWORD *)(a1 + 112);
  v4 = *(_QWORD *)(a1 + 32);
  if ( v4 )
  {
    ProbeForWrite(Address[0], v4, 1u);
    memmove((void *)Address[0], *(const void **)(a1 + 64), *(_QWORD *)(a1 + 32));
  }
  v5 = *(_QWORD *)(a1 + 40);
  if ( v5 )
  {
    ProbeForWrite(Address[1], v5, 1u);
    memmove((void *)Address[1], *(const void **)(a1 + 72), *(_QWORD *)(a1 + 40));
  }
  v6 = *(_QWORD *)(a1 + 56);
  if ( v6 )
  {
    SkFreeSystemHeap(v6);
    *(_QWORD *)(a1 + 56) = 0LL;
  }
  v7 = *(_QWORD *)(a1 + 80);
```

Figure 26: Difference between update KB5041585 and KB5044285 for `SkpMarshalCryptoParamsOut` of `securekernel.exe`

difference can be found for this security patch. As can be seen in the patch, there is an update done to the parameters of `ProbeForWrite` and `memmove`. Instead of directly passing a value to both functions, a value is now first copied to the stack before using the value for `ProbeForWrite` and `memmove`. Furthermore, more values from the second argument of `SkpMarshalCryptoParamsOut` are copied to Secure Kernel memory before they are used. The purpose of `ProbeForWrite` is to check that a user-mode buffer actually is in a user-mode address space, the user-mode buffer is writable, and correctly aligned [21]. This function therefore gives a clear indication that there is some user-mode/kernel-mode interaction happening in this code path. We therefore think this is a Time-Of-Check Time-Of-Use (TOCTOU) vulnerability within the Secure Kernel since a pointer will be dereferenced twice:

1. The first dereference is for `ProbeForWrite` to perform security checks on the pointer.
2. The second dereference is for `memmove` to copy a buffer to the user-space buffer.

In the security patch, the TOCTOU vulnerability is removed by first copying the pointer from the second argument to the stack and then using the stack-allocated pointer for both `ProbeForWrite` and `memmove`. This eliminates the small time window between the two dereferences where the pointer may be modified to point to non-user-space memory.

To confirm our hypothesis, we decided to analyze the code paths that can trigger this functionality and see if this is indeed related to some kernel syscall interaction. `SkpMarshalCryptoParamsOut` only has one incoming call, which comes from `IumCrypto` . `IumCrypto` itself does not have any incoming calls. In the Windows Internals 7th Edition, Part 1 book [52], we found a description of `IumCrypto`, which is one of the 'isolated user-mode services' provided by the Secure Kernel. Windows Internals 7th Edition describes `IumCrypto` as a cryptographic service that allows Trustlets to perform cryptographic operations for which key material or randomness is generated and only

known by the Secure Kernel. Furthermore, `IumCrypto` functions as some sort of TPM for Trustlets to securely obtain information with the guarantee that the data was not tampered with [52].

As discussed in Section 2.3.1, Trustlets are user-mode processes executed in VTL1. Therefore, reaching `IumCrypto` through Trustlets already requires privileges within user-mode ring 3 of VTL1. However, there is another method to reach `IumCrypto` without the usage of Trustlet privileges.

Enclaves are software-based Trusted Execution Environments which can be used when Virtualization Based Security (VBS) is enabled. Third-party developers can use Windows APIs to load and execute an Enclave. An Enclave consists of two components [1]:

1. The Enclave Host App (runs in VTL0)
2. The Enclave DLL (runs in VTL1)

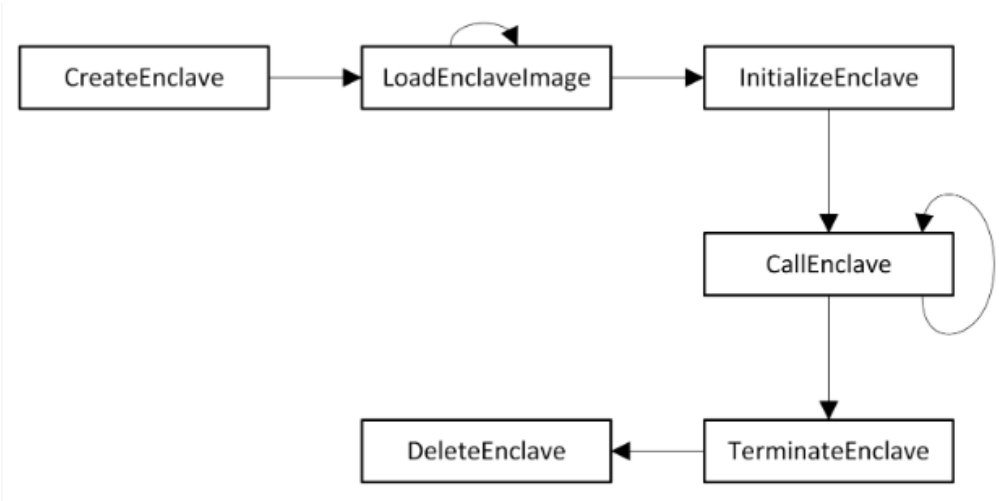


Figure 27: VBS enclave lifecycle from [14]

The Enclave is loaded and initialized in VTL1 and then available for VTL0 to be called. The Enclave Host App interacts with the Enclave DLL through the `CallEnclave` API call, which will cross the security boundary between VTL0 and VTL1. The lifecycle of an Enclave can be seen in Figure 27.

Research from Outflank has shown that `IumCrypto` is the only Secure System Call that can be called from an Enclave [2]. This is sufficient to trigger the vulnerable code path as seen in Figure 26. And since the Enclave API has been accessible to third-party developers since May 2024, we can follow the resources provided by Microsoft to create our own Enclave, similar to the custom kernel driver as described in Section 4.4.2.

5.2.1 Downgrading the Secure Kernel

In order to be able to trigger the vulnerable code path in the function `SkpMarshal CryptoParamsOut` of the Secure Kernel as described in Section 5.2, we need to have the vulnerable version of the Secure Kernel installed. Since Windows will automatically keep the debuggee VM up-to-date with the latest security patches, the Secure Kernel installed in our debuggee VM was not vulnerable anymore. Therefore, we need to downgrade the `securekernel.exe` binary in order to be able to trigger the vulnerable code path.

However, since the Secure Kernel is an OS-critical component that cannot be uninstalled and reinstalled like normal software, downgrading requires a different approach. This section describes our methodology used for downgrading the `securekernel.exe`.

Through the use of Winbindex [43], it is trivial to download the vulnerable `securekernel.exe` version. We use the `securekernel.exe` version for Windows 11 23H2 for x64-based Systems corresponding to knowledge base update KB5041585, which is the version just before the vulnerable code path was patched. It is not possible to overwrite the `securekernel.exe` on a booted Windows installation, since the `securekernel.exe` is actively used by the Windows operating system and therefore cannot be modified during execution.

Instead, it is necessary to boot into the recovery mode of Windows by holding shift when choosing 'restart' in the power-off menu. Next, through recovery mode, it is possible to access a `cmd.exe` session to interact with the filesystem of the unbooted Windows operating system. Here it is possible to copy the older version of `securekernel.exe` to `C:\Windows\System32\securekernel.exe` in order to downgrade the Secure Kernel. Afterwards, Windows can be booted as usual and the Secure Kernel is successfully downgraded.

This downgrading methodology is not recommended for other components within the Windows operating system. Since many components are only used internally by Windows and not exposed by some publicly accessible API, it is possible for certain code to change in a backwards-incompatible way. For example, it could have been the case that the syscall number mapping of internal functionality in the Secure Kernel has changed due to an update. Without downgrading other components in the Windows operating system, it is possible to end up with an incompatible state which will prevent booting the Windows operating system. However, since in Section 5.2 we analyzed the differences between the oldest and newest update, we were aware that such a situation cannot occur with downgrading our specific version of the Secure Kernel. Still, we have taken a snapshot of the debuggee VM before downgrading in case anything would break, which was not the case.

5.2.2 Creating a custom Enclave

This section describes the process of setting up a development environment for creating a custom Enclave. With a working setup for developing a custom Enclave, we can start interacting with the `IumCrypto` function from the Secure Kernel to trigger the vulnerable code path as described in Section 5.2. For the development setup for a custom Enclave, we make use of the development guide for Virtualization Based Security (VBS) Enclaves [14] as well as the example code for VBS Enclaves published by Microsoft [25]. We use the VMWare debuggee VM as the development machine for the custom Enclave, since we also want to execute the custom Enclave in the debuggee VM. The VMWare debuggee VM is used instead of the QEMU/KVM debuggee VM due to the better performance of VMWare, which is strongly recommended in order to develop the custom Enclave in Visual Studio. Note that in order for the VMWare debuggee VM to have a network connection to download the necessary components through Visual Studio Installer, the VMWare debugger VM must also be connected due to the network adapter being modified to expect a debugger attached.

As mentioned in the 'Development guide for Virtualization Based Security (VBS) Enclaves' [14], a development certificate is required in order to sign the Enclave DLL. The Enclave DLL must be signed in order to be loaded by the Secure Kernel. In order to create a development certificate, the following command can be executed in a non-admin

PowerShell prompt:

```
PS C:\Users\user> New-SelfSignedCertificate -
    CertStoreLocation Cert:\\CurrentUser\\My -DnsName "
    MyTestEnclaveCert" -KeyUsage DigitalSignature -KeySpec
    Signature -KeyLength 2048 -KeyAlgorithm RSA -
    HashAlgorithm SHA256 -TextExtension "2.5.29.37={text
    }1.3.6.1.5.5.7.3.3,1.3.6.1.4.1.311.76.57.1.15
     ,1.3.6.1.4.1.311.97.814040577.346743379.4783502.105532346"


PSParentPath: Microsoft.PowerShell.Security\Certificate::
    CurrentUser\My

Thumbprint                                Subject
----------                                -------
08FDF31EA47CA728731AD8550D71E6FB92F57BF2  CN=
    MyTestEnclaveCert
```

After building the Enclave DLL in Visual Studio using the example project provided by Microsoft [25], the Enclave DLL can be signed using the following command:

```
C:\Users\user\Documents\VbsEnclave\x64\Debug>signtool sign /
    ph /fd SHA256 /n "MyTestEnclaveCert" vbsenclave.dll
Done Adding Additional Store
Successfully signed: vbsenclave.dll
```

Note that in order to have access to `signtool`, a `cmd.exe` prompt should be opened through the 'x64 Native Tools Command Prompt for VS 2022', which can be found using the search bar in Windows. This step can also be automated in Visual Studio as a post-build event. This can be configured by going to the 'properties' for the 'Test Enclave' project, going to 'Configuration Properties' → 'Build Events' → 'Post-Build Event' and configuring the 'Command Line' value to the following value:

```
$(VEIID_Command) && signtool sign /ph /fd SHA256 /n "
    MyTestEnclaveCert" "$(OutDir)$(TargetName)$(TargetExt)"
```

This will sign the Enclave DLL each time the DLL is created using the build step.

The Enclave DLL can now be loaded by executing the `VbsEnclaveApp.exe` binary. However, doing so in our current setup gives an 'A device attached to the system is not functioning.' error, and the Enclave DLL is not successfully loaded. This error is due to test signing not being correctly configured on the debuggee VM. Even though our debuggee VM shows a 'Test Mode' watermark on the wallpaper in the bottom right corner due to the kernel debugger being attached, test signing is not configured. Test signing is required in order to allow the Secure Kernel to load an Enclave DLL that is not signed with a production signature through Azure Trusted Signing (which requires verification through Microsoft).

Test signing can be enabled by executing the following commands in an elevated `cmd.exe` prompt and rebooting the debuggee VM afterward.

```
C:\Windows\System32>bcdedit -debug on
The operation completed successfully.
```

```
C:\Windows\System32>bcdedit /set testsigning on
The operation completed successfully.
```

Using this setup, we can create an Enclave Host App as well as an Enclave DLL and load them both successfully.

5.2.3 Interacting with IumCrypto

As mentioned in Section 5.2, we want to interact with the `IumCrypto` API to execute the vulnerable code path. Using the Enclave development setup as described in Section 5.2.2, we can start interacting with the `IumCrypto` API implemented in ring 0 of the Secure Kernel. Interaction with the Secure Kernel API through an Enclave uses special Enclave functions implemented by Microsoft, which together form the API to interact with `IumCrypto` in the Secure Kernel [27]. By inserting a breakpoint at the start of the `IumCrypto` function implemented in the Secure Kernel, we verified that the eight Enclave API functions provided by Microsoft will execute the `IumCrypto` Secure Kernel function. Besides the `IumCrypto` API, Microsoft also provides a limited user-space API to be used within an Enclave [11]. Since only a small subset of the NT Kernel functionalities are also implemented in the Secure Kernel, the user-space API available for an Enclave in VTL1 is greatly limited.

Through experimentation with the Enclave API, we found that the functions `EnclaveSealData` and `EnclaveUnsealData` provided by Microsoft to interact with the `IumCrypto` functionality of the Secure Kernel through an Enclave will trigger the vulnerable code path in the function `SkpMarshalCryptoParamsOut`. More specifically, the Time-Of-Check Time-Of-Use (TOCTOU) vulnerability lies in the output address parameter of both `EnclaveSealData` and `EnclaveUnsealData`. The output address parameter is first verified to be within non-kernel memory, and then the `memmove` is performed (see also Figure 26). The problem here is that the output address is stored in the memory space of the Enclave DLL in VTL1 ring 3, and not in the Secure Kernel VTL1 ring 0 memory. This is because only a single pointer to a buffer stored in user-space of VTL1 containing all parameters for `IumCrypto` is passed to the `IumCrypto` function in kernel-space of VTL1, and the buffer is not copied from user-space to kernel-space before being used by the Secure Kernel. This allows the Enclave DLL to modify the output address in user-space after verification is done, bypassing the verification check. This method is visualized in Figure 28.

However, in order to exploit this behavior, we cannot make use of the `EnclaveSealData` and `EnclaveUnsealData` functions. This is because the output parameter for both functions is passed as a register value to the function, which prevents us from modifying the value after calling the function. Therefore, we have to find a way to directly interact with the `IumCrypto` Secure Kernel API instead of using the provided functions as part of the Enclave API.

Through online searching, we found that `EnclaveSealData` and `EnclaveUnsealData`, as well as the other Enclave API functions, are implemented in `vertdll.dll`. Upon inspecting the `vertdll.dll` binary, we found the methodology used by VTL1 ring 3 (user-space) to perform the syscall to VTL1 ring 0 (kernel-space). The pseudocode for the function `EnclaveSealData` can be found in Figure 29.

As can be seen in Figure 29, a buffer of size 120 is allocated on the stack, and then a call to `IumCrypto` is made. The `IumCrypto` function call here is a function that does a syscall to the Secure Kernel. This function has the exact same name as the kernel-space function `IumCrypto` which actually implements the IumCrypto logic. We call

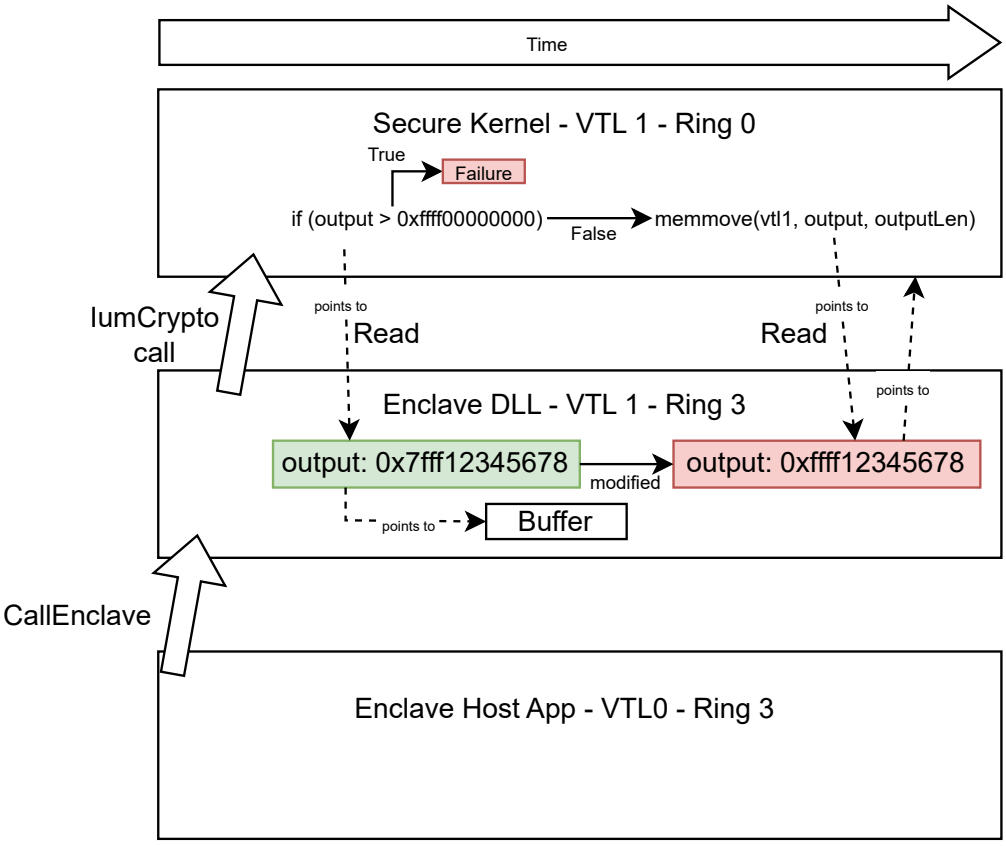


Figure 28: TOCTOU visualized with an Enclave Host App, Enclave DLL, and the Secure Kernel

the user-space `IumCrypto` function the '`IumCrypto` syscall function', which refers to the user-space function that performs the syscall to kernel-space of the Secure Kernel. The `IumCrypto` syscall function is executed in the context of our custom Enclave, so we can try to reimplement this code in our custom Enclave to have full control over the buffer that is used by `IumCrypto` in the Secure Kernel.

We made several attempts at reimplementing the user-space `IumCrypto` syscall functionality in our custom Enclave. We tried the following solutions to access the `IumCrypto` syscall function from within our custom Enclave:

1. Reimplementing the user-space `IumCrypto` syscall function in our custom Enclave to manually perform the syscall with our own buffer.
2. Loading the `iumdll.dll` binary, which also implements an `IumCrypto` user-space syscall function.
3. Resolving the address of the `IumCrypto` syscall function implemented in `vertdll.dll`.

For the first solution, we encountered the problem that the compiler used by Visual Studio does not support inline assembly when compiling to a 64-bit architecture. Therefore, it is not possible to write a function that correctly sets the registers for the syscall and executes the syscall instruction accordingly. A solution for this would be to change to a different compiler than the default used by Visual Studio, but we decided to try a

```
HRESULT __stdcall EnclaveSealData(
        const void *DataToEncrypt,
        UINT32 DataToEncryptSize,
        ENCLAVE_SEALING_IDENTITY_POLICY IdentityPolicy,
        UINT32 RuntimePolicy,
        PVOID ProtectedBlob,
        UINT32 BufferSize,
        UINT32 *ProtectedBlobSize)
{
  __int64 v8; // r14
  int v11; // eax
  __int64 v12; // rcx
  UINT32 v13; // eax
  char v15[120]; // [rsp+20h] [rbp-88h] BYREF

  v8 = DataToEncryptSize;
  if ( DataToEncrypt && ProtectedBlobSize )
  {
    memset_thunk_772440563353939046(v15, 0, 120uLL);
    *(_QWORD *)&v15[56] = DataToEncrypt;
    *(_QWORD *)&v15[64] = ProtectedBlob;
    *(_QWORD *)&v15[32] = BufferSize;
    *(_DWORD *)&v15[8] = IdentityPolicy;
    *(_DWORD *)&v15[12] = RuntimePolicy;
    *(_QWORD *)&v15[24] = v8;
    v11 = IumCrypto(v15);
    v12 = (unsigned int)v11;
    if ( v11 == -1073741789 )
    {
      if ( !ProtectedBlob )
        v12 = 0LL;
    }
    else if ( v11 < 0 )
    {
      return VertNtStatusToWin32Hresult(v12);
    }
    v13 = -1;
    if ( *(_QWORD *)&v15[88] > 0xFFFFFFFFuLL )
      v12 = 0xC0000095LL;
    else
      v13 = *(_DWORD *)&v15[88];
    *ProtectedBlobSize = v13;
    return VertNtStatusToWin32Hresult(v12);
  }
  return 0x80070057;
}
```

Figure 29: Pseudocode for the function `EnclaveSealData` as implemented in `vertdll.dll`

different approach before doing this since it could potentially break the custom Enclave development setup.

The second solution we tried was to import another DLL library into the custom Enclave

to call the IumCrypto syscall function implemented in another library. We found that the iumdll.dll library also implements the IumCrypto syscall function, equal to the implementation in vertdll.dll. However, this approach did not work since we were unable to load any extra libraries into the address space of VTL1 due to the limited user-space API available in our custom Enclave.

For the third solution, we know that the library vertdll.dll is already loaded into memory when our custom Enclave is executed and that vertdll.dll also implements the IumCrypto syscall function. One of the few functions available when developing a custom Enclave is the function GetModuleHandleExW, which allows a developer to receive the base address of a library already loaded into memory. Using GetModuleHandleExW, we were able to retrieve the base address of vertdll.dll. Furthermore, GetProcAddress is also available to use during the execution of our custom Enclave. However, in our specific vertdll.dll version for Windows 23H2 for x64-based Systems, the IumCrypto syscall function is not an exported symbol in vertdll.dll. Therefore, it is not possible to retrieve the address of the IumCrypto syscall function dynamically. Instead, we computed the offset of IumCrypto within the vertdll.dll binary and manually added this to the base address of vertdll.dll to know where the IumCrypto syscall function is located. Note that this causes our implementation to be non-portable since it depends on the exact vertdll.dll library version.

The last step required in order to directly interact with the IumCrypto API in the Secure Kernel without the use of the Microsoft-provided functions such as EnclaveSealData and EnclaveUnsealData is to reimplement the functions in our custom Enclave. By reimplementing the functions, we have direct control over the buffer used to store the parameters for the IumCrypto API, which is necessary to trigger the TOCTOU vulnerability. By inspecting the pseudocode of EnclaveSealData and EnclaveUnsealData, we have written our own implementation, which perfectly matches the buffer construction done in EnclaveSealData and EnclaveUnsealData. Instead of using a stack-based buffer, we instead use a globally allocated buffer so we can access the buffer in different functions in our custom Enclave. Using this reimplementation in our custom Enclave, we can successfully interact with the IumCrypto API in the Secure Kernel without using the functions provided by Microsoft.

5.2.4 Triggering the vulnerability in IumCrypto

In Section 5.2.3, we have developed a way to directly interact with the IumCrypto API in the Secure Kernel without using the functions of the Enclave API provided by Microsoft. This was necessary since, in order to trigger the vulnerability in IumCrypto, we have to have full control over the buffer used to store parameters for the IumCrypto API syscall. In our custom Enclave, we store the buffer as a global variable instead of a stack-based buffer so that multiple functions in our custom Enclave can access the buffer.

As mentioned in Section 5.2, the vulnerability is a Time-Of-Check Time-Of-Use (TOCTOU) vulnerability. In order to trigger the vulnerability, we need to have the following setup:

1. The buffer storing the parameters for the IumCrypto syscall is globally accessible.
2. A first thread is overwriting the output pointer in the global buffer to the correct value (the value is an address to another buffer globally allocated in our custom Enclave).
3. A second thread is overwriting the output pointer in the global buffer to a value pointing to kernel memory of the Secure Kernel (a value which should not be

allowed).

4. In the main thread, the custom Enclave repeatedly calls the `IumCrypto` API in the Secure Kernel with a pointer to the global buffer as the argument.

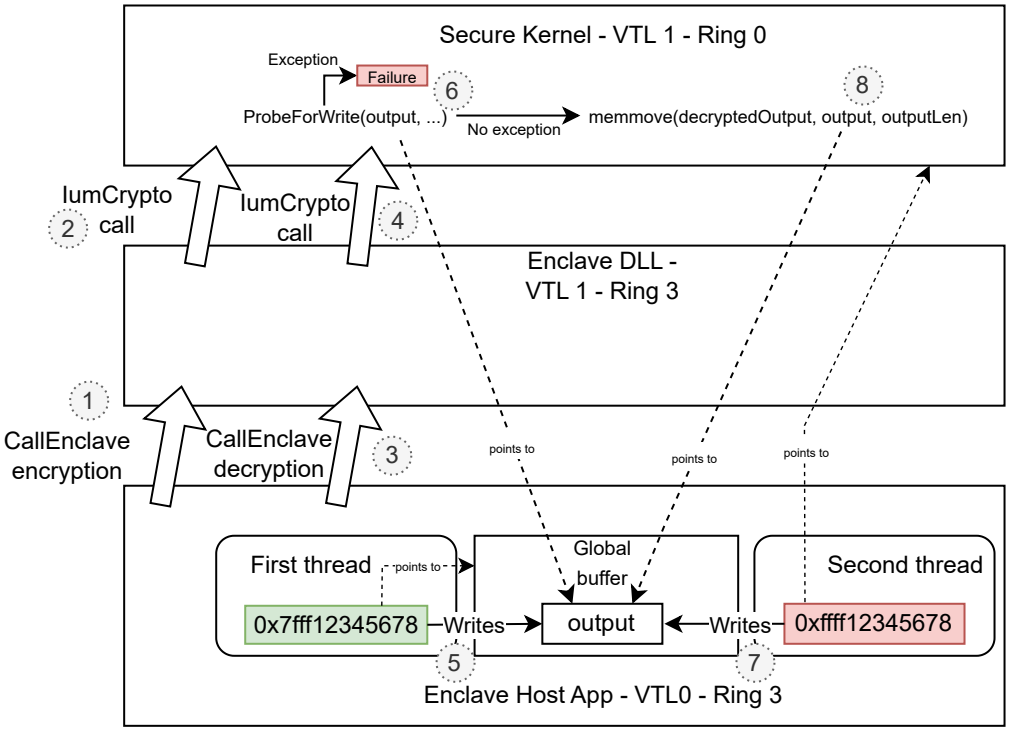


Figure 30: Overview of the exploit scenario for the Time-Of-Check Time-Of-Use (TOCTOU) vulnerability

A perfect exploit scenario will consist of the following steps in the following order:

1. The Enclave Host App will request an encryption of a payload to the Enclave DLL through `CallEnclave`.
2. The Enclave DLL will execute a `IumCrypto` syscall to encrypt the payload.
3. The Enclave Host App will request a decryption of the encrypted payload to the Enclave DLL through `CallEnclave`. The `CallEnclave` call will use a pointer to the global buffer as the only argument.
4. The Enclave DLL will execute a `IumCrypto` syscall to decrypt the payload, also using the pointer to the global buffer as the only argument for this syscall.
5. The first thread in the Enclave Host App will write a pointer to the global buffer in the Enclave Host App to the `output` field.
6. In the Secure Kernel ring 0, `ProbeForWrite` will check if the pointer stored in the `output` field is non-kernel memory and writable.
7. The second thread in the Enclave Host App will write a pointer pointing to kernel memory to the `output` field.
8. In the Secure Kernel ring 0, `memmove` will read the `output` field again (which now points to kernel memory) and will write the decrypted payload to kernel memory,

causing an arbitrary write primitive.

This exploit scenario is visualized in Figure 30. However, the scenario described here is the *perfect* exploit scenario, which assumes that the race condition happened with perfect timing. In reality, steps 1, 2, 3, 4, 6, and 8 are constantly repeated, and steps 5 and 7 are running as a separate loop as well. The exploit will only successfully execute when all the steps are followed in this exact order, but due to the nature of a TOCTOU, this is not guaranteed. We have developed proof-of-concept code[10] to trigger the TOCTOU. During our testing, we found that it requires between one thousand and ten thousand attempts to successfully trigger. This takes between one and thirty seconds for a single attempt. A successful attempt can be noticed by the Blue Screen of Death appearing with the 'SECURE KERNEL ERROR' stop code, which can be seen in Figure 31.

It is important to note that this vulnerability requires the usage of a multi-CPU and multi-threaded environment. Our debuggee VM virtualized by VMWare is allocated 4 vCPUs. Furthermore, while threading is supported in the context of an Enclave, threading in VTL0 is only used since both threads need to write to memory allocated in VTL0.

While writing the code for our custom Enclave to trigger the vulnerability, we noticed that a buffer encrypted by `EnclaveSealData` can only be decrypted once by `EnclaveUnsealData`. We have not seen this behavior documented by Microsoft. Therefore, in the main thread, we were required to first create a newly encrypted payload before triggering the vulnerability through the decryption functionality instead of only encrypting a payload once before repeatedly triggering the decryption functionality.

Since we have now successfully triggered the vulnerability in the `IumCrypto` API in the Secure Kernel, we can further research the exploitation of this vulnerability in Section 6.

5.3 Zero-day vulnerability in the Secure Kernel

In Section 5.2, we have identified a Time-Of-Check Time-Of-Use (TOCTOU) vulnerability in the Secure Kernel. This vulnerability is an n-day vulnerability and has already been patched by Microsoft. Due to this patch, we were able to rediscover the vulnerability using the strategy described in Section 3.

The vulnerability in Section 5.2 has the following pattern:

1. A pointer pointing to a pointer pointing to a user-space buffer is dereferenced, and the resulting pointer is used in `ProbeForWrite` to perform security checks on the user-space buffer.
2. The pointer pointing to a pointer pointing to a user-space buffer is dereferenced a second time and used in `memmove` to copy Secure Kernel memory to user-space.

We decided to audit the latest Secure Kernel binary (at the time of writing) for this specific vulnerable pattern. By checking every call to `ProbeForWrite`, we found exactly one occurrence of this specific vulnerable pattern. We found this vulnerable pattern in the function `SkpspTlsReplaceVector` in the Secure Kernel binary. The pseudocode of `SkpspTlsReplaceVector` can be found in Figure 32.

The call to `ProbeForWrite` is marked yellow in Figure 32, and afterward, a call to `memmove` is made. In the call to `ProbeForWrite`, the pointer at `a2 + 24 * v5 + 24` is dereferenced and used for security checks done by `ProbeForWrite`. Next, the pointer at

[10] `https://github.com/JJ-8/securekernel-n-day-enclave-poc`

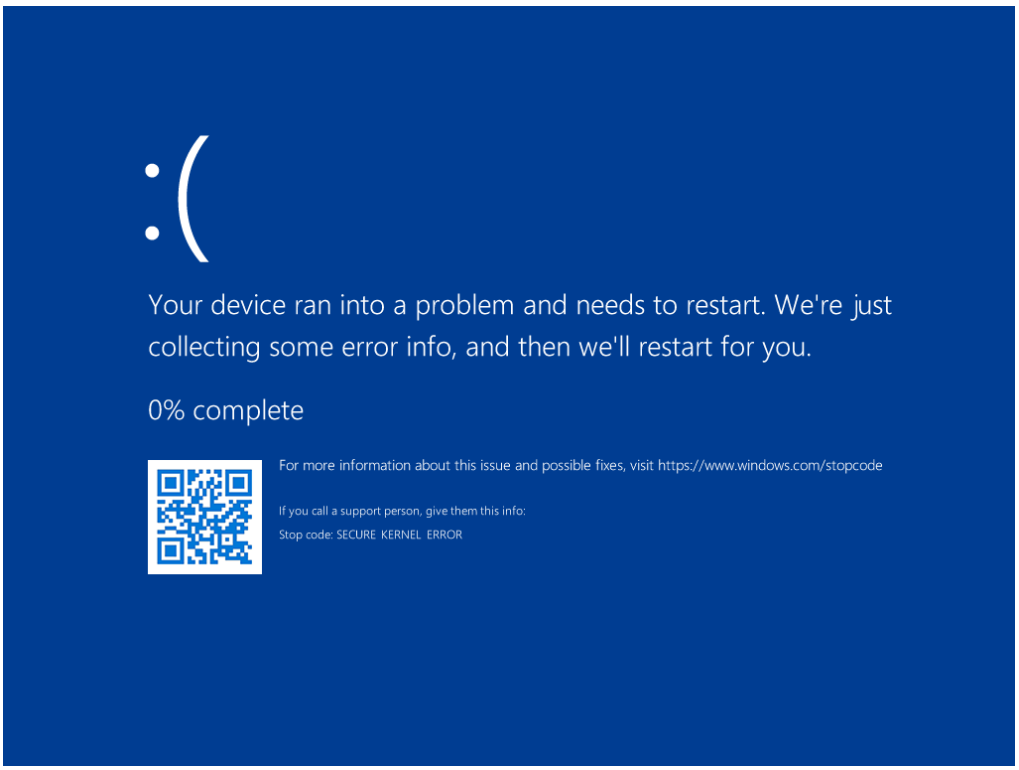


Figure 31: Blue Screen of Death after triggering the Time-Of-Check Time-Of-Use (TOCTOU) vulnerability

`a2 + 8 * v6 + 24` is dereferenced and used as the destination for `memmove`. Note that `v6 = 3 * v5`, which is computed at the start of the function, so the pointer to `memmove` is `a2 + 8 * (3 * v5) + 24 = a2 + 24 * v5 + 24`, which matches the pointer to `ProbeForWrite`. We do not know why this alternative computation is not used in the call to `ProbeForWrite` since it makes the dereferencing inconsistent.

There is exactly one code path to trigger this vulnerable functionality. `SkpspTlsReplaceVector` is only called from the function `SkpspSetTlsInformation`. The pseudocode for this function can be found in Figure 33. The function `SkpspSetTlsInformation` itself is called again from only one place, which is the function `NtSetInformationProcess`. The pseudocode for this function can be found in Figure 34. The function `NtSetInformationProcess` does not have any incoming calls within the Secure Kernel binary.

Research from Outflank has shown that `NtSetInformationProcess` is part of syscall 28 in the Secure Kernel [2]. This syscall is only accessible for user-mode applications running in VTL1, which are called Trustlets (see Section 2.3.1). Therefore, this syscall is not accessible from an Enclave. We have verified that this indeed is not possible by patching our custom Enclave DLL to include a method to call syscall 28, and inspecting the return code which indeed contains an error code. Since Trustlets cannot be developed by third parties at the time of writing, the attack surface is very limited. We think that the reason why this vulnerability has not been patched yet, is that it is not available to (malicious) third parties, and Microsoft Security Response Center requires you to have a proof-of-concept exploit (with a video showcasing the exploit) before they will look into the vulnerability. This is also the reason why we have decided to not further look into this vulnerability.

```
__int64 __fastcall SkpspTlsReplaceVector(__int64 a1, __int64 a2, __int64 a3, _DWORD *a4)
{
  __int64 v5; // r10
  __int64 v6; // r12
  __int64 v7; // r14
  _QWORD *v8; // r15
  _QWORD *v9; // rdi
  unsigned int v10; // ecx
  unsigned __int64 v11; // rdx
  unsigned int v12; // ebx
  _DWORD *v15; // [rsp+88h] [rbp+20h]

  v15 = a4;
  v5 = (unsigned int)*a4;
  v6 = 3 * v5;
  v7 = a3 + 16 + 24 * v5;
  v8 = (_QWORD *)(a1 + 88);
  v9 = *(_QWORD **)(a1 + 88);
  if ( v9 )
  {
    if ( v9 == v8 )
    {
      v9 = 0LL;
    }
    else
    {
      v10 = 8 * *(_DWORD *)(a2 + 12);
      if ( v10 )
      {
        if ( ((unsigned __int8)v9 & 7) != 0 )
          ExRaiseDatatypeMisalignment();
        v11 = (unsigned __int64)&v9[v10 / 8];
        if ( v11 > 0x7FFFFFF0000LL || v11 < (unsigned __int64)v9 )
          MEMORY[0x7FFFFFF0000] = 0;
      }
      v12 = v10;
      ProbeForWrite(*(volatile void **)(a2 + 24 * v5 + 24), v10, 8u);
      memmove(*(void **)(a2 + 8 * v6 + 24), v9, v12);
      a4 = v15;
      LODWORD(v5) = *v15;
    }
    *(_DWORD *)v7 |= 1u;
    *v8 = *(_QWORD *)(a2 + 8 * v6 + 24);
    *(_QWORD *)(v7 + 16) = *(_QWORD *)(a1 + 72);
    *(_QWORD *)(v7 + 8) = v9;
    *(_DWORD *)v7 ^= 3u;
    *a4 = v5 + 1;
  }
  return 0LL;
}
```

Figure 32: Vulnerable code pattern in `SkpspTlsReplaceVector` in the Secure Kernel

```
__int64 __fastcall SkpspSetTlsInformation(void *a1, size_t a2)
{
  int v3; // esi
  _QWORD *StackBase; // rcx
  __int64 v5; // r14
  __int64 NextProcessThread; // rax
  _QWORD *v7; // rbx
  __int64 v8; // rdi
  int v9; // eax
  int v10; // eax
  unsigned int v12; // [rsp+20h] [rbp-40h] BYREF
  _QWORD *v13; // [rsp+28h] [rbp-38h]
  _QWORD v14[2]; // [rsp+30h] [rbp-30h] BYREF
  __int64 v15; // [rsp+50h] [rbp-10h]

  v15 = 0LL;
  v13 = v14;
  memset(v14, 0, sizeof(v14));
  v3 = SkpspCaptureAndValidateTlsInfo(a1, a2);
  if ( v3 >= 0 )
  {
    StackBase = KeGetPcr()->NtTib.StackBase;
    v12 = 0;
    v5 = StackBase[9];
    NextProcessThread = SkeGetNextProcessThread(v5, 0LL);
    v7 = v13;
    while ( 1 )
    {
      v8 = NextProcessThread;
      if ( !NextProcessThread )
        break;
      v9 = *((_DWORD *)v7 + 1);
      if ( v9 )
      {
        if ( v9 != 1 )
        {
          v3 = -1073741595;
LABEL_9:
          if ( v8 )
            SkReleaseRundownProtection((volatile signed __int64 *)(v8 + 208));
          break;
        }
        v10 = SkpspTlsReplaceVector(*(_QWORD *)(v8 + 152), v7, a1, &v12);
      }
      else
      {
        v10 = SkpspTlsReplaceIndex(*(_QWORD *)(v8 + 152), v7, a1, &v12);
      }
      v3 = v10;
      if ( v10 < 0 || v12 >= *((_DWORD *)v7 + 2) )
        goto LABEL_9;
      NextProcessThread = SkeGetNextProcessThread(v5, v8);
    }
    if ( v14 != v7 )
      SkFreePool(1LL, (signed __int64 *)v7);
0003CCE4 SkpspSetTlsInformation:43 (14003D6E4)
```

Figure 33: Pseudocode of the function `SkpspSetTlsInformation` calling the function `SkpspTlsReplaceVector` in the Secure Kernel

```
NTSTATUS __fastcall NtSetInformationProcess(__int64 a1, int a2, unsigned __int64 a3, unsigned int a4)
{
  unsigned __int64 v4; // rbx
  NTSTATUS result; // eax
  __m256i v6; // [rsp+38h] [rbp-30h] BYREF
  __int64 v7; // [rsp+58h] [rbp-10h]

  v4 = a3;
  result = -1073741637;
  if ( a1 == -1 )
  {
    switch ( a2 )
    {
      case 35:
        return SkpspSetTlsInformation(a3, a4);
      case 12:
        return NkSetInformationProcess(-1LL, 12LL);
      case 41:
        memset(&v6, 0, sizeof(v6));
        if ( a4 == 40 )
        {
          v7 = 0LL;
          if ( *((_BYTE *)KeGetPcr()->NtTib.StackBase + 88) == 1 )
          {
            if ( (a3 & 7) != 0 )
              ExRaiseDatatypeMisalignment();
            if ( a3 + 40 > 0x7FFFFFF0000LL || a3 + 40 < a3 )
              MEMORY[0x7FFFFFF0000] = 0;
            v6 = *(__m256i *)a3;
            v7 = *(_QWORD *)(a3 + 32);
            a3 = (unsigned __int64)&v6;
          }
          *(_QWORD *)(a3 + 32) = 0LL;
          result = ZwAllocateVirtualMemory(
                     (HANDLE)0xFFFFFFFFFFFFFFFFLL,
                     (PVOID *)(a3 + 32),
                     *(_QWORD *)(a3 + 24),
                     (PSIZE_T)(a3 + 16),
                     0x2000u,
                     4u);
          if ( result >= 0 && *((_BYTE *)KeGetPcr()->NtTib.StackBase + 88) == 1 )
            *(_QWORD *)(v4 + 32) = v7;
        }
        else
        {
          return -1073741820;
        }
        break;
    }
  }
  return result;
}

0003CC3A NtSetInformationProcess:15 (14003D63A)
```

Figure 34: Pseudocode of the function `NtSetInformationProcess` calling the function `SkpspSetTlsInformation` in the Secure Kernel

6 Exploitation

In this section, we discuss the exploitation phase for the vulnerability we found in Section 5.2. At the end of Section 5.2.4, we have successfully developed a proof-of-concept for an arbitrary write primitive within the Secure Kernel triggered by a custom Enclave. This section will first discuss in Section 6.1 the security mitigations enabled in the Secure Kernel. Section 6.2 will then discuss an exploitation strategy for the Secure Kernel. In Section 6.3 we will then discuss our attempt at developing an exploit using the proof-of-concept we have written for the arbitrary write primitive as discussed in Section 5.2.

6.1 Secure Kernel security mitigations

There is very limited information available online about exploiting the Secure Kernel. The only source that is publicly available online is a Black Hat USA 2020 talk by Saar Amar and Daniel King (researchers at Microsoft Security Response Center) about attacking and hardening the Secure Kernel [60]. A recording has been published on YouTube [61] where the details of the exploitation technique are presented.

In this talk, a Secure Kernel exploitation technique is documented for an arbitrary write primitive. By fuzzing and manual code analysis, they found several bugs in the Secure Kernel which resulted in an arbitrary write primitive. Before we investigate how they used the arbitrary write primitive to exploit the Secure Kernel, some security mitigations in the Secure Kernel should be discussed.

As discussed in the Black Hat talk [60], the Secure Kernel lacks several security mitigations:

1. Second Level Address Translation (SLAT) enforcement
2. Control flow integrity mechanism
3. Partial Kernel Address Space Layout Randomization (KASLR)

We will briefly discuss each security mechanism and its consequences.

6.1.1 Second Level Address Translation enforcement

As discussed in Section 2.1.3.2, Second Level Address Translation (SLAT) is used to introduce an extra translation layer between the physical memory of the machine and the memory as seen by the Secure Kernel or NT Kernel. When SLAT is enabled, a higher Virtual Trust Level (VTL) can enforce restrictions on the memory of lower-level VTLs. This means that the Secure Kernel can enforce memory protections such as marking certain pages as read-only. Even when the NT Kernel running in VTL0 sets the memory protections to writable and writes data to it, this violates the memory protections set by the Secure Kernel in VTL1 and therefore refuses the write operation.

However, SLAT enforcement is not possible for the Secure Kernel running in VTL1. Since there is no higher VTL available that will enforce memory protections on the memory in VTL1, the memory protections are not guaranteed in VTL1. This means that an exploit running in VTL1 ring 0 will be able to change memory permissions by modifying the page tables in VTL1.

There is no solution for SLAT enforcement in the Secure Kernel, except introducing another kernel on top of the Secure Kernel that will enforce memory protections on VTL1. Therefore, this makes abusing the lack of SLAT enforcement a strong and stable attack primitive for attackers.

6.1.2 Control Flow Integrity Mechanism

The Secure Kernel does not implement any form of control flow integrity mechanism. Control flow integrity is implemented in the NT Kernel and protects the NT Kernel against attacks that take advantage of redirecting code execution to unintended locations. For example, return-oriented programming (ROP) is an attack method that is prevented by control flow integrity. With ROP, the return of a function will point to the code epilogue of another function, but with control flow integrity it is determined at compile time which functions are allowed to return to which locations in memory. This information at compile time is then used at runtime to decide if a jump in code is allowed to be taken. This prevents any unauthorized jumps in code, for example by overflowing a buffer stored on the stack, which allows an attacker to overwrite return pointers and thus change code execution of a program.

Due to the lack of control flow integrity in the Secure Kernel, it is more trivial for an attacker to exploit vulnerabilities in the Secure Kernel. For example, writing a fake stack with a ROP-chain in a writable segment of the Secure Kernel and changing the stack pointer to the start of the ROP-chain allows an attacker to trivially determine code execution in the Secure Kernel. Such attacks are much more restricted in the NT Kernel due to control flow integrity.

We do not know the exact reason why no control flow integrity has been implemented in the Secure Kernel. It could be a similar reason as why Second Level Address Translation (SLAT) enforcement is not possible: there is no way to prevent unauthorized modifications to Secure Kernel memory. Control flow integrity is implemented using a bit-map which is checked at runtime to determine if a jump is allowed to be taken. In the NT Kernel, the Secure Kernel can guarantee that this bit-map is not modified at runtime. Such protection is not possible in the Secure Kernel due to the lack of SLAT enforcement. This means that an attacker can change a read-only bit-map memory page to writable and change the bit-map to bypass control flow integrity.

6.1.3 Partial Kernel Address Space Layout Randomization

The Secure Kernel implements partial Kernel Address Space Layout Randomization (KASLR). KASLR is an effective defense against kernel exploitation, since with KASLR fully enabled, it is more difficult for an attacker to use certain primitives. For example, the arbitrary write primitive would be impossible to use unless the attacker manages to leak a Secure Kernel address, which allows the attacker to compute the randomized offset. However, with a partial overwrite of a Secure Kernel address, it would be possible to bypass the KASLR defense since only the higher bits of an address are randomized (which will not be overwritten with a partial overflow). If KASLR is not used or if only partial KASLR is used, this would allow an attacker to know where certain memory pages are located in memory. Using this, an attacker does not need to have a kernel address leak in order to know where to write shellcode or a ROP-chain in memory.

Saar Amar and Daniel King discussed KASLR in the Secure Kernel in the Black Hat talk [60]. They reported that the Secure Kernel implements only partial KASLR since some addresses are hardcoded and some addresses are predictable. The reason that some addresses are predictable is that the boot sequence of the Secure Kernel is deterministic and therefore predictable per version of the Windows operating system. Therefore, some addresses are not properly randomized due to the predictable boot sequence. In Table 7, an overview can be found of the hardcoded and deterministic addresses they reported in their presentation. Furthermore, they reported that some memory pages are shared between VTL0 and VTL1. Note that these shared memory pages are only available in

VTL0 from the context of the NT Kernel. For example, there is a memory page in VTL1 mapped at 0xfffff78000000000 and the same memory page is mapped in VTL0 at 0xfffff78000007000. Another example is the `SkmiFailureLog`, which is used as a logging buffer where the Secure Kernel can log failures that can be read by the NT Kernel.

Type	Name	Address
Hardcoded	PTE_BASE	0xfffff6c800000000
Hardcoded	Pfndb	0xffffe80000000000
Hardcoded	SkmiSystemPTEs Base	0xfffff6c800000000
Hardcoded	SkmiImagePTEs Base	0xfffff6cc80000000
Hardcoded	SkmiIoPTEs Base	0xfffff6fffff80000
Hardcoded	Paged Pool	0xffff9a0000000000
Hardcoded	shared page VTL1	0xfffff78000000000
Hardcoded	shared page VTL0 mapping	0xfffff78000007000
Deterministic	SkpgContext	0xffff9880419b6000
Deterministic	SkmiFailureLog	0xffff988000000000

Table 7: "KASLR - Predictable Addresses" from [60]

Using our debugging setup from Section 4.2.5, we tried to inspect the memory located at these addresses. However, we were unable to inspect any of these addresses in the memory space of the Secure Kernel using GDB. We tried to reproduce this by first setting a hardware breakpoint at the start of the function `IumInvokeSecureService` in the Secure Kernel, and as soon as the breakpoint was reached, we tried to inspect the memory. We think that the reason we were unable to inspect any of these addresses is that Microsoft has hardened the Secure Kernel in the last five years, which makes the information from the Black Hat talk from 2020 outdated.

6.2 Secure Kernel Exploitation techniques using Secure Kernel Patch Guard

In the Black Hat talk [60] [61], Saar Amar and Daniel King discuss an exploit technique for the Secure Kernel that can be used in combination with an arbitrary write primitive. In this section, we will discuss this Secure Kernel exploit technique and how we can use it to achieve code execution for the vulnerability we reproduced in Section 5.2.

As mentioned in Section 2.3.3, Secure Kernel Patch Guard (SKPG) uses a context structure stored in Secure Kernel memory to keep track of information related to the verification procedure of SKPG. This SKPG context structure contains several function pointers used for calling functions during the verification process. While the SKPG also verifies the integrity of the SKPG context structure during runtime, two pointer fields are excluded from verification: `TimerRoutine` and `RuntimeCheckRoutine`. We think that the reason these two pointers are excluded from verification is that these two pointers themselves are used to perform the verification, and somehow it is not possible to include them in the verification procedure.

By taking advantage of the lack of verification of these two pointers in the SKPG context structure, it is possible to achieve arbitrary code execution by overwriting the `RuntimeCheckRoutine` pointer with a different value. Based on the configured timeout for the `TimerRoutine`, the `RuntimeCheckRoutine` will be called regularly (in our testing, it took several minutes before `RuntimeCheckRoutine` was called). This means that by using an arbitrary write to overwrite the value of `RuntimeCheckRoutine` and waiting several minutes, code execution in the context of the Secure Kernel can be achieved.

However, this code execution is very limited since we only have control over the RIP register. This is because the code execution is triggered from a completely different context that we do not control at all. Therefore, triggering a ROP chain to exploit the Secure Kernel will not be possible. Instead, Saar Amar and Daniel King demonstrate a setup that can be used in combination with shellcode to achieve controlled code execution in the Secure Kernel.

The exploit strategy of Saar Amar and Daniel King takes advantage of shared memory pages between the Secure Kernel and the NT Kernel. They use a custom kernel driver in the NT Kernel to write shellcode to a (for the NT Kernel) writable shared page with a fixed address. This shared page is read-only for the Secure Kernel, however. In order to make this shared page executable in VTL1 so the shellcode can be successfully executed by the Secure Kernel, they modify the page table entries in VTL1. As discussed in Section 6.1.1, due to the lack of Second Level Address Translation (SLAT) enforcement, this is possible in VTL1. To summarize, the following steps are used for this exploitation strategy:

1. Write shellcode to a shared page between NT Kernel and Secure Kernel with a fixed address using a custom kernel driver.
2. Corrupt the page table entry of the shared page in VTL1 to make it executable.
3. Overwrite the value of `RuntimeCheckRoutine` in the `SkpgContext` to point to the shellcode.
4. Wait until the shellcode is executed.

6.3 Exploiting the Secure Kernel with the arbitrary write primitive

In Section 6.2, we discuss the exploitation technique described by Saar Amar and Daniel King in their Black Hat talk [60] [61]. It is important to note that this talk was given in 2020, which is five years ago at the time of writing, which affects the reproducibility of their research. The goal of this section is to try to reproduce their research and explore what is possible within our debugging setup using the arbitrary write primitive we found in Section 5.2.

6.3.1 Defeating Kernel Address Space Layout Randomization (KASLR)

The exploitation technique by Saar Amar and Daniel King relies on the lack of full Kernel Address Space Layout Randomization (KASLR) in the Secure Kernel. As can be seen in Table 7, they report that many addresses have a fixed location in memory. However, this is no longer the case for the Secure Kernel in recent versions of Windows. We tried inspecting each of these addresses, but we were not able to inspect any of the memory pages at these fixed addresses in the Secure Kernel using GDB. This suggests that Microsoft has improved the hardening of the Secure Kernel since 2020, making the exploitation technique described by Saar Amar and Daniel King no longer directly applicable in our environment. In this section we discuss our findings that let to this conclusion.

Due to our earlier research in Section 5.1, we were already aware of the symbol `SkmiFailureLog`, which has a 'deterministic' address according to Table 7. `SkmiFailureLog` is a shared page between the Secure Kernel and the NT Kernel with the purpose of communicating failures between the NT Kernel and the Secure Kernel. The memory page of `SkmiFailureLog` is writable by the NT Kernel and writable by the Secure Kernel.

There is a symbol `SkmiFailureLog` in the `securekernel.exe` binary that stores the pointer to the corresponding memory page. When reading this pointer, we found that the memory page is slightly randomized per boot. For example, `0xffffe00000000000` and `0xffff9a0000000000` are Secure Kernel addresses where the `SkmiFailureLog` memory page is located.

The address of `SkpgContext` is randomized in a similar way as `SkmiFailureLog`, which also makes it not possible to reliably know without leaking the address. Since Saar Amar and Daniel King report that both these addresses are 'deterministic' and not fully randomized, it may be the case that knowing one of these two addresses allows one to determine the other address. However, we were not able to verify this.

For all other 'hardcoded' addresses from Table 7, we were unable to inspect these addresses. The 'hardcoded' addresses can be found in `securekernel.exe` by their symbols `SkmiSystemPtes`, `SkmiIoPtes`, and `SkmiImagePtes`. Furthermore, we found that there is also a similar symbol introduced which has not been documented in Table 7: `SkmiNonPagedPtes`. Inspecting the values at these symbols reveals the pointer to the memory pages for each of the Page Table Entry. By inspecting these values again after a reboot of the debuggee VM, we indeed confirmed that these addresses are randomized after each boot. The amount of randomized bytes is greater than the deterministic address of `SkmiFailureLog`. `SkmiFailureLog` has two bytes of randomness, while the other addresses have four to nine bytes of randomness.

From this, we can conclude that it is not possible to exploit the Secure Kernel with only an arbitrary write primitive without an arbitrary read primitive or a leak of a Secure Kernel address. This makes the arbitrary write primitive from Section 5.2 insufficient for exploitation. However, we can still reason about the theoretical situation in which we have a Secure Kernel address leak or an arbitrary read primitive, since we can simulate this using the GDB debugger. For example, we can target the `SkmiFailureLog` memory page, which can store shellcode that can be written by a custom kernel driver and then is available in Secure Kernel memory. We will discuss this theoretical situation in Section 6.3.2.

6.3.2 Making a shared memory page executable

Suppose we have an arbitrary read primitive in the Secure Kernel such that we can defeat Kernel Address Space Layout Randomization (KASLR) in the Secure Kernel, we can continue the exploitation strategy of Saar Amal and Daniel King. We can simulate the arbitrary read primitive through GDB in our debugging setup. In this theoretical scenario, we use `SkmiFailureLog` as the shared memory page where we will write the shellcode, since this memory page is both writable from the NT Kernel as well as readable by the Secure Kernel.

Since `SkmiFailureLog` is not an executable memory page, we have to make it executable through our arbitrary write primitive. This is possible due to the lack of Second Level Address Translation (SLAT) enforcement (see Section 6.1.1). Therefore, we have to update the Page Table Entry (PTE) of the `SkmiFailureLog`, which stores the control bits that determine if a page is readable, writable, and/or executable. For this, we have to locate the Page Table (PT) in the Secure Kernel. In our VMWare debugging setup, we can use the command `monitor phys` to switch to physical memory mode. With the command `monitor r cr3`, we can read the value of the cr3 register. The value of the cr3 register points to the base of the PT in physical memory. However, we cannot write to physical memory with our arbitrary write primitive, since writing to physical memory is restricted by the CPU mode during execution of the Secure Kernel. It is common

that there is also a virtual address space mapping of the PT in physical memory, which is used by the Secure Kernel to manage the PTE. The goal is to use this virtual address space mapping of the PT to update the PTE of the memory page of `SkmiFailureLog` to be executable using our arbitrary write primitive.

We have made extensive reverse engineering efforts to locate the PT in the virtual address space, but we were unable to find the PT in the virtual address space. As shown in Table 7, there are several PTs in the Secure Kernel, each with a different purpose: 'System', 'Image', 'IO', and we also found a 'NonPaged' PT. However, none of these PTs have the same memory content as the PT in physical memory. Therefore, we are unable to update the PTE of `SkmiFailureLog` and thus cannot execute the shellcode. Due to time constraints, we were unable to further investigate this issue.

7 Future Work

The following subsections outline the various research ideas that were developed during the course of this thesis. These ideas aim to provide new directions for future research topics that are related to the Secure Kernel. The ideas listed in this chapter have no particular order and are independent research topics. In Section 7.1, we discuss the possibility of using memory snapshot-based fuzzing for `IumInvokeSecureService` to discover new bugs. Section 7.2 discusses some improvements that can be made to the Secure Kernel debugging setup, which may help in simplifying the setup. Furthermore, Section 7.3 is also related to the research setup, which discusses an idea to simplify NT Kernel interaction by using Python bindings. In Section 7.4, we make new suggestions for a potential new attack surface that can be researched to discover new zero-days in the Secure Kernel. Section 7.5 discusses the idea of also researching Secure Kernel bugs with public exploits available, which may help in gaining new knowledge about the attack surface of the Secure Kernel.

7.1 Memory snapshot-based fuzzing on IumInvokeSecureService

As mentioned in Section 2.3.2, the Hypercalls are handled by `IumInvokeSecureService` in the `securekernel.exe` binary. These Hypercalls are similar to syscalls to the normal NT Kernel. There are already various efforts performed in fuzzing the Hypercall parsing of `IumInvokeSecureService`. There are two methods used in fuzzing Hypercalls:

1. Performing raw VMCALL instructions from the Windows Kernel to the Secure Kernel using a kernel driver (see Section 2.3.2).
2. Using Microsoft's recommended interface from winhv.sys [41].

Interacting with the Secure Kernel is non-trivial to perform, and the recommended interface from Microsoft does not implement all Hypercalls. Therefore, we propose a third way for performing fuzzing: memory snapshot based fuzzing. Instead of manually interacting with the Secure Kernel from the Windows Kernel, we only use the debugging setup described in Section 4.2 to attach to the Secure Kernel. Using this debugger, we can set a breakpoint at the start of the handling of the Hypercalls. When this breakpoint is triggered, we can dump the values of the CPU registers as well as the whole memory address space of the Secure Kernel. A memory snapshot based fuzzer like 'what the fuzz' [68] can be used or extended to support loading the dumped CPU register values and memory address space. After correctly loading this into the fuzzer, a small harness needs to be written in order to mutate the current parameters of the Hypercall. Using this harness, it is possible to perform fuzzing of the `IumInvokeSecureService` function without the need for a kernel driver. See [3] for information about how to create a memory dump for the Secure Kernel.

The advantage of this methodology of fuzzing is that it is possible to take advantage of valid Hypercalls and mutate their values to trigger error flows and potentially hit security bugs. Furthermore, the breakpoint can be moved from the start of `IumInvokeSecureService` to handling a specific Hypercalls which makes fuzzing more targeted compared to fuzzing all Hypercalls. When targeting a Hypercall for fuzzing, it once again already has a Hypercall with valid Hypercall parameters which can be mutated to perform fuzzing. This may also result in a faster development time of implementing a fuzzer, since no time needs to be spent on determining correct values to use in the Hypercall. Furthermore, since we only execute the Hypercall parsing code, it may also result in faster fuzzing iterations compared to fuzzing through the NT Ker-

nel which requires Virtual Trust Level (VTL) switches for every fuzzing iteration (see Section 2.2.2).

7.2 Scanning for securekernel.exe in VTL1

The method described in Section 4.2.1 was not successful since we required access to physical memory which was not provided by the two WinDBG sessions. However, in theory, the Secure Kernel should be loaded somewhere in VTL1 which is accessible to the hypervisor. In practice, it is unknown at which address the Secure Kernel is loaded in the WinDBG session of the hypervisor, and therefore we cannot debug the Secure Kernel through WinDBG. In order to solve this problem, the QEMU/KVM virtualized setup has been introduced in Section 4.2.2.

As potential future work, research can be done to discover the Secure Kernel binary in the memory of the hypervisor. This would allow Secure Kernel debugging without the use of the virtualized setup. One of the possibilities that can be researched is scanning the address space around the `hvix64.exe` binary. This idea has already been performed to recover the Secure Kernel physical base address [49], but it may also work for the virtual address. Another possibility is to intercept execution during a transition from VTL1 to VTL0 and start scanning page aligned backwards from the return. A practical implementation of this methodology has already been shown to be possible [73]. This implementation requires using a bootkit to hijack the execution flow of the boot loader in order to run custom code before the operating system or hypervisor is loaded. This setup is similar to the Hyper-V Backdoor project developed by Dmytro Oleksiuk, which also uses the backdoor to retrieve the virtual base address of the Secure Kernel [50].

As an alternative to this idea, it is also possible to research how LiveCloudKd manages to discover the Secure Kernel virtual base address. As described in Section 4.3.1, LiveCloudKd is a Windows-based setup that uses Hyper-V as the virtualization backend to allow Secure Kernel debugging. LiveCloudKd itself is open source, but searching through the source code does not reveal any code related to discovering the Secure Kernel virtual base address. Since the Secure Kernel base address message is prepended with `hvlib:`, we did have a look into the `hvlib.dll` binary. In Figure 35 the decompilation can be found of the logic used in `hvlib.dll`[11] to retrieve the Secure Kernel virtual base address. Since there are no symbols available, and no open source project exists for hvlib, it is unknown what exactly is done to retrieve the Secure Kernel virtual base address. However, it will probably be related to interacting with Hyper-V APIs to read specific values from memory. As can be seen in Figure 35, some page scanning logic is executed to discover the correct Secure Kernel page. Potentially, this logic can be re-implemented without dependencies on Hyper-V which allows for cross-platform usage. For example, a WinDBG plugin that uses the hypervisor debugging session to interact with memory and compute the Secure Kernel virtual base address in a certain way.

7.3 Python bindings for the custom kernel driver to interact with the Secure Kernel

In Section 4.4.2.2 we have developed a custom kernel driver to interact with the Secure Kernel. Interaction with the custom kernel driver can be done using a user-space client binary that opens the kernel driver and communicates with it through Device Input and Output Control. This user-space client binary is implemented in C. Therefore,

[11] hvlib.dll is bundled in `https://github.com/gerhart01/LiveCloudKd/releases/download/v1.0.22021109/LiveCloudKd.EXDi.debugger.v1.0.22021109.zip`

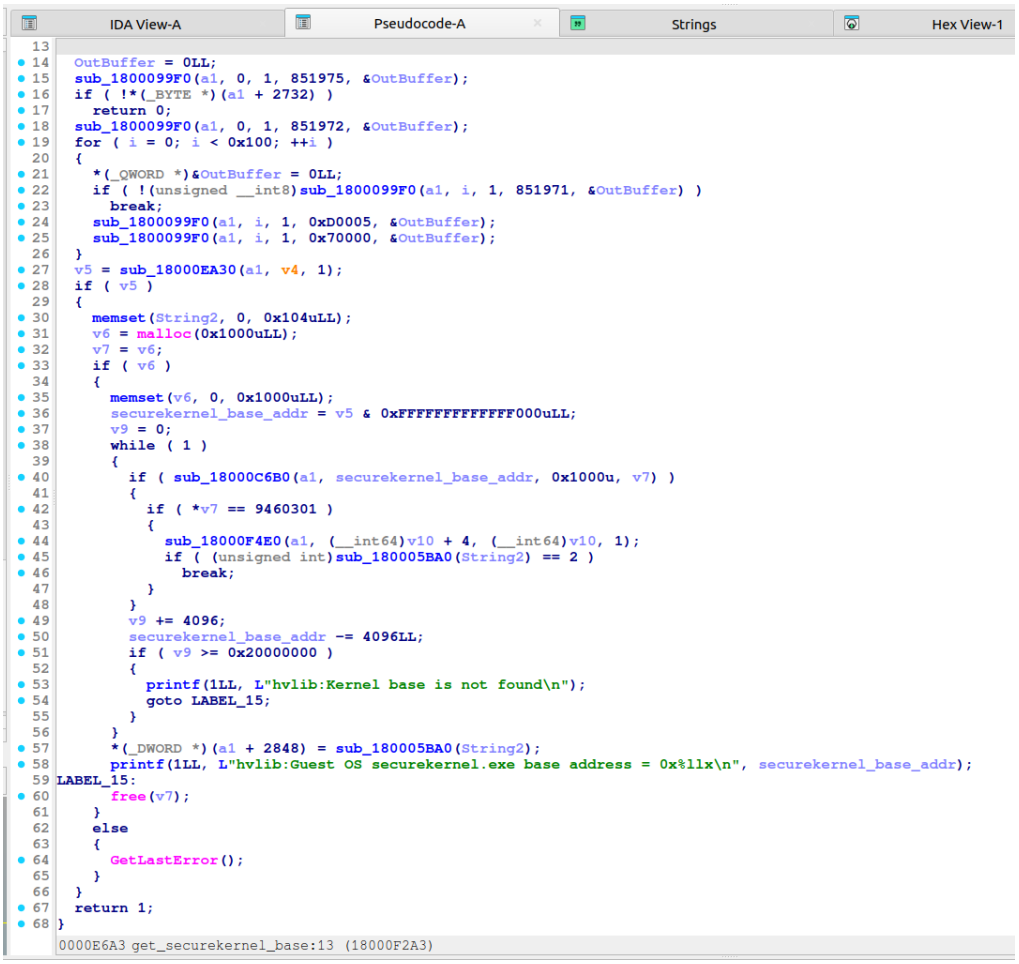


Figure 35: Decompilation of the `hvlib.dll` library logic to retrieve the Secure Kernel virtual base address

if an update is necessary to the user-space client binary to send a different Secure System Call Number (SSCN), the binary needs to be recompiled and re-transferred to the debuggee VM. This workflow works, but is time-consuming. As described in the KernelForge project [51], it would be a much more developer-friendly method to expose this functionality through a Python binding. This means that interaction with the custom kernel driver can be done through a user-space library that is reachable from Python. Since Python is an interpreted language, no recompilation is necessary anymore and therefore eliminates the use of Visual Studio. This makes it more developer-friendly for future researchers to interact with the Secure Kernel and therefore lowers the barrier for researching the Secure Kernel.

In order to make this possible, the current user-space client binary needs to be rewritten to a library that can be imported by Python through C-bindings for Python. Furthermore, the setup for using the user-space client library together with Python needs to be documented. This can then be hosted as an example project for future researchers looking into ways to interact with the Secure Kernel in a developer-friendly way, similar to KernelForge.

7.4 Research interaction between NT Kernel and Secure Kernel for Normal-mode Services

During our research in Section 5.2, we found that the German Federal Office for Information Security published a report related to evaluating the security of Virtualization Based Security (VBS) [36]. This report looks extensively into all the aspects of VBS by reverse engineering the Windows operating system, similar to what we have done in our own research. On page 34, they describe a 'Normal-mode Services' mechanism in the Secure Kernel. Normal-mode Services are services implemented in the NT Kernel which are used by the Secure Kernel to perform certain operations such as process management, registry, and filesystem input/output [36]. This introduces new potential attack surfaces for the Secure Kernel: the Secure Kernel interacts with the NT Kernel and will use the output of the NT Kernel for further operations. During our research, we have only researched the interaction where the NT Kernel interacts with the Secure Kernel, not the Secure Kernel interacting with the NT Kernel. Therefore, for future research, the interaction of the Secure Kernel with the NT Kernel through Normal-mode Services can be further researched to see if any exploitation is possible through this interaction channel. For example, what can a malicious kernel module do with Normal-mode Services to exploit the Secure Kernel through the NT Kernel? The research done in the report of the German Federal Office for Information Security can be used as a starting point to investigate Normal-mode Services.

7.5 Research CVEs with public exploits

As we have seen in Section 5.2 and Section 5.3, it is worth looking into Secure Kernel n-day vulnerabilities and trying to use the gained knowledge to find a zero-day vulnerability in the Secure Kernel. We have conducted our research using only very little information about CVEs for the Secure Kernel. Furthermore, there is no public proof-of-concept available for the n-day we have analyzed in Section 5.2. However, having a public proof-of-concept can greatly help in understanding the bug and the attack vector used for exploitation.

A different approach that can be used in future research is to understand a Secure Kernel bug based on a public proof-of-concept exploit, if available. For example, a public proof-of-concept related to loading an Enclave from the NT Kernel to the Secure Kernel is available for an older build version of Windows 10[12]. This proof-of-concept can be analyzed to understand the underlying bug, which may help in gaining new knowledge about the NT Kernel and Secure Kernel and their interaction. In the future, there may be more proof-of-concepts publicly published, which can be used as a starting point for researching vulnerabilities in the Secure Kernel.

[12] `https://gist.github.com/hfiref0x/1ac328a8e73d053012e02955d38e36a8`

8 Conclusions

In this section, we reflect on the decisions, methods, and results of this thesis. This includes summarizing and reflecting on various decisions and assumptions throughout our research. Furthermore, throughout our research, there were some unexpected situations that resulted in interesting conclusions.

We have made several conclusions throughout our research:

1. In our experience, understanding the Secure Kernel is difficult (see Section 8.1).
2. In our experience, debugging the Secure Kernel is difficult (see Section 8.2).
3. In our experience, interacting with the Secure Kernel is difficult (see Section 8.3).
4. In our experience, exploiting the Secure Kernel is difficult (see Section 8.4).
5. Not every CVE can be reproduced in the virtualized debugging setup (see Section 8.5).
6. Use VMWare for debugging the Secure Kernel (see Section 8.7).
7. Collaboration with experienced researchers is highly recommended (see Section 8.8).
8. Using the knowledge gained from analyzing n-day security bugs, it is possible to discover zero-day security bugs in the Secure Kernel (see Section 8.9).

Furthermore, we have made the following contributions:

1. We have improved documentation related to the Secure Kernel (see Section 2).
2. We have documented a new improved Secure Kernel debugging setup.
3. We have written proof-of-concept code to trigger a Secure Kernel n-day security bug.

8.1 The difficulty of understanding the Secure Kernel

Understanding the Secure Kernel was challenging. While we have shown in Section 2 that there are many online resources available explaining parts of the Secure Kernel, it is still a difficult task. All Secure Kernel code is undocumented, and only some resources are available directly from Microsoft that are related to Secure Kernel functionalities (such as an Enclave, see Section 5.2). The reason for this is that the Secure Kernel is not available to be used by third parties (with some exceptions such as an Enclave), so Microsoft does not have to publish resources about the Secure Kernel.

Our gut feeling is that other researchers who have conducted successful research on the Secure Kernel have:

1. Extensive internal documentation and/or experience with components of and interaction with the Secure Kernel.
2. Access to (parts of the) source code of the NT Kernel and/or Secure Kernel, through Microsoft or other means.

For example, bugs discovered and reported by the Microsoft Security Response Center (MSRC) itself will likely have access to the source code of the Secure Kernel, since MSRC is part of Microsoft itself. There are also bugs reported by third-party researchers, who may work in larger teams over a long period to understand and document the Secure Kernel, which helps in discovering bugs. In our experience, it is challenging to discover

bugs and exploit the Secure Kernel within a few months without any prior knowledge about the NT Kernel or Secure Kernel.

Not all documentation related to the Secure Kernel is internal documentation within an organization. There are some online resources that greatly helped with understanding the Secure Kernel. These resources are [36], [52], and [6], which describe in much detail important parts of the Secure Kernel.

8.2 The difficulty of debugging the Secure Kernel

Creating a debugging environment for the Secure Kernel was more difficult than expected, since it requires a very specific setup that is prone to errors or stability issues. It was not clear beforehand that specialized hardware is required in order to debug the Secure Kernel on physical hardware (see Section 4.3). However, we were able to successfully create and document a new (improved) Secure Kernel debugging setup which is one of the contributions of our research.

There are several Secure Kernel debugging environments documented [56] [49] [32]. The LiveCloudKd setup as described in Section 4.3.1 appears to be the most commonly used Secure Kernel debugging setup, since it is best documented [67] [32]. However, after reproducing this setup, we conclude that LiveCloudKd requires a very specific setup and suffers from stability issues. It was not possible to consistently attach the Secure Kernel debugger, and the reason for this was unknown. Therefore, we have concluded that we cannot use this setup, and therefore we have developed our own Secure Kernel debugging setup (see Section 4.2.2). The common factor between these setups is that they use a virtualized setup and do not use physical hardware, but this requirement has never been stated explicitly anywhere.

Another problem in debugging the Secure Kernel is that Microsoft does not publish the symbols for the `hvix64.exe` hypervisor binary. Therefore, understanding and reverse engineering this binary is challenging. Understanding the hypervisor binary may help in discovering how the hypervisor interacts with the Secure Kernel. However, our Secure Kernel debugging QEMU/KVM setup described in Section 4.2.2 does not make use of the hypervisor binary and is therefore more trivial to reproduce compared to research by Quarkslab [56].

8.3 The difficulty of interacting with the Secure Kernel

8.3.1 Custom kernel driver development

For our Exploitability Assessment in Section 5, we have interacted with the Secure Kernel (VTL1, ring 0) by writing a custom kernel driver for the NT Kernel (see Section 4.4.2) as well as writing a custom Enclave (see Section 5.2.2). In our experience, interacting with the Secure Kernel (VTL1, ring 0) has been more difficult than expected. Even though the Secure Kernel has existed for over a decade, there has not been a lot of documentation on how to interact with the Secure Kernel. Microsoft does not provide any documentation for this, which makes sense since the Secure Kernel is not supposed to be used by third parties (besides the Enclave functionality). But there has been a surprising lack of documentation or example projects interacting with the Secure Kernel through undocumented functionalities.

In Section 4.4.1, we tried to use the 'KernelForge' project, which has documented a method to interact with the Secure Kernel through the NT Kernel. The KernelForge project has not been updated in over four years, which made it difficult to use. There were various installation problems that took too much time, and the KernelForge project

itself is complicated to understand. Therefore, we decided to look into writing our own custom kernel driver (see Section 4.4.2).

Since we do not have any experience with writing a Windows kernel driver, we have searched for documentation and example code on how to write a custom kernel driver to interact with the Secure Kernel. We have found that Alex Ionescu has written a blog post about writing a Hyper-V "bridge" kernel driver for fuzzing purposes [37]. At first, this blog post was promising, since it explains in a lot of detail how to write a custom kernel driver with the exact purpose of interacting with the Secure Kernel. However, in the end, we were unable to reproduce his research. The blog post only provides code snippets for the implementation of certain functions and assumes a lot of knowledge about Windows kernel driver development and the Secure Kernel. There is no repository given which includes the Visual Studio project configurations to build and deploy this custom kernel driver. Furthermore, the blog post consists of three parts but only the first two are published which makes the documentation incomplete. While more experienced Windows kernel driver developers may be able to reproduce his research, we were not. Therefore, we decided to write our own custom kernel driver which is described in Section 4.4.2.2.

During the development of our own custom kernel driver, we have found that the most helpful resources for creating a custom Windows kernel driver were documented by the game cheating and modding community [31] [45]. We found various tutorials and forum posts which were helpful for debugging issues as well as implementing common functionality in a Windows kernel driver [5] [72]. This is because the game cheating and modding community has a lot of resources available for bypassing anti-cheat, which is often done by leveraging the Windows kernel to bypass user-mode anti-cheat.

8.3.2 Custom Enclave development

In Section 5.2.2, we developed a custom Enclave to trigger a Time-Of-Check Time-Of-Use (TOCTOU) bug in the Secure Kernel. The development of the custom Enclave was necessary in order to interact with the Secure Kernel from VTL1 ring 3 to VTL1 ring 0. Furthermore, it was required in order to trigger the TOCTOU bug in the Secure Kernel, since the exposed functionality was only available for Enclaves or Trustlets, and Trustlets cannot be developed by third parties.

While there is some documentation available from Microsoft about developing a custom Enclave [14], it was still difficult to implement the custom Enclave in such a way that it was possible to trigger the TOCTOU bug. This is because Enclaves have different available functionalities compared to normal user-space programs, since Enclaves run in VTL1 instead of VTL0. Microsoft does provide documentation about publicly available functionalities, but these were not sufficient in order to trigger the TOCTOU bug. We have reverse engineered the `vertdll.dll` library as well as other core libraries to understand the underlying interaction between an Enclave and the Secure Kernel. We then developed the custom Enclave to be able to directly interact with the Secure Kernel without using core library functionalities, so we have complete control over the interaction (see Section 5.2.3). This allowed us to trigger the TOCTOU bug, which is described in Section 5.2.4.

Development was also difficult due to the lack of debugging capabilities through WinDBG for debugging our custom Enclave. Even though our custom Enclave was properly marked as debuggable, WinDBG was unable to properly break on inserted breakpoints. However, by using the GDB debugger directly attached to the VMWare debuggee VM (see Section 4.2.5) combined with hardware breakpoints, we were able to debug the

code execution of our custom Enclave. Furthermore, debugging our custom Enclave through GDB also allowed us to see the Secure Kernel execution by single-stepping over a `syscall` instruction. Still, WinDBG is the preferred debugging method since it better integrates with the Windows ecosystem. For example, WinDBG is able to properly detect allocated memory regions and load symbols for libraries and executables.

8.4 The difficulty of exploiting the Secure Kernel

In our experience, exploiting the Secure Kernel is a difficult task. In Section 6, we began working on the exploit for the Time-Of-Check Time-Of-Use (TOCTOU) n-day vulnerability we reproduced in Section 5.2. During our investigation of security mitigations in the Secure Kernel in Section 6.1, we discovered a lack of public documentation about security mitigations in the Secure Kernel as well as exploitations techniques for the Secure Kernel. We only found a Black Hat talk from 2020 about exploiting the Secure Kernel [60] with an accompanying recording [61]. In this talk, information was shared about exploit mitigations in the Secure Kernel as well as an exploit strategy. For the exploit mitigations, we noticed that the 'hardcoded addresses' in the slides used to bypass Kernel Address Space Layout Randomization (KASLR) no longer exist, at least at that address offset. This means that some change to the Secure Kernel KASLR implementation has been made by Microsoft since 2020.

The exploitation technique using the Secure Kernel Patch Guard (SKPG) structure located in memory of the Secure Kernel is still a valid attack vector. Microsoft has not hardened this implementation to prevent code execution by overwriting the `Runtime CheckRoutine` function pointer. This means that with a Secure Kernel address leak or an arbitrary read primitive, it is possible to defeat KASLR and continue exploiting the Secure Kernel.

However, due to KASLR, we were unable to find the Page Table (PT), which is necessary to update the Page Table Entry (PTE) corresponding to a shared memory page between the NT Kernel and the Secure Kernel. It seems that this logic has been updated in the last five years and the PT does not have a fixed location in memory anymore. Due to time constraints, we were not able to find the PT in virtual memory, which is required for exploitation of the Secure Kernel.

8.5 Not every CVE can be reproduced in the virtualized debugging setup

From the research done in Section 5.1.2, we can conclude that it is not possible to reproduce every CVE on a given Windows installation. Especially with the QEMU/KVM virtualized debugging setup as described in Section 4.2.2, virtualization can make a big difference in reproducing a specific bug. We found out during our research into CVE-2024-43528 that the code changes related to the knowledge base update can only be triggered when the Windows machine is booting or going into hibernation. Since a custom kernel module is not loaded yet on boot, it is most likely that the bug is supposed to be triggered through hibernation. However, hibernation is not a supported functionality when running Windows virtualized through QEMU/KVM since it requires firmware support, which is not provided by QEMU/KVM when Hyper-V is enabled.

Therefore, it is important to keep in mind that when analyzing CVEs, the setup in which the bug is being reproduced matters. The Windows operating system is complex and has many functionalities that are not supported on every hardware or execution platform. This sometimes makes it impossible to execute and debug code related to a CVE.

8.6 KB-number updates are unreliable

After our research into CVE-2024-43528 in Section 5.1, we discovered that the Microsoft Security Response Center has made some mistakes and unusual assignments of Knowledge Base (KB) numbers. A knowledge base number is a number assigned to a downloadable (security) update, which is referred to by the Microsoft Security Response Center when a patch is released for a specific Windows system that patches the CVE. In Section 5.1.3, we created an overview in Table 5 of the KB-numbers related to CVE patches for Windows 11 22H2 for x64-based Systems. In this table, it can be seen that the patch for CVE-2024-43528 (which we extensively analyzed in Section 5.1) is assigned KB-number 5046633. However, there are four other CVEs that are also assigned this KB-number. In Section 5.1.1, we have shown in Figure 18 and Figure 19 the patch difference for KB5046633. This is only a few lines of code, while the Microsoft Security Response Center claims that this is a patch for five different CVEs.

We think this is very likely to be a mistake by the Microsoft Security Response Center. The knowledge base number should probably be referencing the update KB5044285 and/or KB5044380, which is the update before KB5046633 and contains a lot more code changes. However, it is difficult to prove this since we do not know the exact details of the CVEs. In Table 6 we have listed all KB-numbers for the CVEs for Windows 11 Version 24H2 for x64-based Systems and from that overview we can conclude that since the KB-number updates do not match the updates for Windows 11 Version 22H2 for x64-based Systems, there has been made mistakes by Microsoft Security Response Center when publishing the updates. From this we conclude that KB-numbers are unreliable and should not be used as a reference for discovering CVE details.

8.7 Use VMWare for debugging the Secure Kernel

8.7.1 Using QEMU/KVM for debugging the Secure Kernel

In Section 4.2.2, we have described a QEMU/KVM setup to use for Secure Kernel debugging. Besides debugging the Secure Kernel, this setup can also be used for debugging the NT Kernel, hypervisor, the Windows boot loader, and the EFI Windows loader. This debugging setup has been made using virtualization provided by QEMU/KVM and managed through Virt-Manager [44]. We started using QEMU/KVM in this research since that was the debugging setup already installed and quickly available. Furthermore, we already had experience with NT Kernel debugging through the QEMU/KVM debugging setup, and therefore we decided to extend this to Secure Kernel debugging. While this setup is usable, it is not perfect. There are some problems with this setup:

1. The GDB stub provided by QEMU/KVM for Secure Kernel debugging is not stable. Stepping through instructions with GDB does not work at all. Furthermore, inserting hardware breakpoints sometimes fails without a clear reason. Encountering one of these bugs will result in a hang of the debuggee VM and requires a full reboot of the debuggee VM in order to resolve.
2. QEMU/KVM does not support snapshotting the live VM state when certain configurations are set, such as using EFI for the bootloader. This is a problem since, in order to make use of the Secure Kernel, it is required to have the EFI bootloader enabled. Furthermore, we made use of 'virtio' drivers for shared filesystems and display emulation, which are all not supported for snapshotting a live VM state. This is a problem since having the possibility to make a snapshot of a live VM makes it possible to easily revert to a previous state when a bug is encountered or an irreversible action is taken. For example, as soon as the custom kernel driver is loaded into memory, it requires a reboot of the VM in order to delete it. A

snapshot of the live VM state just before loading the custom kernel driver would significantly speed up the process of replacing the custom kernel driver with an updated version.

3. Enabling Virtualization Based Security (VBS) in the debuggee VM results in poor performance. The Windows UI is slow to respond, and it takes about ten seconds to copy a file of only a few kilobytes. Furthermore, a reboot cycle takes about five minutes during our testing. While this performance is still in a barely usable state, it makes the Exploitability Assessment of Section 5, as well as the development of the setup to interact with the Secure Kernel as described in Section 4.4, take more time than necessary.

4. QEMU/KVM is not platform-independent. It is required to have a Linux host in order to make use of QEMU/KVM virtualization. While there are alternative setups possible when using Windows as the host operating system (see Section 4.3.1), it would be better to have a fully platform-independent solution available.

We used the QEMU/KVM Secure Kernel debugging setup during Section 5.1. However, the poor performance of the debuggee VM became a bottleneck for our research. Therefore, we decided to move back to the setup phase to improve our Secure Kernel debugging setup.

8.7.2 Using VMWare for debugging the Secure Kernel

We know from the research of Quarkslab [56] that it is possible to use VMWare with a GDB stub to allow for physical memory access. Furthermore, VMWare is a platform-independent solution, and it supports snapshotting the state of a VM even with the EFI bootloader. Therefore, we decided to choose VMWare as the virtualization software used for our improved Secure Kernel debugging setup. A detailed explanation of our setup can be found in Section 4.2.5.

The VMWare Secure Kernel debugging setup has the following improvements:

1. The GDB stub provided by VMWare for Secure Kernel debugging is stable. We did not notice blocking issues with stepping through instructions. Sometimes there is a significant slowdown of several seconds before the next instruction is hit, but this is still a significant improvement compared to QEMU/KVM.

2. VMWare supports snapshotting the VM state, which also works even with an active WinDBG and GDB session. Reverting to a previous snapshot does stop the GDB session, but reconnecting after applying the snapshot is possible, and debugging can continue as normal.

3. The performance of the debuggee VM with VBS enabled is good. The VM can run with several cores without issues with debugging through GDB or WinDBG. Furthermore, we do not notice any significant slowdown or overhead when VBS is enabled in Windows. Note that we do not enforce memory integrity within the debuggee VM, since it is not required for loading the Secure Kernel.

4. VMWare is platform-independent. The debugging setup described in Section 4.2.5 can be reproduced on Windows, macOS, and Linux. However, our debugging setup depends on GDB, which would not make it easy to support Windows. An alternative to GDB is to use IDA to connect and interact with the GDB stub, which is supported on Windows.

The downside, however, is that VMWare is commercial software and not open-source. While at the time of writing it is possible to use VMWare for personal use, this pol-

icy may change in the future, which may prevent individuals from reproducing this debugging setup in the future.

8.8 Collaborate with experienced researchers

The Secure Kernel is a complex topic with very few documentation available. Therefore talking with people with hands-on experience with the Secure Kernel and/or (Windows) exploitation is very useful. We think that this collaboration has been key to the success of this research and therefore this collaboration is highly recommended for other researchers as well working on the Secure Kernel. We would like to give special thanks to Daan Keuper and Thijs Alkemade from Computest for supervising this research and especially helping with working on the Exploitation phase of the Secure Kernel (see Section 6). Furthermore, special thanks to Cedric Van Bockhaven from Outflank who also helped as an external advisor for Secure Kernel internals and giving advise related to finding and exploiting bugs.

8.9 Zero-day vulnerability in the Secure Kernel

In Section 5.3, we discuss a zero-day we found in the Secure Kernel. We found this zero-day vulnerability by looking for similar vulnerable code patterns to the patterns we have seen by analyzing the n-day vulnerability we discussed in Section 5.2. We have looked for the specific vulnerable code pattern that resulted in the Time-Of-Check Time-Of-Use (TOCTOU) vulnerability (see Section 5.2 and Section 5.3).

An interesting finding is that the n-day vulnerability of Section 5.2 has been fixed by Microsoft, but the same vulnerable code pattern for the zero-day vulnerability of Section 5.3 has not been fixed by Microsoft. From this, we can conclude that it is likely that Microsoft does not apply some sort of static code analysis tooling during the development of the Secure Kernel or triage of Secure Kernel bugs, since this vulnerable code pattern can trivially be found using tooling like CodeQL [33].

A reason that this zero-day vulnerability has not been patched could be that the functionality cannot be triggered by a malicious third party. Since Microsoft Security Response Center requires you to have a proof-of-concept exploit (with a video showcasing the exploit) before they will look into the vulnerability, it is not possible for vulnerability researchers to write a proof-of-concept and thus report this vulnerability.

However, our method of analyzing n-day vulnerabilities has proven to help finding zero-day vulnerabilities through learning about patched vulnerable code patterns in the Secure Kernel and discovering similar new code patterns in other parts of the Secure Kernel.

References

[1] Cedric Van Bockhaven. *Secure Enclaves for Offensive Operations (Part I)*. Corporate blog post. Feb. 3, 2025. URL: https://www.outflank.nl/blog/2025/02/03/secure-enclaves-for-offensive-operations-part-i/.

[2] Cedric Van Bockhaven and Matteo Malvica. *Secure Enclaves for Offensive Operations*. Blog post will be published in summer 2025. URL: https://www.outflank.nl/blog/2025/06/16/secure-enclaves-for-offensive-operations-part-ii/.

[3] Hans Kristian Brendmo. "Live forensics on the Windows 10 secure kernel." MA thesis. NTNU, 2017.

[4] Microsoft Security Response Center. *Windows Secure Kernel Mode Elevation of Privilege Vulnerability - CVE-2024-43528*. URL: https://msrc.microsoft.com/update-guide/vulnerability/CVE-2024-43528.

[5] Amey Chavan. *Development & Demo of Windows Kernel Driver*. Personal blog post. 2022. URL: https://apchavan.medium.com/development-demo-of-windows-kernel-driver-47fc2150e128.

[6] Adrien Chevalier. *Virtualization Based Security - Part 1: The boot process*. Corporate blog post. 2017. URL: https://www.amossys.fr/insights/blog-technique/virtualization-based-security-part1/.

[7] Adrien Chevalier. *Virtualization Based Security - Part 2: kernel communications*. Corporate blog post. 2017. URL: https://www.amossys.fr/insights/blog-technique/virtualization-based-security-part2/.

[8] clearbluejar. *Ghidriff*. 2025. URL: https://github.com/clearbluejar/ghidriff.

[9] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel Documentation. 2016. URL: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3c-part-3-manual.pdf.

[10] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel Documentation. 2024. URL: https://cdrdv2-public.intel.com/835781/325462-sdm-vol-1-2abcd-3abcd-4.pdf.

[11] Microsoft Corporation. *APIs available in VBS enclaves*. Microsoft Learn page. 2024. URL: https://learn.microsoft.com/en-us/windows/win32/trusted-execution/available-in-enclaves.

[12] Microsoft Corporation. *Credential Guard overview*. Microsoft Learn page. 2024. URL: https://learn.microsoft.com/en-us/windows/security/identity-protection/credential-guard/.

[13] Microsoft Corporation. *DebugView v4.90*. Microsoft Learn page. 2021. URL: https://learn.microsoft.com/en-us/sysinternals/downloads/debugview.

[14] Microsoft Corporation. *Development guide for Virtualization-based security (VBS) Enclaves*. Microsoft Learn page. 2024. URL: https://learn.microsoft.com/en-us/windows/win32/trusted-execution/vbs-enclaves-dev-guide.

[15] Microsoft Corporation. *Enable virtualization-based protection of code integrity*. Microsoft Learn page. 2024. URL: https://learn.microsoft.com/en-us/windows/security/hardware-security/enable-virtualization-based-protection-of-code-integrity.

[16] Microsoft Corporation. *Hyper-V Architecture*. Microsoft Learn page. 2022. URL: https://learn.microsoft.com/en-us/virtualization/hyper-v-on-windows/reference/hyper-v-architecture.

[17] Microsoft Corporation. *Hypercall Interface*. Microsoft Learn page. 2022. URL: https://learn.microsoft.com/en-us/virtualization/hyper-v-on-windows/tlfs/hypercall-interface.

[18] Microsoft Corporation. *Hypervisor Top Level Functional Specification*. Microsoft Learn page. 2022. URL: https://learn.microsoft.com/en-us/virtualization/hyper-v-on-windows/tlfs/tlfs.

[19] Microsoft Corporation. *Isolated User Mode (IUM) Processes*. Microsoft Learn page. 2021. URL: https://learn.microsoft.com/en-us/windows/win32/procthread/isolated-user-mode--ium--processes.

[20] Microsoft Corporation. *Partitions*. Microsoft Learn page. 2021. URL: https://learn.microsoft.com/en-us/virtualization/hyper-v-on-windows/tlfs/partition-properties.

[21] Microsoft Corporation. *ProbeForWrite function (wdm.h)*. Microsoft Learn page. 2023. URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-probeforwrite.

[22] Microsoft Corporation. *Provision a computer for driver deployment and testing (WDK 10)*. Microsoft Learn page. 2024. URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/provision-a-target-computer-wdk-8-1.

[23] Microsoft Corporation. *Tutorial: Write a Hello World Windows Driver (Kernel-Mode Driver Framework)*. Microsoft Learn page. 2025. URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/writing-a-very-small-kmdf--driver.

[24] Microsoft Corporation. *Virtual Secure Mode*. Microsoft Learn page. URL: https://learn.microsoft.com/en-us/virtualization/hyper-v-on-windows/tlfs/vsm.

[25] Microsoft Corporation. *Virtualization-based security (VBS) Enclave Sample Code*. GitHub repository. 2024. URL: https://github.com/microsoft/Windows-classic-samples/tree/main/Samples/VbsEnclave.

[26] Microsoft Corporation. *Windows 10 Version 2004 - English 64-bit ISO*. 2020. URL: https://archive.org/details/win-10-2004-english-x-64_202010.

[27] Microsoft Corporation. *winenclaveapi.h header*. Microsoft Learn page. 2025. URL: https://learn.microsoft.com/en-us/windows/win32/api/winenclaveapi/.

[28] Bruce Dawson. *Symbols the Microsoft Way*. Personal blog post. 2013. URL: https://randomascii.wordpress.com/2013/03/09/symbols-the-microsoft-way/.

[29] Benjamin Delpy. *Mimikatz*. 2024. URL: https://github.com/gentilkiwi/mimikatz.

[30] Kyle Dotterrer. *Trustlets (Secure Processes)*. GitHub repository README. 2019. URL: https://github.com/turingcompl33t/windows-internals/blob/eba1bc5401b7b57b58d1a966e7dbc633b8e3f87c/Processes/Trustlets.md.

[31] Frostiest. *[Coding] GetKernelBase*. Personal forum post. 2020. URL: https://www.unknowncheats.me/forum/general-programming-and-reversing/427419-getkernelbase.html.

[32] Gerhart. *Hyper-V live debugging*. GitHub README. 2024. URL: https://github.com/gerhart01/LiveCloudKd/blob/b0418bef34087e8a921a976cfbf990dceffb3314/ExdiKdSample/LiveDebugging.md.

[33] Inc. GitHub. *CodeQL*. 2025. URL: https://codeql.github.com/.

[34] Google. *BinDiff*. 2025. URL: https://github.com/google/bindiff.

[35] Google. *BinExport*. 2024. URL: https://github.com/google/binexport.

[36] Federal Office for Information Security. *Work Package 6: Virtual Secure Mode*. German Federal Office for Information Security. 2018. URL: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Cyber-Sicherheit/SiSyPHus/Workpackage6_Virtual_Secure_Mode.pdf?__blob=publicationFile.

[37] Alex Ionescu. *Writing a Hyper-V "Bridge" for Fuzzing — Part 1: WDF*. Personal blog post. 2019. URL: https://www.alex-ionescu.com/writing-a-hyper-v-bridge-for-fuzzing-part-1-wdf/.
[38] Metis IT. *Virtualization Issues with VMware Workstation Pro on Windows 11 version 24H2*. Corporate blog post. Apr. 4, 2025. URL: https://metisit.com/nieuws/virtualization-issues-with-vmware-workstation-pro-on-windows-11-version-24h2/.
[39] joxeankoret. *Diaphora*. 2024. URL: https://github.com/joxeankoret/diaphora.
[40] kceiw. *Hibernation is not available to the kvm guest*. KVM forum post. 2024. URL: https://www.reddit.com/r/kvm/comments/1bskyqx/comment/kzcepc1/.
[41] WithSecure™ Labs. *Ventures into Hyper-V - Fuzzing hypercalls*. Corporate blog post. 2019. URL: https://labs.withsecure.com/publications/ventures-into-hyper-v-part-1-fuzzing-hypercalls.
[42] m417z. *Introducing Winbindex - the Windows Binaries Index*. Personal blog post. 2020. URL: https://m417z.com/Introducing-Winbindex-the-Windows-Binaries-Index/.
[43] m417z. *Winbindex*. URL: https://winbindex.m417z.com/.
[44] Virtual machine Manager contributors. *Virt-Manager*. 2025. URL: https://virt-manager.org/.
[45] mbn. *The-Kernel-Driver-GUIDE*. GitHub repository. 2024. URL: https://github.com/mbn-code/The-Kernel-Driver-Guide-External.
[46] Kevin McGrath. *Hyper-V Automation for Windows Patch Diffing*. Corporate whitepaper. 2022. URL: https://www.trellix.com/assets/docs/atr-library/tr-hyper-v-automation-for-windows-patch-diffing.pdf.
[47] Salma El Mohib. *A virtual journey: From hardware virtualization to Hyper-V's Virtual Trust Levels*. Corporate blog post. 2021. URL: https://blog.quarkslab.com/a-virtual-journey-from-hardware-virtualization-to-hyper-vs-virtual-trust-levels.html.
[48] Amit Moshel. *Virtualization-Based Security with Hyper-V: Exploring Hyper-V mechanisms and Virtualization Based Security*. Personal blog post. Feb. 8, 2025. URL: https://amitmoshel1.github.io/posts/virtualization-based-security-with-hyper-v-exploring-hyper-v-mechanisms-and-virtualization-based-security/.
[49] Camille Mougey. *Debugging Secure Kernel*. GitHub README. 2020. URL: https://github.com/commial/experiments/tree/1cc5344457a671f090a9303e38840436c427bf0f/debugging-secure-kernel.
[50] Dmytro Oleksiuk. *Hyper-V Backdoor*. GitHub repository. 2024. URL: https://github.com/Cr4sh/s6_pcie_microblaze/tree/master/python/payloads/DmaBackdoorHv.
[51] Dmytro Oleksiuk. *Kernel Forge library for Windows*. GitHub repository. 2021. URL: https://github.com/Cr4sh/KernelForge.
[52] Mark E. Russinovich Pavel Yosifovich Alex Ionescu and David A. Solomon. *Windows Internals Seventh Edition Part 1 System architecture, processes, threads, memory management, and more*. Online PDF. 2024. URL: https://empyreal96.github.io/nt-info-depot/Windows-Internals-PDFs/Windows%20System%20Internals%207e%20Part%201.pdf.
[53] libvirt project. *Sharing files with Virtiofs*. libvirt documentation. URL: https://libvirt.org/kbase/virtiofs.html.
[54] pwndbg. *pwndbg*. 2025. URL: https://pwndbg.re/.
[55] QEMU. *GDB usage*. 2025. URL: https://www.qemu.org/docs/master/system/gdb.html#examining-physical-memory.

[56] Quarkslab. *Debugging Windows Isolated User Mode (IUM) Processes*. Corporate blog post. 2023. URL: https://blog.quarkslab.com/debugging-windows-isolated-user-mode-ium-processes.html.
[57] Quarkslab. *QBinDiff: A modular diffing toolkit*. Corporate blog post. 2023. URL: https://blog.quarkslab.com/qbindiff-a-modular-diffing-toolkit.html.
[58] Quarkslab. *Quokka*. 2025. URL: https://github.com/quarkslab/quokka.
[59] Conor Richard. *Exploring Virtual Memory and Page Structures*. Personal blog post. 2021. URL: https://blog.xenoscr.net/2021/09/06/Exploring-Virtual-Memory-and-Page-Structures.html.
[60] Daniel King Saar Amar. *Breaking VSM by Attacking Secure Kernel - Hardening Secure Kernel through Offensive Research*. Presentation at Black Hat USA 2020. 2020. URL: https://i.blackhat.com/USA-20/Thursday/us-20-Amar-Breaking-VSM-By-Attacking-SecureKernal.pdf.
[61] Daniel King Saar Amar. *Breaking VSM by Attacking SecureKernel*. Recording of Black Hat USA 2020 presentation. 2020. URL: https://www.youtube.com/watch?v=pm1ejZ3LkYU.
[62] Off By One Security. *Getting Started with Debugging Hyper-V for Vulnerability Research, Part 2*. Personal YouTube channel. 2024. URL: https://www.youtube.com/live/9utI4qKCWH4.
[63] Alan Sguigna. *JTAG debug of Windows Hyper-V / Secure Kernel with WinDbg and EXDI: Part 1*. Corporate blog post. 2024. URL: https://www.asset-intertech.com/resources/blog/2024/01/jtag-debug-of-windows-hyper-v-secure-kernel-with-windbg-and-exdi-part-1/.
[64] Yarden Shafir. *HyperGuard - Secure Kernel Patch Guard: Part 1 - SKPG Initialization*. Windows Internal blog post. 2022. URL: https://windows-internals.com/hyperguard-secure-kernel-patch-guard-part-1-skpg-initialization/.
[65] Yarden Shafir. *HyperGuard - Secure Kernel Patch Guard: Part 2 - SKPG Extents*. Windows Internal blog post. 2022. URL: https://windows-internals.com/hyperguard-secure-kernel-patch-guard-part-2-skpg-extents/.
[66] Yarden Shafir. *HyperGuard Part 3 - More SKPG Extents*. Windows Internal blog post. 2022. URL: https://windows-internals.com/hyperguard-part-3-more-skpg-extents/.
[67] Yarden Shafir. *Secure Kernel Research with LiveCloudKd*. Personal blog post. May 2, 2024. URL: https://windows-internals.com/secure-kernel-research-with-livecloudkd/.
[68] Axel Souchet. *what the fuzz*. 2025. URL: https://github.com/0vercl0k/wtf.
[69] Matt Suiche. *LiveCloudKd*. 2025. URL: https://github.com/msuiche/LiveCloudKd.
[70] swiat. *First Steps in Hyper-V Research*. Microsoft Security Blog. 2018. URL: https://msrc.microsoft.com/blog/2018/12/first-steps-in-hyper-v-research/.
[71] Microsoft Security Team. *Introducing Kernel Data Protection, a new platform security technology for preventing data corruption*. Microsoft Security Blog. 2020. URL: https://www.microsoft.com/en-us/security/blog/2020/07/08/introducing-kernel-data-protection-a-new-platform-security-technology-for-preventing-data-corruption/.
[72] Nikhil John Thomas. *Kernel Exploitation Primer 0x0 - Windows Driver 101*. Personal blog post. 2025. URL: https://ghostbyt3.github.io/blog/Kernel_Exploitation_Primer_0x0.
[73] Samuel Tulach. *From firmware to VBS enclave: bootkitting Hyper-V*. Personal blog post. Dec. 8, 2024. URL: https://tulach.cc/from-firmware-to-vbs-enclave-bootkitting-hyper-v/.
[74] VirusTotal. *API Overview*. VirusTotal documentation. 2025. URL: https://docs.virustotal.com/docs/api-overview.

[75] Peter Viscarola. *Using WinDbg Over KDNet on QEMU-KVM*. Corporate blog post. Oct. 5, 2021. URL: https://www.osr.com/blog/2021/10/05/using-windbg-over-kdnet-on-qemu-kvm/.

[76] wumb0. *Extracting and Diffing Windows Patches in 2020*. Personal blog post. 2020. URL: https://wumb0.in/extracting-and-diffing-ms-patches-in-2020.html.

Appendices

A Hypercall handling of IumInvokeSecureService

Table 8 shows the mapping from Secure System Call Number (SSCN) to the corresponding relevant called function for that SSCN (functions within the `securekernel.exe` binary). However, this only corresponds to the `securekernel.exe` binaries we have reverse engineered in this master thesis. The Secure System Call Numbers (SSCNs) can change between different versions of the Windows operating system. See Section 2.3.2 for information about Hypercalls and Secure System Call Numbers.

Table 8: Hyper-V Hypercall Handling with the Secure System Call Number (SSCN) through `IumInvokeSecureService`

SSCN	Relevant Called Functions
1	SkmmInitializeUserSharedData, SkInitSystem, SkCheckHibernationSupport, SkWritePerfTraceEntry
2	IumpStartProcessor
3	IumpFinishStartProcessor
4	SkUpdateUserSharedDataSystemRoot, SkpgConnect, SkpsRegisterSystemDlls
5	SkeLockProcessorStartup, SkpsRegisterSystemProcess
6	SkpsCreateProcess
7	IumpInitializeProcess
8	SkpsCreateThread
9	SkeReferenceThread, SkeTerminateThread
10	SkpsTerminateThread
11	SkpsRundownProcess
12	SkpsReferenceProcessByHandle, SkpsIsProcessDebuggingEnabled, SkpsEnableDebugging
13	SkeReferenceThread
14	IumpGetSetThreadContext
15	IumpGetSetThreadContext
16	SkpsReferenceProcessByHandle, SkpsSendDebugAttachNotifications
17	SkpsReferenceProcessByHandle, SkmmMapDataTransfer, SkmmGetDebugId, SkmmUnmapDataTransfer
18	SkpsReferenceProcessByHandle, SkmmMapDataTransfer, SkGetOnDemandDebugChallenge, SkmmUnmapDataTransfer
19	SkpsReferenceProcessByHandle
20	IumpRetrieveMailbox
21	SkpsReferenceTrustlet
22	SkmmCreateSecureAllocation
23	SkmmFillSecureAllocation
24	SkmmConvertSecureAllocationToCatalog
25	SkmmCreateSecureImageSection
26	SkmmFinalizeSecureImageHash
27	SkmmFinishSecureImageValidation
28	SkmmPrepareImageRelocations
29	SkmmRelocateImage
30	SkobCloseHandleEx
31	SkmmValidateDynamicCodePages

Continued on next page

Table 8 – continued from previous page

SSCN	Relevant Called Functions
32	SkmmTransferImageVersionResource
33	SkmmSetCodeIntegrityPolicy
34	EntropyProvideData, EntropyPoolTriggerReseedForIum, BCryptGenRandom
35	Unknown
36	SkAllocateHibernateResources
37	SkFreeHibernateResources
38	SkmmConfigureDynamicMemory
39	SkmmReferenceAddressSpace, SkmmDebugProtectVirtualMemory
40	SkmmReferenceAddressSpace, SkmmDebugReadWriteMemory
41	SkmmReferenceAddressSpace, SkmmQueryVirtualMemory
42	SkmiOperateOnLockedNar, SkmiCaptureSecureImageIat
43	SkmmFreeSecureImageIat
44	SkmmApplySecureFixups
45	SkmmMarkImageAsProtected
46	SkmmCreateEnclave
47	SkmmLoadEnclaveData
48	SkmmLoadEnclaveModule
49	SkmmInitializeEnclave
50	SkmiReferenceEnclaveByHandle, SkmmTerminateEnclaveByPointer
51	SkmmDeleteEnclave
52	IumpConnectSwInterrupt
53	SkpgRelaxQuotas
54	IumpRegisterBootDrivers
55	SkGetSkLivedumpDescriptorSize
56	SkLiveDumpStart
57	SkLiveDumpAddBuffer
58	SkLiveDumpSetupBuffer
59	SkLiveDumpFinalize
60	SkLiveDumpAbort
61	SkpsReferenceProcessByHandle, SkiAttachProcess, SkMinidumpWriteDump
62	SkeChangeTimerMode, SkIdleResiliencyActive, SkCsScenarioInstanceId, SkeNotifyConnectedStandbyScenario
63	IumpDispatchQueryProfileInformation
64	SkpsReferenceProcessByHandle, SkpsUpdateFreezeTimeBias
65	SkpsReferenceProcessByHandle, SkmmCreateExposedSecureSection
66	SkmmDeleteExposedSecureSection
67	SkpnpQuerySecureDevice
68	SkpnppUnprotectDevice
69	SkmmDetermineHotPatchType
70	SkmmDetermineHotPatchUndoTableSize
71	SkmmObtainHotPatchUndoTable
72	SkmmApplyHotPatch
73	SkmmRevertHotPatch
74	SkmmPrepareDriverForHotPatch
75	SkProvisionDumpKeys
76	Unknown
77	Unknown
78	Unknown
79	SkSpCreateSecurePool

Continued on next page

Table 8 – continued from previous page

SSCN	Relevant Called Functions
80	SecurePoolDestroy
81	SkSecurePoolAllocate
82	SecurePoolFree
83	SkSecurePoolUpdate
84	SkmmSetImageTracePoint
85	SkTransformDumpKey
86	SkmmPublishSyscallProviderServiceTables
87	SkmmRevokeSyscallProviderServiceTables
192	SkpsReferenceProcessByHandle
193	SkmmValidateSecureImagePages
208	SkmmInitSystem, SkeLoadSystemData, RtlInitializeHistoryTable, SkpsCreateAndPrepareSystemProcess, SkInitSystem, SkhalInitSystem, SkWritePerfTraceEntry
209	SkPerformPeriodicWork
210	SkExecuteWorkItems
211	SkmmReserveProtectedPages
212	SkmmApplyNormalDriverDynamicRelocations
213	SkEtwEnableCallback
214	SkmmInitializeSecurePool, SkInitializeSecurePool
215	SkmmInitializeNtKernelCfg, SkpgInitializeNtKernelCfg
216	SkmiOperateOnLockedNar, SkmiLoadNormalDriver
217	SkmiOperateOnLockedNar, SkmiUnloadNormalDriver
218	SkmiOperateOnLockedNar, SkmiEnableNtosCfgTarget
219	SkmmCompleteSlabConfiguration
220	SkmmReapplyBootDriverHotPatch
221	SkmmInitializeRetpoline
222	SkmiOperateOnLockedNar, SkmiPerformRetpolineFixups
223	SkmmUpdateImportRelocationsOnImage
224	SkmmReapplyImportRelocationsOnImage
225	SkmmGetFunctionOverridesCapabilities
226	SkmmApplyFunctionOverridesOnImage
227	SkmmTranslateKernelShadowStackType, SkmmCreateNtKernelShadowStack
228	SkmmDestroyNtKernelShadowStack
229	SkmmTranslateKernelShadowStackType, SkmmResetNtKernelShadowStack
230	SkmmRegisterSyscallProviderServiceTableMetadata
231	SkmmInitializeSyscallProviders
240	SkobReferenceObjectByHandle, SkmmFlushAddressSpace, SkobpDereferenceObject
241	SkobReferenceObjectByHandle, SkmiFlushRangeList, SkeLowerIrql, SkobpDereferenceObject
242	SkobReferenceObjectByHandle, SkmmSlowFlushRangeList, SkobpDereferenceObject
243	SkmmRemoveProtectedPage
244	SkmmCopyProtectedPage
245	SkmmRegisterProtectedPage
246	SkmmDisambiguateProtectedPage
247	SkmmMakeProtectedPageWritable
248	SkmmMakeProtectedPageExecutable
249	SkmmQueryStrongCodeFeatures

Continued on next page

Table 8 – continued from previous page

SSCN	Relevant Called Functions
250	SkhalEfiInvokeRuntimeService
251	SkLiveDumpCollect
252	SkmmRegisterFailureLog
253	SkmiObtainPartition, SkmiReclaimPartitionPages, SkmiDereferencePartition
255	SkmmSetPlaceholderPages
256	SkeQuerySpeculationFeaturesInformation
257	SkmmProtectKernelDataPage
258	SkpgVerifyPage
259	SkPrepareForHibernate
260	SkPrepareForCrashDump, IumpLimitedDispatchFromNormalDispatch
261	SkhalReportBugCheckProgress
262	ShvlConfigureMemory

B Debuggee VM Virt-Manager Configuration

In Section 4.2.2 the relevant configuration for the QEMU/KVM debuggee VM is explained. For completeness, below the full XML configuration of Virt-Manager is listed for comparison in case of troubleshooting. This Virt-Manager XML configuration was specifically made to make NT Kernel debugging as well as Secure Kernel debugging working with QEMU/KVM through Virt-Manager. Note that this VM is configured to have virtio functionality and spice support enabled, but this should not impact the debugging capabilities. Furthermore, the following QEMU version is used: `QEMU emulator version 6.2.0 (Debian 1:6.2+dfsg-2ubuntu6.24)`

```
<domain xmlns:qemu="http://libvirt.org/schemas/domain/qemu
   /1.0" type="kvm">
<name>Debuggee-VM</name>
<uuid>ab404d5e-49d0-403c-85de-5f6474be4d0e</uuid>
<metadata>
        <libosinfo:libosinfo xmlns:libosinfo="http://
           libosinfo.org/xmlns/libvirt/domain/1.0">
        <libosinfo:os id="http://microsoft.com/win/10"/>
        </libosinfo:libosinfo>
</metadata>
<memory unit="KiB">8388608</memory>
<currentMemory unit="KiB">8388608</currentMemory>
<memoryBacking>
        <source type="memfd"/>
        <access mode="shared"/>
</memoryBacking>
<vcpu placement="static">1</vcpu>
<os>
        <type arch="x86_64" machine="pc-q35-6.2">hvm</type>
        <loader readonly="yes" secure="yes" type="rom">/usr/
           share/OVMF/OVMF_CODE.fd</loader>
        <nvram>/var/lib/libvirt/qemu/nvram/Debuggee-VM_VARS.
           fd</nvram>
</os>
<features>
        <acpi/>
        <apic/>
        <hyperv mode="custom">
        <relaxed state="on"/>
        <vapic state="on"/>
        <spinlocks state="on" retries="8191"/>
        <vendor_id state="on" value="KVMKVMKVM"/>
        </hyperv>
        <kvm>
        <hidden state="on"/>
        </kvm>
        <vmport state="off"/>
        <smm state="on"/>
        <ioapic driver="kvm"/>
</features>
<cpu mode="custom" match="exact" check="partial">
```

```
        <model fallback="allow">Skylake-Client-noTSX-IBRS</
           model>
        <feature policy="disable" name="hypervisor"/>
        <feature policy="require" name="vmx"/>
        <feature policy="disable" name="mpx"/>
</cpu>
<clock offset="localtime">
        <timer name="rtc" tickpolicy="catchup"/>
        <timer name="pit" tickpolicy="delay"/>
        <timer name="hpet" present="no"/>
        <timer name="hypervclock" present="yes"/>
</clock>
<on_poweroff>destroy</on_poweroff>
<on_reboot>restart</on_reboot>
<on_crash>destroy</on_crash>
<pm>
        <suspend-to-mem enabled="no"/>
        <suspend-to-disk enabled="no"/>
</pm>
<devices>
        <emulator>/usr/bin/qemu-system-x86_64</emulator>
        <disk type="file" device="disk">
        <driver name="qemu" type="qcow2" discard="unmap"/>
        <source file="/path/to/storage.qcow2"/>
        <target dev="vda" bus="virtio"/>
        <boot order="1"/>
        <address type="pci" domain="0x0000" bus="0x04" slot
           ="0x00" function="0x0"/>
        </disk>
        <controller type="usb" index="0" model="qemu-xhci"
           ports="15">
        <address type="pci" domain="0x0000" bus="0x02" slot
           ="0x00" function="0x0"/>
        </controller>
        <controller type="pci" index="0" model="pcie-root"/>
        <controller type="pci" index="1" model="pcie-root-
           port">
        <model name="pcie-root-port"/>
        <target chassis="1" port="0x10"/>
        <address type="pci" domain="0x0000" bus="0x00" slot
           ="0x02" function="0x0" multifunction="on"/>
        </controller>
        <controller type="pci" index="2" model="pcie-root-
           port">
        <model name="pcie-root-port"/>
        <target chassis="2" port="0x11"/>
        <address type="pci" domain="0x0000" bus="0x00" slot
           ="0x02" function="0x1"/>
        </controller>
        <controller type="pci" index="3" model="pcie-root-
           port">
        <model name="pcie-root-port"/>
```

```
<target chassis="3" port="0x12"/>
<address type="pci" domain="0x0000" bus="0x00" slot
   ="0x02" function="0x2"/>
</controller>
<controller type="pci" index="4" model="pcie-root-
   port">
<model name="pcie-root-port"/>
<target chassis="4" port="0x13"/>
<address type="pci" domain="0x0000" bus="0x00" slot
   ="0x02" function="0x3"/>
</controller>
<controller type="pci" index="5" model="pcie-root-
   port">
<model name="pcie-root-port"/>
<target chassis="5" port="0x14"/>
<address type="pci" domain="0x0000" bus="0x00" slot
   ="0x02" function="0x4"/>
</controller>
<controller type="pci" index="6" model="pcie-root-
   port">
<model name="pcie-root-port"/>
<target chassis="6" port="0x15"/>
<address type="pci" domain="0x0000" bus="0x00" slot
   ="0x02" function="0x5"/>
</controller>
<controller type="pci" index="7" model="pcie-root-
   port">
<model name="pcie-root-port"/>
<target chassis="7" port="0x16"/>
<address type="pci" domain="0x0000" bus="0x00" slot
   ="0x02" function="0x6"/>
</controller>
<controller type="pci" index="8" model="pcie-root-
   port">
<model name="pcie-root-port"/>
<target chassis="8" port="0x17"/>
<address type="pci" domain="0x0000" bus="0x00" slot
   ="0x02" function="0x7"/>
</controller>
<controller type="pci" index="9" model="pcie-root-
   port">
<model name="pcie-root-port"/>
<target chassis="9" port="0x18"/>
<address type="pci" domain="0x0000" bus="0x00" slot
   ="0x03" function="0x0" multifunction="on"/>
</controller>
<controller type="pci" index="10" model="pcie-root-
   port">
<model name="pcie-root-port"/>
<target chassis="10" port="0x19"/>
<address type="pci" domain="0x0000" bus="0x00" slot
   ="0x03" function="0x1"/>
```

```
</controller>
<controller type="pci" index="11" model="pcie-root-
   port">
<model name="pcie-root-port"/>
<target chassis="11" port="0x1a"/>
<address type="pci" domain="0x0000" bus="0x00" slot
   ="0x03" function="0x2"/>
</controller>
<controller type="pci" index="12" model="pcie-root-
   port">
<model name="pcie-root-port"/>
<target chassis="12" port="0x1b"/>
<address type="pci" domain="0x0000" bus="0x00" slot
   ="0x03" function="0x3"/>
</controller>
<controller type="pci" index="13" model="pcie-root-
   port">
<model name="pcie-root-port"/>
<target chassis="13" port="0x1c"/>
<address type="pci" domain="0x0000" bus="0x00" slot
   ="0x03" function="0x4"/>
</controller>
<controller type="pci" index="14" model="pcie-root-
   port">
<model name="pcie-root-port"/>
<target chassis="14" port="0x1d"/>
<address type="pci" domain="0x0000" bus="0x00" slot
   ="0x03" function="0x5"/>
</controller>
<controller type="pci" index="15" model="pcie-root-
   port">
<model name="pcie-root-port"/>
<target chassis="15" port="0x1e"/>
<address type="pci" domain="0x0000" bus="0x00" slot
   ="0x03" function="0x6"/>
</controller>
<controller type="pci" index="16" model="pcie-to-pci
   -bridge">
<model name="pcie-pci-bridge"/>
<address type="pci" domain="0x0000" bus="0x08" slot
   ="0x00" function="0x0"/>
</controller>
<controller type="sata" index="0">
<address type="pci" domain="0x0000" bus="0x00" slot
   ="0x1f" function="0x2"/>
</controller>
<controller type="virtio-serial" index="0">
<address type="pci" domain="0x0000" bus="0x03" slot
   ="0x00" function="0x0"/>
</controller>
<interface type="network">
<mac address="52:54:00:2c:44:5a"/>
```

```
<source network="default"/>
<model type="e1000e"/>
<address type="pci" domain="0x0000" bus="0x01" slot
   ="0x00" function="0x0"/>
</interface>
<serial type="pty">
<target type="isa-serial" port="0">
        <model name="isa-serial"/>
</target>
</serial>
<console type="pty">
<target type="serial" port="0"/>
</console>
<channel type="spicevmc">
<target type="virtio" name="com.redhat.spice.0"/>
<address type="virtio-serial" controller="0" bus="0"
    port="1"/>
</channel>
<channel type="spiceport">
<source channel="org.spice-space.webdav.0"/>
<target type="virtio" name="org.spice-space.webdav
   .0"/>
<address type="virtio-serial" controller="0" bus="0"
    port="2"/>
</channel>
<input type="mouse" bus="ps2"/>
<input type="keyboard" bus="ps2"/>
<tpm model="tpm-tis">
<backend type="emulator" version="2.0"/>
</tpm>
<graphics type="spice" autoport="yes">
<listen type="address"/>
<image compression="off"/>
</graphics>
<sound model="ich9">
<audio id="1"/>
<address type="pci" domain="0x0000" bus="0x00" slot
   ="0x1b" function="0x0"/>
</sound>
<audio id="1" type="spice"/>
<video>
<model type="qxl" ram="65536" vram="65536" vgamem
   ="16384" heads="1" primary="yes"/>
<address type="pci" domain="0x0000" bus="0x00" slot
   ="0x01" function="0x0"/>
</video>
<redirdev bus="usb" type="spicevmc">
<address type="usb" bus="0" port="2"/>
</redirdev>
<redirdev bus="usb" type="spicevmc">
<address type="usb" bus="0" port="3"/>
</redirdev>
```

```
        <memballoon model="none"/>
</devices>
<qemu:commandline>
        <qemu:arg value="-s"/>
</qemu:commandline>
</domain>
```