NTNU
Norwegian University of
Science and Technology

Live forensics on the Windows 10 secure kernel.

Hans Kristian Brendmo

Master in Information Security
Submission date: May 2017
Supervisor: Stephen Wolthusen, IIK

Norwegian University of Science and Technology
Department of Information Security and Communication

Live Forensics on the Windows 10 secure kernel.

Hans Kristian Brendmo

Master's Thesis
Master of Applied Computer Science
30 ECTS
Department of Computer Science and Media Technology
Gjøvik University College,

Avdeling for
informatikk og medieteknikk
Høgskolen i Gjøvik
Postboks 191
2802 Gjøvik

Department of Computer Science
and Media Technology
Gjøvik University College
Box 191
N-2802 Gjøvik
Norway

Live Forensics on the Windows 10 secure kernel.

Hans Kristian Brendmo

2017/06/15

Abstract

The field of Digital Forensics is always changing together with developments in computer hardware and operating systems. In June of 2015 Windows 10 was released to consumers offering numerous changes to the operating system. One of the most interesting features from the perspective of Digital Forensics is a feature known as Device Guard. Device Guard is said to offer protection from advanced malware such as rootkits, polymorphic viruses and even zero day exploits. Device Guard accomplishes this by introducing VBS based security which introduces a new secure kernel that is separate from the normal kernel. This thesis takes a look at the internals of the secure kernel and the trustlets running within. This thesis also discusses different methods on how to best acquire a full live memory dump as well as making a program that can analyze it.

Preface

Ever since discovering Mark Russinovich' blog in 2005 I have always been interested in learning more about Windows and how it worked internally. The idea that it was possible for an outsider to understand how a complex proprietary system worked in great detail inspired me to learn reverse engineering. I would like to thank Stephen Wolthusen for allowing me to take on this master thesis on a subject that has interested me for years.

Contents

List of Figures

1 Introduction and structure.

This thesis takes a look at the newly developed Windows 10 Isolated User Mode (IUM) as well as the associated secure kernel. In order to realize IUM several software and hardware requirements must be met. The background chapter gives an explanation of how all the necessary functionality works, as well as presenting some of the software that runs in IUM. As this thesis focuses on forensics the next chapter will discuss the aspects of memory forensics relevant to the secure kernel. It will also discuss different methods on how to obtain full memory dumps from the secure kernel and IUM.

After obtaining the memory dump the next chapter takes a look at secure kernel internals. This knowledge is obtained by investigating the memory dumps as well as reverse engineering program binaries. The main focus is on obtaining information on potentially useful forensic artifacts found in secure kernel memory. To accomplish all this, a set of tools called the Secure Kernel Forensic Toolkit was built. The last section gives a brief description on how this tool works as well as providing a overview of the available commands.

Since reverse engineering has been a important aspect in obtaining information, a final appendix is provided to demonstrate how this was accomplished.

For actual testing a preview version of Windows 10 Enterprise was used, the kernel build number was the following:

```
Windows 10 Kernel Version 10586 MP (4 procs) Free x64
Product: WinNt, suite: TerminalServer SingleUserTS
Built by: 10586.162.amd64fre.th2_release_sec.160223-1728
```

2 Background

2.1 Isolated User Mode

Isolated User Mode (IUM) is a security feature introduced in Windows 10 Enterprise which enables the execution of special applications in a context isolated from regular applications. To achieve this isolation IUM relies on a secure kernel which handles basic operating system tasks such as scheduling and memory management. As the secure kernel will only handle the mere basics it will run in parallel with the regular Windows kernel which handles the main operation of the machine. The main advantage of using a separate kernel to handle IUM is that the secure kernel does not load any third party modules. This enables the secure kernel to operate without the risk of interference from third party code, a problem still affecting the normal kernel. Ensuring that the kernel itself is free from interference is a important first step in ensuring secure user mode processes.

To achieve this isolation the secure kernel relies on the Windows hypervisor to manage both the normal and secure kernel. This means that the hypervisor can set different memory permissions for both kernels, making it impossible for the normal kernel to access memory belonging to the secure kernel.

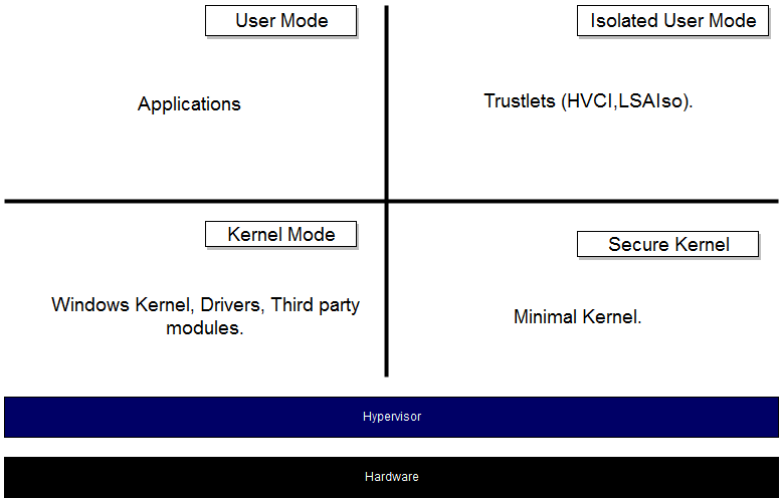

Figure 1: A diagram showing the high level architecture for a system where IUM has been activated.

The applications that run in IUM are known as trustlets and there are currently only a few trustlets implemented by Microsoft The reason for wanting these applications to run in a context that is isolated from the rest of the operating system is mainly to prevent access from malicious parties. This means that it is easier to keep secrets hidden such as in the case of credential guard which stores user credentials which is important to keep hidden. In previous versions of Windows a user with administrative privileges could extract these credentials in plaintext making it harder to keep this information confidential. Another benefit of process isolation is that it would make tampering more difficult. Tampering is particularly egregious for components that implement security functionality since they have no way of guaranteeing that they will function properly. This is the main motivation with moving code integrity verification into the secure kernel.

2.1.1 How IUM is realized

Most PC's today use a processor that is built on the x86 architecture. Since the Intel 80286 processor was released in 1982, all x86 processors have supported a operational mode known as protected mode. This introduced several new features, one of which was the idea of privilege levels. Although the x86 architecture supports 4 privilege levels, known as rings, Windows only uses two rings. The reason why Windows uses only two rings is that originally Windows NT was designed to run on the Intel i860 RISC processor[1]. This processor used only two protection levels where the user level was intended for regular applications, and the supervisor level intended for the operating system [2].

The kernel, along with third-party modules, occupy the most privileged ring at ring 0, while regular applications run in ring 3. Instead of referring to them as rings, Microsoft most commonly uses the terminology kernel mode for executing in ring 0 and user mode for ring 3. As mentioned before it would be beneficial for the operating system to operate in a mode where no third-party modules can execute. To achieve this the operating system tries to isolate processes further by using a hypervisor. This hypervisor can be informally thought to be running in "ring -1" since it is considered more privileged than the kernel itself.

Windows own hypervisor, Hyper-V, was introduced in Windows Vista and was originally intended to take advantage of new viritualization features in the x86 architecture introduced to increase the performance of virtual machines. The most prolific x86 vendors have their own implementation of this viritualization technology known as Intel VT-x and AMD-V. Ultimately the job of the hypervisor in the context of IUM is to control which parts of the operating system that can access given parts of physical memory. Utilizing the hypervisor to provide process isolation is referred to Virtualization Based Security (VBS) in Windows operating systems.

To fully achieve memory isolation another piece of viritualization technology known as the Second Layer Address Translation (SLAT) is used. SLAT technology was originally intended to speed up translation of virtual addresses on virtual machines as these originally had to be translated twice (once on the guest machine and once on the host machine) before resolving the actual physical address. SLAT implementations on the x86 architecture is known as Intel EPT or AMD RVI depending on the make of the processor. SLAT provides its own set of page tables allowing the hypervisor to set different access masks for the secure kernel and the "normal" kernel. This means that the secure kernel can section off parts of memory that the normal kernel can not access.

As an additional protection mechanism, IUM will also take advantage of the IOMMU on systems where this is present. An IOMMU is a piece of hardware that sits on any I/O bus capable of DMA. The role of the IOMMU is to translate the device addresses of the devices on the bus into physical addresses in main memory. This means that the IOMMU have its own set of page tables to map these addresses, which again means that the IOMMU can choose to restrict DMA-access to any device with insufficient access. This means that IOMMU's can be used to prevent so called DMA-attacks as the devices would be unable to access physical memory that has not been allocated by the device itself. The actual implementations of IOMMU's are known as VT-d for Intel processors and AMD I/O Virtualization Technology[3] for AMD processors.

2.2 Device Guard

Device Guard is a set of services offered in the enterprise versions of Windows 10 [4]. The overall goal of Device Guard is to ensure that only code that meets the requirements of the system policies will be executed. The main reason behind only executing code defined in the system policies is to provide protection against malicious or unwanted code. Since Device Guard will deny any code not described in the given policy this also ensures protection against future malware not yet known. This also gives protection against malware that might use polymorphism or encryption to escape detection from signature based anti-malware services. Previous versions of Windows had some of this functionality already in place to provide basic verification of kernel mode code. This meant verifying the main operating system files, kernel mode drivers and some user mode verification [5]. Device guard improves on this verification process by moving code verification into the secure kernel. This ensures that no third-party kernel drivers can influence how code verification is performed.

One of the features that Device Guards adds is Hypervisor based code integrity (HVCI), allowing the secure kernel to perform code integrity checks. HVCI offers improved KMCI and adds full UMCI. The main benefit of performing integrity checks in the secure kernel is that the secure kernel should in theory not load any third party extensions, meaning that the kernel is less easily compromised. Remember that in order for the secure kernel to run Windows uses the hypervisor as the most privileged component, giving full access to system memory.

In addition to HVCI Device Guard offers full user mode integrity checks (UMCI). This allows the administrator complete control of any code that is executed in user mode, performing the same checks as done in kernel mode. In addition to allowing for greater control of code integrity, Device Guard will also use new hardware enhancements to ensure that memory based attacks are more difficult to perform.

2.2.1 Requirements

Device guard VBS has a set of hardware and software requirements that must be met before being enabled. Note that this is the requirements for Device Guard VBS, enforcing the code integrity is still possible although with no protection from various memory based attacks.

Requirement	**Classification**
UEFI firmware at least version 2.3.1	Absolute
Virtualization extensions	Absolute
x64 Architecture	Absolute
A VT-d or AMD-Vi IOMMU	Recommended
Secure Boot enabled	Absolute

Table 1: Overview of System Requirements for Device Guard. The recommended specifications are not necessary to run Device Guard VBS, although necessary to offer full protection. The absolute requirements must be met in order for Device Guard services to work.

UEFI Firmware at least version 2.3.1

To enable Secure Boot the system motherboard must boot using UEFI firmware system version 2.3.1. This means that legacy BIOS systems are not supported.

Virtualization extensions

The system must support hardware virtualization extensions such as Intel VT-x or AMD V technology. The system must also support SLAT technology to enable memory protection. These requirements are in place because Device Guard VBS requires the Hyper-V hypervisor to be active in order to work. Since a hypervisor is required for VBS to work, one can not run Device Guard on a virtual machine directly, because then a hypervisor would already be running. The only exception is on systems that support nested virtualization where more than one hypervisor can be operational.

x64 Architecture

Device Guard is only available on 64-bit systems.

VT-d or AMD-Vi

The IOMMU is a piece of hardware that allows for controlling the DMA access of DMA enabled devices. On systems without an IOMMU, DMA capable devices would have full access to system memory. The IOMMU adds memory management for DMA devices, opening up support for memory protection. This essentially means that a DMA device usually only access memory that it allocated for itself. On systems having a IOMMU one can enable DMA protection which gives protection against DMA-attacks. DMA-attacks relies on using a DMA capable device to access important memory structures. On Intel systems the IOMMU technology is known as VT-d, while on AMD systems it is known as AMD-Vi.

Secure Boot

Secure Boot must be enabled for Device Guard VBS to work. During testing disabling Secure Boot would disable VBS.

2.2.2 Function

Basic functionality to maintain code integrity has been present since Windows Vista. This part of code integrity was aimed at ensuring that code that is executed in kernel mode met certain standards. Before starting a driver Windows would ensure that the driver first had a valid digital certificate before allowing it to run. When the driver had been executed, a cryptographic hash would be created for each executable page in memory so that the integrity of the memory could be verified. Code integrity in earlier Windows versions was implemented in a system dll, CI.dll that would launch as a mandatory kernel mode module. If CI.dll is not present Windows will simply refuse to start.

Similarly, Device Guard is implemented in a kernel mode module known as SKCI.dll and is a mandatory part of the kernel. The main difference is that SKCI is loaded as part of the secure kernel and not the normal kernel. Since the secure kernel cannot perform file I/O it has to rely on the normal kernel to read the file for verification to begin. Although the normal kernel is needed to read the file, one cannot simply bypass code integrity by hijacking the normal kernel as it is ultimately the secure kernel which controls which part of memory that is executable. Code integrity checks also includes ensuring that process memory does not change based on hash signatures.

2.2.3 Activation and Usage

When enabling Device Guard one can choose between using it only for ensuring code integrity in kernel mode (KMCI), or also using it for ensuring user mode code integrity (UMCI).

Enabling VBS enforced KMCI

KMCI has been present since Windows Vista on 64-bit systems albeit with less features. The initial implementation forced all kernel mode modules to be digitally signed as to improve the stability and reduce the presence of kernel-mode malware, although this would not eliminate malware completely. Device Guard improves on the KMCI functionality by introducing VBS, enabling the code verification process to take place in the secure kernel.

To enable VBS one has to first make sure that both the hypervisor feature and the Isolated User Mode feature is active on the system. These features are enabled using the control-panel. After

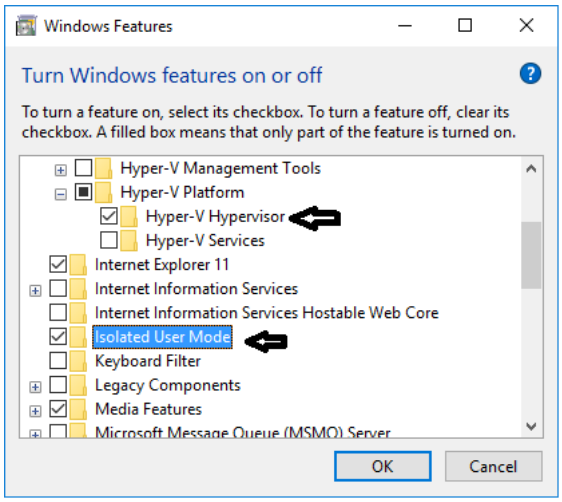

Figure 2: Make sure to enable IUM and the Hypervisor before enabling VBS.

enabling these features the system should be restarted. One can now enable VBS. The simplest way of enabling VBS on a local machine is by using the group policy. Using the group policy mmc snap-in one can navigate to the Device Guard settings and then selecting "Turn on virtualization based security". To enable VBS enforced KMCI make sure to check "Enable Virtualization Based Protection of Code Integrity". If the system is equipped with IOMMU technology the Secure Boot with DMA protection can be enabled for defense against DMA-attacks. If no IOMMU is present the Secure Boot setting can be selected. In order for the changes to take effect the system should be restarted once again.

2.2.4 Device guard UMCI

In addition to providing improved verification of kernel mode modules, Device Guard also introduces User Mode Code Integrity (UMCI) that ensures verification of user mode code. This functionality was introduced to provide a better defense against zero-day exploits, as well as allowing for better control over which code that can be executed on any given system.

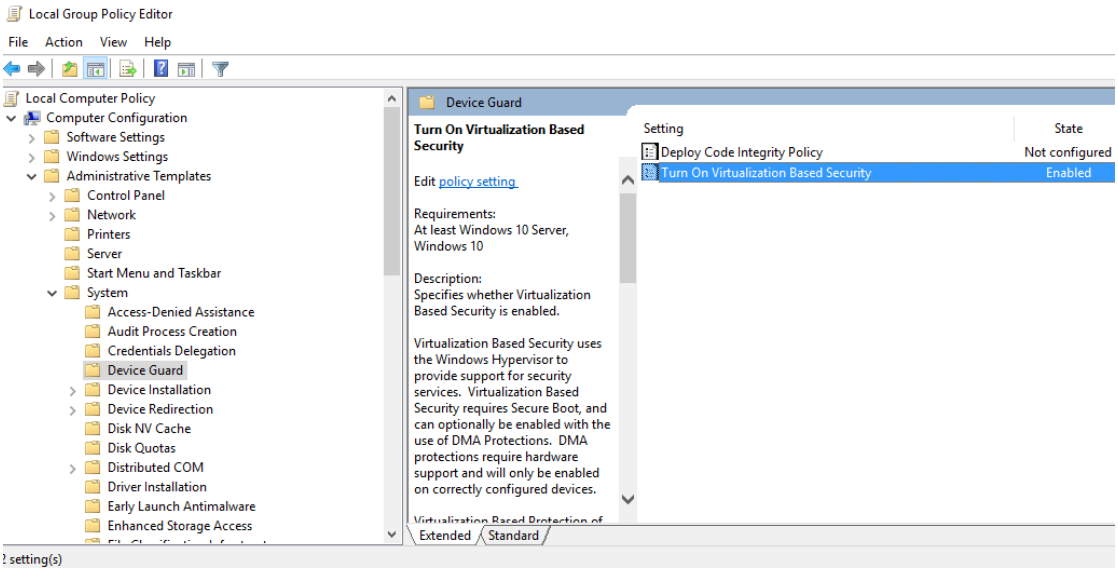

Figure 3: Using gpedit.msc one can enable VBS.

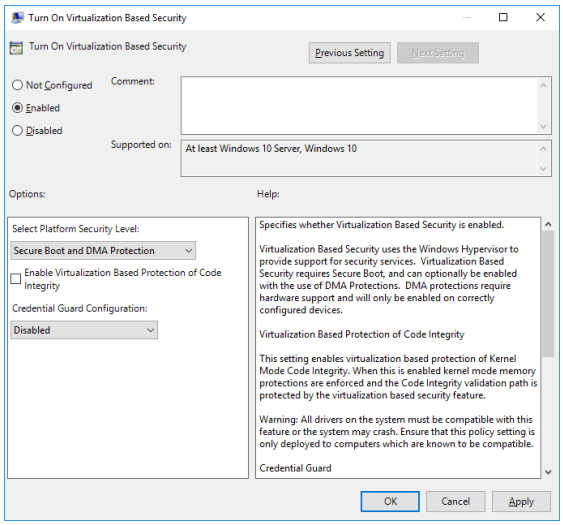

Figure 4: VBS GPO object.

UMCI allows a system administrator to establish a Code Integrity Policy (CIP) that will specify which code is allowed to execute. Any code that does not meet the requirements specified in the policy is not allowed to run unless the system runs in audit mode. In kernel mode all modules must be digitally signed when secure boot is enabled and so verifying images in kernel mode is done by verifying that the digital certificate is correct.

For regular user mode applications however, it is much more common that binaries are not digitally signed, meaning that the CIP is not sufficient in this case. To address this problem, Windows allows the administrator to create catalog files for unsigned binaries that contain the information necessary to verify the binary. Together the CIP and the optional catalog files contain all the information necessary to enforce UMCI.

Creating a CIP

When creating the CIP the system administrator can enable or disable numerous features. The table below describes the most notable options:

Option	Description
0 - UMCI	Enables the UMCI feature
3 - Audit Mode	Instead of enforcing the policy, violations are logged and the code is allowed to execute. This option is used for testing purposes allowing the administrator to see which code violates the policy without enforcing it. Enabled by default.
6 - Unsigned System Integrity Policy	Allows the policy file itself to be loaded unsigned. When this option is disabled the policy need to include a list of UpdatePolicySigners to allow for further modification. Enabled by default.
10 - Boot Audit on Failure	If any system drivers fails during boot the policy is reverted to audit mode to allow for troubleshooting. Enabled by default.

To specify which files that can execute, the CIP allows for specifying different requirements that must be fulfilled before UMCI will allow the file to execute. These requirements or rules provides different levels of security based on how strict the system administrator wishes to be. Strict policies such as cryptographic hashes offer great security, but are more difficult to manage since the policy must be updated each time the hash changes. Some of the most notable rules are:

Rule level	Description
Hash	Stores the cryptographic hash for every binary that is allowed to execute.
FileName	Stores the name of every binary that is allowed to execute. This is significantly less secure, but easier to manage than using cryptographic hashes.
PcaCertificate	The primary certificate authority (PCA) is the highest available CA in the certificate chain. Typically this is the CA right below the root.
LeafCertificate	Verifies the leaf certificate in the certificate chain. Leaf certificates are more specific, but has a shorter validity than CA certificates.
Publisher	Verifies the PcaCertificate and the common name (CN) for the leaf certificate.

There are also additional rules pertaining to binaries signed by the Windows Hardware Quality Labs (WHQL), only allowing execution of binaries that meet this requirement. What can be seen is that most rules require the binary to be digitally signed, where only the hash and filename rules applies to unsigned binaries.

Instead of manually creating rules for each individual file, Microsoft recommends that one starts with a clean machine and use it as a basis for the CIP. The **New-CIPolicy powershell** command will scan the machine and create a new CIP based on the currently available binaries. Unless otherwise specified, this CIP will run in audit mode allowing the administrator to log any breaches to the CIP and review them. **New-CIPolicy** will output a xml file which can be reviewed to make sure that the generated policy is correct. Once the administrator is happy with the rules in the xml file, one can then convert the xml file to a binary format which is the actual format used by Device Guard. To convert the xml file one uses the **ConvertFrom-CIPolicy** powershell command. Once any errors have been dealt with, a new policy can be created from the audit logs by using the -Audit switch when using **New-CIPolicy**. One can also merge existing policies using the **Merge-CIPolicy** powershell command, allowing for easy unification of the audit policy with the existing one.

Creating a Catalog file

If the system administrator have any unsigned binaries that are necessary to run under UMCI - catalog files are needed. To create catalog files Microsoft has provided the Package Insepctor tool. The tool works by monitoring modifications made by applications to the local drive. To monitor drive C the Package Inspector can be started by using the following command:

```
PackageInspector.exe Start c:
```

Once the package inspector is running one should install and run the application to the local drive which is currently beeing monitored. Once this procedure has been repeated by all unsigned applications one can stop package inspector by using the stop command:

```
PackageInspector.exe Stop C: –Name $CatFileName –cdfpath $CatDefName
```

This will output two files, a catalog file (cat) and a catalog definition (cdf) file. The system administrator should then sign the catalog file and deploy it using either group policy or the System Center Configuration manager. After signing the catalog file one must also remember to add the signing certificate to the CIP.

2.3 Credential Guard

Credential Guard is a trustlet that was introduced to prevent certain attacks against the Local Security Authority Subsystem Service (LSASS). A user that had access to the memory of the lsass process could dump the credentials or manipulate lsass memory in a so called pass-the-hash attack . This attack can be easily performed using public tools such as Mimikatz or Metasploit[6]. Credential Guard introduces a new process LsaIso which stores the hash and credential data in IUM to prevent access from regular applications. Lsass runs in parallel with LsaIso providing the main cryptographic protocol services, the main difference is that cleartext information is moved into LsaIso. Although isolated, LsaIso and Lsass can still communicate using RPC as needed.

Once credential guard is enabled the use of certain authentication protocols will also change. This means that applications that use NTLMv1, MS-CHAPv2, Digest, or CredSSP cannot use the sign in credentials for authentication. Applications that use NTLMv1 will no longer work once Credential Guard is active. For systems that use Kerberos, further restrictions apply. This means that unsafe Kerberos implementations, such as using only DES encryption, is no longer allowed. Note that Credential Guard only supports the modern NTLM protocols as well as the Kerberos protocol. It will not protect credentials provided using different protocols [7].

2.4 Hardware security mechanisms

To fully realize IUM the system has to rely on several hardware mechanisms for system security. This section takes a deeper look at how these mechanisms work and what security enhancements they offer. For simplicity only Intel architecture is considered, although AMD has similar protection mechanisms on their architecture.

2.4.1 Intel Vt-x

To realize hardware virtualization Intel introduced a set of new processor instructions known as the VMX set. This instruction set introduces commands to control virtualization such as entering and exiting virtualization mode. To control the execution of virtual machines Intel allows the operating system to register a piece of software called the Virtual-machine monitor (VMM) [8]. The VMM on Windows systems is also known as the hypervisor. To isolate the VMM from the rest of the operating system virtualization mode features two modes of operation; non-root and root. Generally the VMM executes in root mode, while guest operating systems operate in non-root mode. Transitioning from root to non-root is known as a VM entry, while transitioning from non-root to root is known as a VM exit. The difference between these modes of operation is that the VMM will have full access to the VMX instructions while certain events and instructions called by the guest automatically returns control to the VMM through an VM exit. Instructions that will always cause an VM exit are the VMX instructions and special instructions such as CPUID and XSETBV.

To begin VMX operation, VMX is first enabled by enabling the VMXE bit (bit 13) in the CR4 register on the processor. Once the VMXE bit is set to 1 the system can enter VMX operation by executing VMXON. After this instruction has been performed, the system can no longer change the VMXE bit until VMX operation is turned off.

Virtual Machine Control Structures

To store the data needed to manage VMX operation at least one virtual machine control structure (VMCS) is created and stored in a special memory region known as the VMCS region. To handle multiple virtual machines the processor can create several VMCSs at a time, but only one of them is considered current at any given time. The information in any given VMCS can be divided into six main areas:

- **The Guest-state Area**
  Stores information relating to the state of the guest which is updated each time the guest performs an VM exit. This allows the processor to easily resume the state upon VM entry.
- **The Host-state area**
  Stores state information used to initialize the host upon VM exits.
- **VM-execution control**
  Controls which events and instructions causes an VM exit in non-root mode. This allows the VMM to set more restrictions on what the guest can do.
- **VM-exit control**
  Stores information related to controlling VM exits.
- **VM-entry control**
  Stores information related to controlling VM entries.
- **VM-exit information**
  Before performing an VM exit the processor will store information related to the cause of the VM exit in this area. This information is used to determine what the VMM should do after the VM exit has been performed.

Intel EPT

The Intel extended page-table (EPT) mechanism is Intels version of SLAT. Once EPT is enabled by setting the VM-exectuion control to 1, the guest-physical addresses are translated using the special EPT tables instead of performing "traditional" address translation. Just like regular virtual address space EPT also allows for setting specific privileges of every page, meaning that one can control which pages the guest can access. Attempting to access restricted memory will result in an EPT violation causing an VM exit. On a Windows system that uses the secure kernel the EPT paging structures is used to ensure that the normal kernel cannot access restricted pages belonging to the secure kernel.

Intel VT-d

Intel Virtualization Technology for Directed I/O (VT-d) is a set of technology components that allows for greater control on how a guest communicates with physical hardware. On the earliest virtual machines the host would emulate all the hardware accessible to the guest which meant some performance loss. When running a VM Intel VT-d introduces the following capabilities:

- **I/O device assignment**
  This allows the VMM to directly assign a physical I/O device to a VM. This means that the driver for the device runs directly in the VM and allows for direct commnication with the device.
- **DMA remapping**
  Allows for address translation on DMA capable devices.
- **Interrupt remapping**
  Adds support for isolation and routing of interrupts from different devices and interrupt controllers to given VMs.
- **Interrupt posting**
  Supports direct delivery of virtual interrupts to virtual processors.
- **Reliability**
  Allows for logging and reporting DMA and interrupt errors to software.

The DMA remapping feature is the component that allows for protecting Windows device guard against DMA-attacks by isolating DMA-capable hardware [9]. This isolation works by first assigning every part of physical memory into one or more domains. Then every DMA-capable device is assigned to at least one domain that it can access. If the device tries to access a domain that it has not been assigned to, this access is denied.

3 Memory Forensics on the Secure Kernel

3.1 Memory acquisition on IUM activated systems

One of the main challenges associated with memory forensics on IUM systems is how to capture a complete memory dump of the system. This section will go through various methods for live acquisition of memory and the advantages and disadvantages of these methods.

3.1.1 Software acquisition

Software acquisition means running software that will attempt to copy the live contents of the main memory onto a file on disk. This process is known as a memory dump and the resulting file is called a memory dump file. The idea is that the memory dump file should represent a perfect snapshot of the state of the computer at the time of creation. In order to accomplish this the software has to solve two main problems, it needs sufficient access and it needs to suspend the system so that the state represents a uniform point in time.

Software that can accomplish these tasks require administrative access and will dump the system memory onto a file that can later be analyzed with the appropriate software. For Windows there are several tools that can accomplish this such as Memoryze, Mdd, DumpIt and FTK Imager. The main advantage of this method is that is well described and should be quite reliable depending upon the software. One the main disadvantages is that running any kind of software on a live system will alter the state of the system which is important for the forensic analyst to consider.

Regarding IUM the main problem with software acquisition is that the software can only be granted normal kernel mode access and as such can not necessarily access the secure kernel and IUM memory. This means that one can no longer trust traditional memory acquisition to provide full dumps of system memory.

3.1.2 Hardware acquisition

Hardware acquisition typically rely on a method known as a DMA-attack which means using hardware that has DMA access to dump the system memory [10]. The advantage of using hardware to dump memory is that there is no reliance on operating system API-calls. Hardware acquisition can be performed either by using DMA capable systems such as firewire, or using dedicated hardware such as CaptureGuard [1].

The problem with this method in regards to IUM is that the strictest configuration of IUM supports IOMMU protections which means that hardware devices no longer have full access to system memory. IOMMU's ensure that devices can only access memory that they themselves have allocated. This means that hardware acquisition is not sufficient for complete memory dumps.

[1] https://www.bluerisc.com/captureguard/

3.1.3 Cold boot attacks

In 2008 a group of researchers at Princeton showed that by cooling the RAM modules they could recover the data even after the system was shut down [11]. This means that the system can be turned off, the modules extracted and then inserted into another system while still having a chance of retaining state. The advantage of this could be that the memory modules could be inserted into a system without IOMMUs and so the memory could be extracted with full memory access. The disadvantage of this method is that it appears very unreliable meaning that it will work well for some systems while not being able to recover any data from other systems [12]. There are also several variables such as the temperature, the way the system was cooled, the type of ram modules and how long the system was turned off that makes it difficult to asses the forensic soundness of memory capturing using this method.

3.1.4 System management mode

System management mode (SMM) is a special processor operating mode introduced in the 386SL line of processors[13]. The main purpose of SMM is to perform management functions such as handling power management, system errors and TPM functionality. To do this SMM operates at a high privilege operating mode which has full access to system memory - even hypervisor memory. If one could execute code at this level this could be quite beneficial for memory forensics as it will allow blocking all other processor operations, meaning that one could ensure that the system does not change state during the capturing of the memory dump. Another benefit is the previously mentioned high privilege level which has full access to system memory. One drawback of SMM however, is that by default it can only access the lowest 4gb of memory. This problem can supposedly be solved by enabling physical address extensions (PAE) which can address up to 64gb of memory [14].

The main problem with SMM is that the system must have a working system management handler installed on the system BIOS in order to execute specific functionality. Since systems running IUM must have secure boot enabled it means that changing the system BIOS is not trivial and this makes using SMM to dump memory currently unfeasible.

3.1.5 Nested virtualization

Nested virtualization means running a hypervisor inside one or more hypervisors. The benefit of this is that if hypervisor x runs inside hypervisor y, then from the context of hypervisor y one would have complete memory access to hypervisor x. Nested virtualization is supported on newer builds of Windows 10 and opens up for enabling IUM inside a virtual machine using Hyper-V [15]. Although other virtualization packages support nested virtualization, only Hyper-V currently supports secure boot in conjunction with nested virtualization. Keep in mind that Secure Boot is required to fully enable all of Device Guards functionality.

Using nested virtualization is also an option for physical machines running IUM if one can manage to break secure boot and load another hypervisor before the Hyper-V hypervisor starts. Although this was the same problem as with SMM, loading another hypervisor does not require installing a SM-handler as well as triggering an SMI.

3.2 The usefulness of full memory dumps

Since obtaining memory dumps that contain IUM and secure kernel memory is more technically challenging, it is worth considering the benefits of acquiring a full memory dump in regards to forensic investigations. Dumping memory is one of the techniques of live forensics which could in and itself be considered a less traditional forensic technique. Most digital forensic books focus more on obtaining forensic artifacts from a disk rather than in depth analysis of live memory. Live forensics are considered most useful in specific areas such as incident response and malware analysis [16].

So when considering these two scenarios it is important to think about what information is contained in IUM and the secure kernel that one excepts to be helpful to the investigation. Although IUM should only contain Microsoft processes, it has been demonstrated that it is possible to load third party applications in IUM [17]. If it is suspected that malware or other suspicious processes are running in IUM then one might need to consider obtaining a full memory dump.

Another concern is if there is a suspicion that encrypted information found elsewhere might be decrypted using information in the memory. Previously one could recover the credentials of the user by dumping lsass memory to perform analysis. With credential guard the credentials have been isolated to LsaIso so theoretically one might be able to recover the credentials given full access to LsaIso memory. As mentioned before, live forensics is still not part of the "standard" digital forensics procedure as often the disk would supply enough information or a live system is not available.

3.3 Memory Descriptor Lists and Driver-locked memory

A fundamental aspect of the virtual memory model is that the translation of virtual addresses will depend upon process context. This creates a problem if a process wishes to allocate memory to contain a I/O buffer accessible from kernel mode. Since the driver responsible for filling the I/O buffer can execute at any context, this means that the virtual address used by the process might be pointing to a different physical address from the one used to fill the buffer. To solve this problem Windows uses structures known as Memory Descriptor Lists (MDLs). The MDL structure is defined in wdm.h as:

```
struct _MDL {

struct _MDL *Next;
CSHORT Size;
CSHORT MdlFlags;
struct _EPROCESS *Process; //Process context for the MDL.
PVOID MappedSystemVa; // Kernel address
PVOID StartVa; // User mode address (if applicable), else equal to MappedSystemVa
ULONG ByteCount; // Total size of the allocation.
ULONG ByteOffset; // Offset into the first page of the allocation.
} MDL, *PMDL;
```

The MDL structure allows for locking physical memory to allow for consistent access regardless of context. This allows the kernel module to write directly to physical memory while also sup-

plying a valid user mode address for the relevant process [18].

The actual size of the MDL structure is 48 bytes, and following the MDL is a list of page frame numbers (PFNs) of the physical pages described by the MDL. The reason MDLs are important in the context of virtual machines is that they are used to describe driverlocked memory. Driver-locked memory is used to guarantee that the physical memory used by the virtual machine is not paged out by the memory manager on the host machine. During virtual machine creation the host machine will allocate MDLs using *IoAllocateMdl*, and locks the pages using *MmProbeAndLock* meaning that data is not paged out before *MmUnlockPages* is called. One can see how many pages that is locked in memory at any time by looking at the output produced by the **!vm** extension in WinDbg:

```
Available Pages:          600868 (     2403472 Kb)
ResAvail Pages:           853568 (     3414272 Kb)
Locked IO Pages:               0 (           0 Kb)
Free System PTEs:     4294984542 (17179938168 Kb)
Modified Pages:            15745 (       62980 Kb)
Modified PF Pages:         15736 (       62944 Kb)
Modified No Write Pages:       0 (           0 Kb)
NonPagedPool Usage:          212 (         848 Kb)
NonPagedPoolNx Usage:      20853 (       83412 Kb)
NonPagedPool Max:     4294967296 (17179869184 Kb)
PagedPool 0 Usage:         29958 (      119832 Kb)
PagedPool 1 Usage:         16609 (       66436 Kb)
PagedPool 2 Usage:           263 (        1052 Kb)
PagedPool 3 Usage:           259 (        1036 Kb)
PagedPool 4 Usage:           315 (        1260 Kb)
PagedPool Usage:           47404 (      189616 Kb)
PagedPool Maximum:    4160749568 (16642998272 Kb)
Session Commit:             3501 (       14004 Kb)
Shared Commit:             28186 (      112744 Kb)
Special Pool:                  0 (           0 Kb)
Shared Process:             8096 (       32384 Kb)
Pages For MDLs:          1074101 (     4296404 Kb)
Pages For AWE:                 0 (           0 Kb)
NonPagedPool Commit:       21632 (       86528 Kb)
PagedPool Commit:          47404 (      189616 Kb)
Driver Commit:             12014 (       48056 Kb)
Boot Commit:               25653 (      102612 Kb)
System PageTables:           418 (        1672 Kb)
VAD/PageTable Bitmaps:      4502 (       18008 Kb)
ProcessLockedFilePages:      253 (        1012 Kb)
Pagefile Hash Pages:           0 (           0 Kb)
Sum System Commit:       1225760 (     4903040 Kb)
Total Private:            241513 (      966052 Kb)
Misc/Transient Commit:      1681 (        6724 Kb)
Committed pages:         1468954 (     5875816 Kb)
Commit limit:            4288847 (    17155388 Kb)
```

The section denoted "Pages for MDLs" shows driver-locked memory and in this example one can see that 1074101 physical pages are currently locked. This output is from a system having 8GB of system memory where 4GB is reserved for running a virtual machine. One can see that the physical page count for MDLs is a little over 4GB which is what you would expect running a virtual machine which commits 4GB of memory. Although Windows can use driver-locked memory

for other purposes, usually high use is due to running virtual machines.

Another tool that can be used to display driverlocked memory usage is RAMMAP. RAMMAP is a tool that divides physical memory usage into several section, having a separate section for driverlocked memory. The main problem with these tools is that neither tool can tell you which module is responsible for locking memory. Knowing which module that is responsible for locking the memory could would make it easier to obtain some idea of what the driverlocked memory is used for. It is for example reasonable to assume that memory locked by the virtual machine driver is used for purposes related to handling virtual machines. In this case one is interested in driverlocked memory that is used to provide physical memory for the virtual machine running on the system.

When studying driverlocked memory for this project, a somewhat obscure function was discovered that appeared to be tracking per process locked page use.
By setting the DWORD key found at **\HKLM\SYSTEM\CurrentControlSet\Control\Session Manager \Memory Management\TrackLockedPages** to 1 this enables more extensive tracking of locked memory on a per process basis [19].

Theoretically one should then be able to use the **!lockedpages** WinDbg extension to reveal this information. Upon using this command it became apparent that this functionality was no longer working:

```
0: kd> !lockedpages ffffe001b547e840
Process: ffffe001b547e840
35 locked pages...
Unable to get BalancedRoot.RightChild at ffffe001b43810d0
```

Although the function is no longer working, it appears to still be able to access some information related to the number of locked pages. To see if the rest of this information could be reconstructed manually, the first step is to figure out where it is stored. Since the locked pages tracker works by tracking information on a per-process level it is perhaps natural that a pointer to some relevant structure might be found in the EPROCESS block. At offset 0x4b0 the following promising member can be found:

```
+0x4b0 LockedPagesList  : 0xffffe001`b43810d0 Void
```

Dumping the contents of the pointer reveals the following:

```
0: kd> dq 0xffffe001`b43810d0
ffffe001`b43810d0  ffffe001`b51203a0 00000000`00000023
ffffe001`b43810e0  00000000`00000000 00000000`00000001
ffffe001`b43810f0  6e496d4d`021c0003 00200074`0065006e
ffffe001`b4381100  ffffe001`b43fc010 ffffe001`b44b53f8
ffffe001`b4381110  ffffe001`b4381110 ffffe001`b4381110
ffffe001`b4381120  00000000`00060000 ffffe001`b4381128
ffffe001`b4381130  ffffe001`b4381128 00000000`00060000
ffffe001`b4381140  ffffe001`b4381140 ffffe001`b4381140
```

At address ffffe001`b43810d8 one finds the hexadecimal number 0x23 which corresponds to the number of locked pages (35) given by the !lockedpages command. Verifying that this holds

true for all processes, the LockedPagesList appears to be the place where the information on locked pages is found. Locating a pointer to some structure is a good starting point for further investigation, now one need to understand which members are part of this structure. Fortunately Microsoft provides symbol information for several structures, and so a decent start is to search for structures having the keyword "LOCK" somewhere in it's name.

```
0: kd> dt nt!*LOCK*
          ntkrnlmp!_KSPIN_LOCK_QUEUE
          ntkrnlmp!_IO_STATUS_BLOCK
          ntkrnlmp!_IO_STATUS_BLOCK
          ntkrnlmp!_WHEAP_INFO_BLOCK
          ntkrnlmp!_MI_CONTROL_AREA_WAIT_BLOCK
          ntkrnlmp!_OB_HANDLE_REVOCATION_BLOCK
          ntkrnlmp!IRPLOCK
          ntkrnlmp!_KWAIT_BLOCK
          ntkrnlmp!_CM_NOTIFY_BLOCK
          ntkrnlmp!_MI_VAD_EVENT_BLOCK
          ntkrnlmp!_CM_KEY_CONTROL_BLOCK
          ntkrnlmp!_KLOCK_ENTRY
          ntkrnlmp!_CM_INTENT_LOCK
          ntkrnlmp!_UMS_CONTROL_BLOCK
          ntkrnlmp!_LOCK_TRACKER
          ntkrnlmp!_EX_PUSH_LOCK_AUTO_EXPAND_STATE
          ntkrnlmp!_PLUGPLAY_EVENT_BLOCK
          ntkrnlmp!_ETW_REF_CLOCK
          ntkrnlmp!_KLOCK_ENTRY_LOCK_STATE
          ntkrnlmp!_WAIT_CONTEXT_BLOCK
          ntkrnlmp!_CM_NAME_CONTROL_BLOCK
          ntkrnlmp!_HBASE_BLOCK
          ntkrnlmp!_INTERLOCKED_RESULT
          ntkrnlmp!_CM_CELL_REMAP_BLOCK
          ntkrnlmp!_RTL_SRWLOCK
          ntkrnlmp!_LOCK_HEADER
```

In this listing one can see several datastructures related to synchronization, there are however two promising structures that deserves further investigation. The first interesting structure is the LOCK_HEADER which has the following members:

```
0: kd> dt _LOCK_HEADER
nt!_LOCK_HEADER
   +0x000 LockTree         : _RTL_AVL_TREE
   +0x008 Count            : Uint8B
   +0x010 Lock             : Uint8B
   +0x018 Valid            : Uint4B
```

The Count refers to the number of pages locked by the process, while the LockTree is a binary search tree containing LOCK_TRACKER nodes. By checking all the processes one can see that the LockedPagesList member fits nicely with the LOCK_HEADER structure. Next one can take a look at the LOCK_TRACKER structure:

```
0: kd> dt _LOCK_TRACKER ffffe001`b51203a0
nt!_LOCK_TRACKER
   +0x000 LockTrackerNode  : _RTL_BALANCED_NODE
   +0x018 Mdl              : 0xffffe001`b4767c60 _MDL
   +0x020 StartVa          : 0x0000013a`d1042000 Void
   +0x028 Count            : 3
   +0x030 Offset           : 0xf98
   +0x034 Length           : 0x201a
   +0x038 Page             : 0x12faa9
```

```
+0x040 StackTrace        : [8] 0xfffff801`be85b04c Void
+0x080 Who               : 3
+0x088 Process           : 0xffffe001`b547e840 _EPROCESS
```

To verify that one is indeed looking at a valid LOCK_TRACKER structure one should be able to trace the process EPROCESS block back to the process that contained the LockedPagesList. Working from these assumptions it is easy to verify that indeed the locked pages used by a process is tracked by the LOCK_TRACKER and LOCK_HEADER structures. The entire LockedPagesList structure is an AVL tree consisting of several LOCK_TRACKER nodes. By traversing the AVL tree one should be able to get a complete overview of the pages locked by the process. To see which processes that uses this lock tracking functionality a WinDBG extension was built to locate every process that had locked pages listed in the LockedPagesList:

```
0: kd> !findlockedpages
Listing all processes that have locked pages...

Process: ffffe001b18a2700 [System] ,locked pages: 3874
Process: ffffe001b4679080 [csrss.exe] ,locked pages: 4368
Process: ffffe001b4675080 [wininit.exe] ,locked pages: 2
Process: ffffe001b46fe840 [services.exe] ,locked pages: 2
Process: ffffe001b462d840 [lsass.exe] ,locked pages: 3
Process: ffffe001b476e300 [svchost.exe] ,locked pages: 2
Process: ffffe001b4820080 [dwm.exe] ,locked pages: 16386
Process: ffffe001b478c600 [svchost.exe] ,locked pages: 5
Process: ffffe001b4786840 [svchost.exe] ,locked pages: 4
Process: ffffe001b4a69740 [spoolsv.exe] ,locked pages: 2
Process: ffffe001b5138840 [NisSrv.exe] ,locked pages: 898
Process: ffffe001b3fa9840 [explorer.exe] ,locked pages: 69
Process: ffffe001b376a840 [ShellExperienc] ,locked pages: 2034
Process: ffffe001b43fe440 [SearchIndexer.] ,locked pages: 1
Process: ffffe001b5249080 [SearchUI.exe] ,locked pages: 269
Process: ffffe001b547e840 [vmwp.exe] ,locked pages: 35
```

WinDBG support extensions as dlls and these extensions can be loaded at any time by using the **.load** command. Once loaded one can simply type the name of the function that you want to execute. The **!findlockedpages** function works by traversing the active process list and looking for a process that has a locked pages count larger than 0. What was interesting was to compare the result when you wanted to compare the number of locked pages in use on a system that has an active VM versus a system with no VM. What was apparent is that the number of locked pages did not go up significantly when the virtual machine was active, meaning that these locked pages is not tracked by this structure.

After studying these structures for some time it became apparent that the pages locked in use by a VM could not be found by tracking pages using this method. The focus thus shifted to studying the actual PFNs themselves to see if they could reveal any significant pattern.

3.4 PFNs and the PFN database

Every physical page in system memory is associated with a *physical frame number* (PFN). Each PFN is implemented by a MMPFN structure having the following format:

```
nt!_MMPFN
   +0x000 ListEntry         : _LIST_ENTRY
   +0x000 TreeNode          : _RTL_BALANCED_NODE
   +0x000 u1                : <unnamed-tag>
   +0x008 PteAddress        : Ptr64 _MMPTE
   +0x008 VolatilePteAddress : Ptr64 Void
   +0x008 PteLong           : Uint8B
   +0x010 OriginalPte       : _MMPTE
   +0x018 u2                : _MIPFNBLINK
   +0x020 u3                : <unnamed-tag>
   +0x024 NodeBlinkLow      : Uint2B
   +0x026 Unused            : Pos 0, 4 Bits
   +0x026 VaType            : Pos 4, 4 Bits
   +0x027 ViewCount         : UChar
   +0x027 NodeFlinkLow      : UChar
   +0x028 u4                : <unnamed-tag>
```

Although not fully documented, the PFN structure contains information regarding page state and page attributes. To contain every MMPFN structure Windows reserves the 512GB memory range between 0xfffffa80‘00000000 and 0xfffffaff‘ffffffff [20]. This memory range is also known as the *PFN database*.

To output the contents of the PFN database one can use the WinDbg command **!address -p** which also presents the state of the associated page. Below follows example output of the 11 inital pages found in the PFN database.

```
PFN  Address           PageLocation      Attributes      Ref Cach Usage
================================================================================

1 fffffa8000000030 6:ActiveAndValid -M----------     1 1:C  Private
    Process ffffe001b18a2700 [System]
2 fffffa8000000060 6:ActiveAndValid ------------     2 1:C  DriverLocked
    Process fffff801bea1fa40 [Idle]
3 fffffa8000000090 6:ActiveAndValid ------------     2 1:C  DriverLocked
    Process fffff801bea1fa40 [Idle]
4 fffffa80000000c0 6:ActiveAndValid ------------     2 1:C  DriverLocked
    Process fffff801bea1fa40 [Idle]
5 fffffa80000000f0 6:ActiveAndValid ------------     2 1:C  DriverLocked
    Process fffff801bea1fa40 [Idle]
6 fffffa8000000120 6:ActiveAndValid ------------     2 1:C  DriverLocked
    Process fffff801bea1fa40 [Idle]
7 fffffa8000000150 6:ActiveAndValid ------------     2 1:C  DriverLocked
    Process fffff801bea1fa40 [Idle]
8 fffffa8000000180 6:ActiveAndValid ------------     2 1:C  DriverLocked
    Process fffff801bea1fa40 [Idle]
9 fffffa80000001b0 6:ActiveAndValid ------------     2 1:C  DriverLocked
    Process fffff801bea1fa40 [Idle]
a fffffa80000001e0 6:ActiveAndValid ------------     2 1:C  DriverLocked
    Process fffff801bea1fa40 [Idle]
b fffffa8000000210 6:ActiveAndValid ------------     2 1:C  DriverLocked
    Process fffff801bea1fa40 [Idle]
```

Useful elements of the PFN includes its page location, usage as well as the additional information relating it to the associated process. The page location refers to the current state of the page. In Windows 10 a page can exist in one of 8 states where the state describes gives some information on the use of the page. Windows maintains lists over all the pages in every state allowing the kernel to keep track of available memory. Below follows a table containing all the valid memory states, as well as a description on what the state means.

Page state	Description
Zeroed	A page in this state has previously been in use by another process before returning to the standby list. The zero page writer has overwritten the contents with zeroes to ensure that the page is ready to be allocated by another process.
Free	A page that has never been used. Ready to be used by any process.
Standby	A page that has previously existed in the working set of another process. The contents of the page is still present and the page must be zeroed before allowing another process to allocate it. Standby pages can be used again by the same process without zeroing, a situation known as a soft page fault.
Modified	Similar to the standby state. The only difference is that this page contains modified data that has not been written to disk.
ModifiedNoWrite	Similar to the modified state, only in this situation the data does not need to be written to disk.
Bad	Windows maintains a list over all the physically defective pages in system memory. This list is maintained by monitoring bad memory writes or reads. This state can also be used internally by the kernel to handle state transitions [21]
ActiveAndValid	A page currently in the working set of an active process.
Transition	A temporary state denoting a page in the process of performing an I/O operation.

The memory state gives a basic understanding on what the page is currently used for. Knowing that a page is in the Active state does not tell you more on what it is used for, only that it is in active use. Fortunately WinDbg is able to determine more on the page usage which allows for understanding page usage further. By reverse engineering how WinDbg determines the usage, one can find that there are 25 different page usages. The page usage is determined in the *KmPfnStrPageType* function in the general *ext.dll* WinDbg extension. Each page usage is determined by returning a specific integer value.

In the context of this project the most interesting usage types are the pages marked MDL or DRIVERLOCKED. The way that WinDbg determines if a page is MDL or driverlocked is by looking at the unnamed u4 member in the MMPFN structure. The u4 member is in fact a substructure containing the following members:

```
+0x000 PteFrame          : Pos 0, 36 Bits
+0x000 Channel           : Pos 36, 2 Bits
+0x000 Unused1           : Pos 38, 1 Bit
+0x000 Unused2           : Pos 39, 1 Bit
+0x000 Partition         : Pos 40, 10 Bits
+0x000 Spare             : Pos 50, 2 Bits
+0x000 FileOnly          : Pos 52, 1 Bit
+0x000 PfnExists         : Pos 53, 1 Bit
+0x000 PageIdentity      : Pos 54, 3 Bits
+0x000 PrototypePte      : Pos 57, 1 Bit
+0x000 PageColor         : Pos 58, 6 Bits
+0x000 EntireField       : Uint8B
```

The way that WinDbg determines whether or not a PFN is MDL or driverlocked is by looking at the PteFrame. Any PteFrame having the following bitmask:

```
00001111 11111111 11111111 01101000 0000
```

Will be either MDL or Driverlocked memory. The final test to determine whether the usage type is driverlocked or MDL is done by looking at the owning process.

Since we are interested in knowing which pages are used in the context of driverlocked memory it is interesting that there is a separate type that denotes driverlocked memory. What is also interesting is that the process that "owns" the page is also known. This means that we can separate driverlocked memory on the process level, making it easier to focus on driverlocked memory related to our interests. One thing that can be somewhat confusing is that the output separates between the driverlocked and MDL types. The only difference between the two types is whether the owning process is the idle process or not. Driverlocked memory owned by the Idle process is marked as driverlocked, while driverlocked memory owned by any other process is marked as MDL. In the end we want to focus on driverlocked memory allocated by the virtual machine, so we focus on driverlocked memory owned by vmwp.exe, the virtual machine worker process. Upon searching the entire PFN database for all driverlocked pages owned by vmwp one finds that the total pages are enough to describe the virtual machine, albeit slightly larger. Thus simply searching the PFN database is both slow, since it has to search the entire 512GB memory area, and not specific enough to only contain pages strictly owned by the virtual machine.

3.5 Recreating the virtual machine

At the beginning of this chapter several methods for obtaining a memory dump containing every page used by the secure kernel and IUM were discussed. For this thesis the nested-virtualization approach is used. The reason for choosing this method is that it makes it easy to obtain consistent results and it also does not require tampering with Windows or firmware. This approach requires us to recreate the complete memory space of the virtual machine based on a dump of the host physical machine. The end goal is to recreate the virtual machine from the host memory dump in order to analyze it.

Remember that the host has unrestricted access to the hypervisor running in the VM rendering all pages in the VM fully accessible. Remember that the software responsible for acquiring the VM-memory is ultimately responsible for releasing it, meaning that it is reasonable to expect this information to be stored somewhere.

A reasonable first guess is that this information is stored somewhere in the system heaps, so one can first take a look at the non-paged pool allocations:

```
                NonPaged
Tag      Allocs          Used

EtwB        252      13918240   Etw Buffer , Binary: nt!etw
VdMm         36       9707776   VM-VID , Binary: Vid.sys
MmPb          3       2621440   Paging file bitmaps , Binary: nt!mm
KDNF       1537       2385424   Network Kernel Debug Adapter,Binary: kdnic.sys
Thre       1050       2225888   Thread objects , Binary: nt!ps
VoSm         32       1903824   Bitmap allocations , Binary: volsnap.sys
Pool          6       1726016   Pool tables, etc.
File       4562       1660256   File objects
Ntfx       3684       1241696   General Allocation , Binary: ntfs.sys
VsRD       1034       1164800   VM-NS(RNDIS device) , Binary: vmswitch.sys
AmlH          2       1048576   ACPI AMLI Pooltags
```

One of the highest consumers of non-paged pool is tagged by the VdMm pool tag which is owned by the *Virtualization Infrastructure Driver* (Vid.sys). Another thing that is interesting to note is that the pooltag is suffixed by Mm which often is an abbreviation for memory management in the Windows API. These two factors means that it would be interesting to take a closer look at the allocations done by this pool tag.

Scanning only the large pool allocation table, one finds the following allocations with the VdMm tag:

```
ffffe001b57ea000 : tag VdMm, size    0xc000, Nonpaged pool
ffffe001b6400000 : tag VdMm, size  0x400000, Nonpaged pool
ffffe001b55a8000 : tag VdMm, size    0x4030, Nonpaged pool
ffffe001b55ad000 : tag VdMm, size    0x4000, Nonpaged pool
ffffe001b6000000 : tag VdMm, size  0x400000, Nonpaged pool
ffffe001b55f1000 : tag VdMm, size   0x10000, Nonpaged pool
ffffe001b5a63000 : tag VdMm, size   0x80000, Nonpaged pool
ffffe001b5ae3000 : tag VdMm, size    0xc000, Nonpaged pool
ffffe001b59ef000 : tag VdMm, size   0x10000, Nonpaged pool
ffffe001b5ea5000 : tag VdMm, size   0x80000, Nonpaged pool
```

Most noteworthy is the two largest allocations, each 4 194 304 bytes (0x400000). Looking at the contents reveals the following:

```
ffffe001`b6400000  10400000`001c2200  10400000`001c2201
ffffe001`b6400010  10400000`001c2202  10400000`001c2203
ffffe001`b6400020  10400000`001c2204  10400000`001c2205
ffffe001`b6400030  10400000`001c2206  10400000`001c2207
ffffe001`b6400040  10400000`001c2208  10400000`001c2209
ffffe001`b6400050  10400000`001c220a  10400000`001c220b
ffffe001`b6400060  10400000`001c220c  10400000`001c220d
ffffe001`b6400070  10400000`001c220e  10400000`001c220f
```

What is interesting about the data is that the low 32 bits appear to be sequential and they fit the size of an PFN. One can test this assertion by checking if the data correspond to a valid PFN:

```
Page Frame Number:  1c2200, at address: fffffa8005466000
Page Location:      6 (ActiveAndValid)
PTE Frame:          0000000fffffffd
Attributes:         Cached
Usage:              MDL; Process ffffe001b547e840 [vmwp.exe]
```

After some testing it becomes apparent that these addresses all correspond to valid PFNs used for driverlocked memory owned by the virtual machine worker process (wmwp). In this example the virtual machine in question had 4GB of reserved physical memory. Given that a page is 4kb then the number of pages required to map this address space would be:

$$Pages = \frac{addressSpace}{pageSize} = \frac{4294967296}{4096} = 1048576 \tag{3.1}$$

With two allocations, each 4 194 304 bytes, in which every entry is 8 bytes the total number of pages one can describe is:

$$Pages = \frac{allocSize}{entrySize} = \frac{2 \cdot 4194304}{8} = 1048576 \tag{3.2}$$

Which is the exact size needed to map a 4GB address space. Testing with virtual machines of different sizes confirms that this changes the size of the allocations accordingly which once again confirms our suspicions that these PFNs are used to map virtual machine memory.

At this point one should dump these PFNs to disk so that further analysis can be performed.

3.5.1 Recreating the virtual address space

Having acquired a memory dump of the VM, a new problem surfaces. We now have a raw memory dump that has none of the virtual address space structure implemented by the operating system. Since pointers used by the operating system will also use virtual addresses it is highly important to understand how these are translated properly in order to understand what the operating system is doing. To explain virtual address translation the following definitions are useful:

Definition 3.5.1. The *logical address* space is a continuous address space that describes every address in the main computer memory. These means that a machine having 4GB of RAM has a 4GB logical address space numbered sequentially.

Definition 3.5.2. The *physical address* space is the address space used by Windows to refer to physical memory. The physical address space includes memory from all available devices, not just main memory. For this reason one cannot assume that a address in the logical address space directly translates to a physical address. From the Windows perspective the logical address space is divided into n partitions that can be defined by the vector $\mathbf{x} = (x_1, x_2, ...x_n)$. To map any logical address into the physical address one can define a vector $\mathbf{y} = (y_1, y_2, ...y_n)$ that tracks starting offsets. The physical address space is then defined by the vector $\mathbf{z} = \mathbf{x} + \mathbf{y}$.

Definition 3.5.3. The *virtual address* space is the main address space used by Windows processes and the Windows kernel. As such, most pointers will refer to virtual addresses and every process will have their own virtual address space. Translating a physical address into a virtual address is done through a set of lookup tables known as page tables. To start translation each process keeps track of the physical address of the initial page table, after a context switch this address is loaded into the CR3 register for the current processor. For Intel x86-64 architecture the translation process consists of four page tables laid out like this:

$$(\text{CR3}) \rightarrow \text{PXE} \rightarrow \text{PPE} \rightarrow \text{PDE} \rightarrow \text{PTE} \tag{3.3}$$

With these definitions in place it is clear that we need to find the linear map from logical to physical address space and then find the page tables necessary for virtual address space translation. The easiest way of finding the physical address space layout is by using the SysInternals tool RAMMAP.

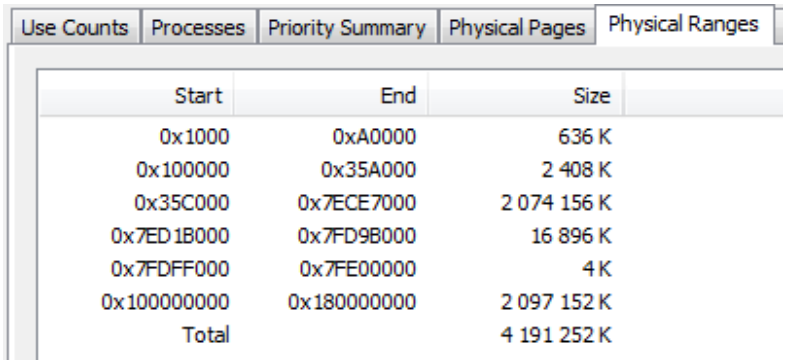

Figure 5: Showing how the physical ranges are laid out on a virtual machine with 4GB of physical memory.

To perform virtual address translation one need to find the page directory of a valid KPROCESS block since this will contain the physical address to begin translation. The simplest way of achieving this is by searching the memory dump for valid pool allocations with the pool tag "Proc". Pool allocations less than the size of a page is allocated with a pool header containing the pool tag. Note that the pool header itself takes up 16 bytes so the allocation can not be exactly the size of a page.

The pool header is laid out like this:

```
nt!_POOL_HEADER
 +0x000 PreviousSize      : Pos 0, 8 Bits
 +0x000 PoolIndex         : Pos 8, 8 Bits
 +0x000 BlockSize         : Pos 16, 8 Bits
 +0x000 PoolType          : Pos 24, 8 Bits
 +0x000 Ulong1            : Uint4B
 +0x004 PoolTag           : Uint4B
 +0x008 ProcessBilled     : Ptr64 _EPROCESS
 +0x008 AllocatorBackTraceIndex : Uint2B
 +0x00a PoolTagHash       : Uint2B
```

Since the EPROCESS and KPROCESS block is always allocated in the non-paged pool one can also specify that the pool type should be 2, and further that the index should be 0. The block size can be determined empirically, but it needs to be at least the size of the OBJECT HEADER + POOL HEADER and EPROCESS block[22]. This allows us to make ballpark estimations on block sizes thus allowing us to reject blocksizes that are either too small or too large. After having found a viable EPROCESS block, the page directory can be retrieved and virtual addresses can be translated. The kernel address space is shared by all processes meaning that any kernel address can be translated using the page directory of any process.

3.5.2 Recreating the secure kernel address space

The description given above works well for recreating the normal kernel address space, but since the secure kernel operates in its own address space one need to recreate this as well. Since the secure kernel does not implement the traditional EPROCESS blocks nor uses pool headers one can not simply follow the same technique, meaning that a different strategy is required. The first thing one need to start virtual address translation in the address of the page directory. A promising candidate can be found by searching for any variables that reference the word "pagedirectory".

```
0:000> x /d /N securekernel!*pagedirectory*
00000001`40059038 securekernel!SkmiLoaderPageDirectoryBase
00000001`40059118 securekernel!ShvlpPageDirectoryBase
```

The most interesting candidate is the *ShvlpPageDirectoryBase*, located at offset 0x59118 from the base. To find the base of the secure kernel one can simply do a brute force search using the 16 first opcodes from the secure kernel entry point. In this example the assembly translates into the following string of opcodes: "48 83 ec 48 48 8b 05 6d 4b 05 00 48 33 c4 48 89". Searching for this pattern gives one hit at physical address 0x024AF150. Knowing that the entrypoint is offset by 0x1150 one can compute where *ShvlpPageDirectoryBase* is located:

$$\mathrm{physAdrVar} = (\mathrm{physAddr} - \mathrm{entryOffset}) + \mathrm{pageDirOffset} \tag{3.4}$$

Once the correct context is set, one can finally translate virtual addresses in the secure kernel address space. Secure processes still retain their own address space in user mode, so it is important to remember to change the context for user mode addresses.

4 Secure Kernel Internals

4.1 Secure Kernel Architecture

The secure kernel is likely based on the normal Windows 10 kernel, albeit a much simpler version of it. The secure kernel is built to provide only basic functionality such as virtual memory and thread dispatching. Compared to the normal kernel it lacks several executive components such as the I/O manager, the configuration manager, the security reference monitor, the PnP manager and the power manager. This means that the secure kernel can not function on its own, but has to rely on functionality found in the normal kernel to operate. The advantages of this simplified architecture is that it reduces the possible attack surface and allows the secure kernel to focus on providing process isolation. The secure kernel can also load kernel extensions which extends the basic functionality as needed.

To begin investigating the secure kernel it is useful to understand the naming conventions used when naming functions and global variables. The Windows API naming convention generally starts with a prefix describing the component responsible for the routine. As an example, the normal kernel uses the "Pp" prefix to denote functionality related to the Plug and Play manager. To denote internal functions one either modifies the last letter in the prefix to include an "i" or adds the letter "p" at the end of the prefix. The main prefixes used in the secure kernel can be found in the table below. Since the secure kernel has no way of accessing the registry or files

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.

directly, its configuration has to be loaded by the normal kernel. Once read into memory, the secure kernel will map the memory into its own address-space by creating section objects to store the data. Before the section object is successfully created the secure kernel will verify the image to make sure that no malicious or corrupt images are loaded.

4.1.1 Secure kernel extensions

The secure kernel on its own does provide only basic functionality needed to create virtual address spaces and handle threads and process objects. The secure kernel can also load kernel extensions, just like the normal kernel, that will introduce new functionality that allows the kernel to do more advanced tasks. What kernel extensions are loaded will depend upon the configuration of the secure kernel. The currently supported kernel extensions are:

- Skci.dll - Secure Kernel code integrity
- cng.sys - Next Generation Cryptography

Skci is the module responsible for implementing Device Guard, while cng provides cryptographic functions. The kernel keeps track of all the currently loaded modules in the global variable *SkLoadedModuleList*. This variable is a pointer to a doubly linked list containing information about all the loaded modules. Each entry in the list consists of a _KLDR_DATA_TABLE_ENTRY structure containing the following information:

```
3: kd> dt ntkrnlmp!_KLDR_DATA_TABLE_ENTRY
   +0x000 InLoadOrderLinks : _LIST_ENTRY
   +0x010 ExceptionTable   : Ptr64 Void
   +0x018 ExceptionTableSize : Uint4B
   +0x020 GpValue          : Ptr64 Void
   +0x028 NonPagedDebugInfo : Ptr64 _NON_PAGED_DEBUG_INFO
   +0x030 DllBase          : Ptr64 Void
   +0x038 EntryPoint       : Ptr64 Void
   +0x040 SizeOfImage      : Uint4B
   +0x048 FullDllName      : _UNICODE_STRING
   +0x058 BaseDllName      : _UNICODE_STRING
   +0x068 Flags            : Uint4B
   +0x06c LoadCount        : Uint2B
   +0x06e u1               : <unnamed-tag>
   +0x070 SectionPointer   : Ptr64 Void
   +0x078 CheckSum         : Uint4B
   +0x07c CoverageSectionSize : Uint4B
   +0x080 CoverageSection  : Ptr64 Void
   +0x088 LoadedImports    : Ptr64 Void
   +0x090 Spare            : Ptr64 Void
   +0x098 SizeOfImageNotRounded : Uint4B
   +0x09c TimeDateStamp    : Uint4B
```

For forensic purposes the most interesting entries is the *DllBase* pointing to the base virtual address where the modules is loaded, as well as the *BaseDllName* containing the name of the loaded module. The forensic toolkit contains the **!sklm** command to easily display all the currently loaded modules:

```
>!sklm 0xFFFFF8024AE4CFC0
Number of arguments: 1
Command: !sklm.

Input string 0xFFFFF8024AE4CFC0, output number FFFFF8024AE4CFC0

--------------Loaded modules------------------
Starting Address:  Ending Address:    Name:
FFFFF8024AE9D000   FFFFF8024AF35000   cng.sys
FFFFF8024AE70000   FFFFF8024AE9D000   skci.dll
FFFFF8024ADF5000   FFFFF8024AE70000   securekernel.exe
```

As can be seen from the output the secure kernel also references itself as a loaded module. As the cng module is also found in the normal kernel the most interesting extension is skci.dll. The next section will cover how this module is actually implemented.

4.1.2 Device Guard implementation

As previously mentioned the device guard portion of the secure kernel is implemented in a separate module, skci.dll. When loading the module it works the way a regular WDM driver

would, the kernel verifies the module and then calls the DriverEntry function when loaded. Once loaded, the kernel routine *SkmmInitalizeCodeIntegrity* calls *SkciInitalize* which will enable device guard verification. Successful completion of *SkciInitalize* includes the following steps:

1. Testing the encryption system. (*CiFipsCheck*)
2. Loads the active CIP policy by calling *CiInitalizePolicyFromPolicies*. This function will perform a series of substeps. The first of these substeps includes initializing several global variables, reading active policies from the secure boot firmware and finally reading the actual CIP binary by calling *CipInitalizeSiPolicy*.
3. Sets up the encryption system used to verify images. (*MinCrypK_Initalize*)
4. Activates the encryption system.
5. If catalog files are available, these files are loaded and verified.

As with the secure kernel skci also uses several prefixes to name the functions and variables. The most notable prefixes are:

Prefix	**Description**
Ci	Main device guard functionality for enforcing code integrity in the secure kernel.
g_	Global variable.
SymCrypt	Symmetric encryption and hashing functionality
SIPolicy	Functionality specific to parsing and enforcing the CIP.
Min	The minimal encryption system
Skci	General device guard functionality

It is important to keep in mind these prefixes as they explain the context that the operation is taking place in. As an example *SIPolicyValidateImage* and *SkciValidateImageData* performs operations in a different context although their names are quite similar. The first function validates any image launched in the normal kernel, while the second is related to validating secure kernel images.

4.2 Secure Kernel Objects

Secure Kernel objects (SKOs) are objects unique to the secure kernel. The object concept has existed since the beginning of Windows NT and offers a way for the operating system to keep track of the resources available to the system. The specific component of the operating system that is responsible for creating, deleting, protecting and tracking objects is known as the object manager. In Windows 7 the number of different object types governed by the object manager had grown to 4242[23], meaning that the operating system uses objects quite extensively. The secure kernel also has an object manager, but supports fewer objects and a different object structure. Since the object structure is quite different this section will provide an overview of the details on the secure objects. For a forensic investigator it is useful to understand secure objects since they give a good insight into what resources the operating system was using at the time.

All SKOs are created by calling *SkobCreateObject* and destroyed once the number of references reaches zero, similar to how objects are handled in the normal kernel. Unlike traditional objects, each SKO starts with a 16 byte header having the following structure:

```
SKO_HEADER:
+0x000 Tag: UInt4B , (Always 0x534B4F42)
+0x004 NoReferences: UInt4B
+0x008 ObjectType: OBJECT_TYPE (Pointer)
```

The object tag translates to the ASCII string "BOKS" and is the same for every object. This means that it would be easy to locate every SKO by searching for this tag. The second entry in the SKO header keeps track of the number of references, this is always initalized to 1 by *SkobCreateObject* and can be increased and decreased using *SkobReferenceObject* and *SkobDereferenceObject*. Finally the object header contains a 64 bit pointer to a OBJECT_TYPE structure that is specific to the object that is created. The OBJECT_TYPE structure itself has the following general layout:

```
OBJECT_TYPE:
+0x000 Destructor: UInt8b
+0x008 Size: (Low 32 bits)
```

The destructor member is a pointer to the routine responsible for cleanup when the reference count reaches 0. The objects currently supported by the secure kernel are the following:

Object name	**Destructor**	**Size (in bytes)**	**Implementation module**
Image Section	SkmiDeleteImage	96	securekernel.exe
Process	SkiDeleteProcess	320	securekernel.exe
Thread	N/A	168	securekernel.exe
Secure Allocation	SkmiDeleteSecureAllocation	32	securekernel.exe
Worker Factory	IumRemoveWorkerFactory	64	securekernel.exe
Event	N/A	64	securekernel.exe
Catalog	SkciDeleteCatalog	8	skci.dll

4.2.1 Image Section Objects

Section Objects are created by calling *SkmmCreateSecureImageSection* to map the desired image file into memory. The OBJECT_TYPE for these objects is the *SkmiImageType*. Although section objects are created for any mapped file in the context of the normal Windows kernel, trustlets cannot do file I/O so the only type of mapped files appear to be images. As a consequence of the secure kernel beeing unable to do file I/O the normal kernel has to read the image into memory before the secure kernel maps this view by calling *SkmmMapDataTransfer*. To ensure that the image is valid the loaded image section is verified, if verification fails *SKMI_BAD_IMAGE* is called. The standard size for section objects are 96 bytes and although their structure has not been mapped out entirely, the most interesting data is two pointers right at the start of the structure. These pointers point to starting and ending address of the page table entries of the mapped data.

4.2.2 Processes

Processes are created by calling IumInvokeSecureService. The OBJECT_TYPE for processes is the *SkeProcessType*. As understanding processes is important from an forensic perspective they will be covered in greater detail in the next section.

4.2.3 Threads

Threads are created by the *SkiCreateThread* internal function which is used by both *SkCreateThread* and *SkeInitalizeThreadPool* to create threads. The OBJECT_TYPE for threads is *SkeThreadType*. More information about threads can be found in the next section.

4.2.4 Secure Allocations

Created by calling *SkmmCreateSecureAllocation*. The object type is the *SkmiSecureAllocationType* and the default size is 32 bytes. During memory forensics of a machine running the secure kernel no secure allocations could be located, meaning that this is not the preferred way of allocating memory. One possible use for secure allocations is to allocate catalog files used by HVCI. The way that catalog files are created is by first allocating a Secure allocation object and then converting it by using *SkmmConvertSecureAllocationToCatalog*. This creates a catalog object that contains the verification data necessary for ensuring code integrity.

4.2.5 Worker Factories

Worker factories is a mechanism that allows for implementing user-mode thread pools in Windows. This thread-pool consists of one or more worker threads that are dynamically allocated as needed by the kernel. The secure kernel imports *NtCreateWorkerFactory* from the normal kernel meaning that is ultimately the normal kernel that is responsible for creating the Worker Factory object. The OBJECT_TYPE for the worker factory object is *SkeWorkerFactoryObjectType* and the size of the object is 64 bytes.

4.2.6 Events

Event objects in the secure kernel offer a seemingly reduced version of the event functionality available to the normal kernel. In fact the secure kernel do not have functionality to create events directly and have to rely on the normal kernel to create the actual object. This event object is represented in the secure kernel as the *SkeShadowSyncObjectType*.

4.2.7 Catalogs

The catalog type is not directly supported by the secure kernel itself, but added by the skci module. All catalog objects is referenced in a table pointed to by *g_CatalogTable*, where the OBJECT_TYPE for each object is *SkciCatalogType*. Catalog objects contain the actual information contained in the catalog files on disk. Since the secure kernel cannot access files directly, the file has to be opened by the normal kernel and the memory copied into the catalog object. Catalog objects are used to verify images that has no digital signature. For more information on catalogs see section 2.2.4.

4.2.8 Using the SKFT to locate secure kernel objects

The SKFT can use the **!skob** command to do a brute force scan of the memory dump for SKOs that has a valid header. The **!skob** command will among other things, output the resulting statistic of the search.

```
Statistics:
Number of processes: 2
Number of images: 110
Number of events: 11
Number of threads: 11
Number of Secure Allocations: 0
Number of Worker Factories: 1
Number of Catalogs: 30
Number of Unknowns: 2
```

This is the output produced from the test machine running device guard and credential guard. The two processes is the secure kernel process and the credential guard process. One of the most prevailent objects are image section objects, totaling 110 instances. The unknowns are false positives that matches the search pattern, but are not valid SKOs.

4.3 Secure processes and threads

Just like the normal kernel, the secure kernel facilitate a simple system to manage processes and threads. For every process or thread created, the kernel will create a object that contains information specific to the entity. The size of each process object is 320 bytes, while the thread object is 168 bytes. The data in the objects contain some of the same information found in the normal kernel objects, but the structure is different. As the structures are not documented this section will provide information on some of the members of each respective object.

4.3.1 Secure processes and trustlets

Processes that run in IUM are known as trustlets and they are handled by the secure kernel similar to the familiar processes in the normal kernel. As in the regular kernel the kernel is also responsible for maintaining data structures for each process, containing information specific to the process. A brief overview of how these structures are laid out will be given later in this section, but the first thing to focus on is how processes are created.

Trustlet creation

After creating the initial process object, the kernel calls *SkeInitalizeProcess* which will first initialize the attributes related to the Trustlet. These attributes are read from the actual file on disk meaning that the executable must be read by the normal kernel and read by calling *SkReadNtosMemory*. Once the initial attributes have been read, the kernel will attempt to execute the executable by calling *IumInitalizeProcess*.

IumInitalizeProcess must perform several steps to load the executable and start the process. These basic steps are:

1. Create a image section object that will contain the process image.
2. Create the Handle Table
3. Map the process image into memory
4. Read policy metadata
5. Create the process environment block (PEB).
6. Create the memory heap used by the process.

Once *IumInitalizeProcess* finishes, the process object is added to the process list - *SkiProcessList*. The *SkiProcessList* is a doubly linked datastructure containing the addresses of every SKPROCESS structure active in IUM. Using the secure kernel forensic toolkit one can see how to get this listing:

```
>!dv 0xFFFFF8024AE4F3F0
Number of arguments: 1
Command: !dv.

Input string 0xFFFFF8024AE4F3F0, output number FFFFF8024AE4F3F0
FFFFF8024AE4F3F0:  FFFF908000090218 FFFF908000090218  . . . . . .
      . . .
FFFFF8024AE4F400:  0000000000000000 0000000000000000  . . . . . .
      . . .
FFFFF8024AE4F410:  0000000000000000 0000000000000000  . . . . . .
      . . .
FFFFF8024AE4F420:  0000000000000000 0000000000000000  . . . . . .
      . . .
FFFFF8024AE4F430:  0000000000000000 0000000000000000  . . . . . .
      . . .
FFFFF8024AE4F440:  0000000000000000 0000000000000000  . . . . . .
      . . .
FFFFF8024AE4F450:  0000000000000000 0000000000000000  . . . . . .
      . . .
FFFFF8024AE4F460:  0000000000000000 0000000000000000  . . . . . .
      . . .
```

Here the process list is located at 0xFFFFF8024AE4F3F0 and in this example only a single process run in IUM meaning that the *flink* and *blink* is the same. The *SkiProcessList* pointer is offset at 0x28 in the SKPROCESS structure and it is this offset that is referenced in the *SkiProcessList*. To read the start of SKPROCESS structure we simply subtract by 0x28.

```
>!sub 0xFFFF908000090218 0x28
Number of arguments: 2
Command: !sub.

Input string 0xFFFF908000090218, output number FFFF908000090218

Input string 0x28, output number 0000000000000028
Result: 0xFFFF90800009O1F0
>!dv 0x0xFFFF90800009O1F0
Number of arguments: 1
Command: !dv.

Input string 0x0xFFFF90800009O1F0, output number FFFF90800009O1F0
FFFF90800009O1F0:  00000000000000A3 0000000000000000  . . . . . .
    . . .
FFFF908000090200:  0000000000000001 6A57644D872835AC  . . . . . 5 ( . M
    d W j
FFFF908000090210:  87F370584B4F92E0 FFFFF8024AE4F3F0  . . O K X p . . . . . J .
    . . .
FFFF908000090220:  FFFFF8024AE4F3F0 FFFF90800000B2D00  . . . J . . . . . - . . .
    . . .
FFFF908000090230:  FFFF90800000C4260 FFFFE001823CE080  ‘ B . . . . . < . .
    . . .
FFFF908000090240:  00000000000001F4 000000000535E000  . . . . . 5 . .
    . . .
FFFF908000090250:  0000000000000009 FFFF90800000903E0  . . . . . .
    . . .
FFFF908000090260:  0000000000000000 00000000000001A7  . . . . . .
    . . .
```

The layout of the SKRPROCESS structure will be given later in this section, but for now we can notice that this is the LsaIso process judging by the Trustlet ID of 1 located at 0x10.

Trustlet data structures

The process object is a 320 byte data structure consisting of a 16 byte SKO_HEADER followed by the 304 byte SKPROCESS structure. The SKPROCESS structure contains similar information to what is found in the EPROCESS structure in the normal kernel, but the offsets are often different. Although not all members of the structure are known, here follows a overview of some important offsets:

```
SKRPOCESS:
+0x010 - Trustlet ID (Integer, 8 byte)
+0x028 - ProcessList (Pointer, 8 byte)
+0x030 - Next Thread Object (Pointer, 8 byte)
+0x038 - Previous Thread Object (Pointer, 8 byte)
+0x050 - Process Id (Integer, 8 byte)
+0x058 - PageDirectory (Physical pointer, 8 byte)
+0x080 - VadRoot (Pointer, 8 byte)
+0x0A8 - PEB (Pointer, 8 byte)
+0x0B0 - Handle Table (Pointer, 8 byte)
+0x0B8 - Trustlet Image Object (Pointer, 8 byte)
+0x130 - Worker Factory Object (Pointer, 8 byte)
+0x138 - Worker Factory Object (Pointer, 8 byte)
```

The trustlet ID is a unique, non-random ID that identifies which kind of trustlet the process object belongs to. The following trustlet IDs are known [17]:

- ID 0 - Secure kernel process
- ID 1 - LSAISO, Credential Guard
- ID 2 - VMSP. Hosts the Virtual TPM(vTPM) on the host
- ID 3 - vTPM provisioning tool.

The thread object pointers references the first and last thread object given in the global thread table. The process ID is the ID used by the normal kernel to reference the process. This ID is generated each time the process is run by the normal kernel and can not be used to identify the process. The page directory points to the physical address of the inital page table used to translate virtual addresses in the context of the given process. Once the appropriate context has been set, one can then look at the Process Environement Block for the particular process. The VadRoot points to the starting node of the VAD tree. A VAD or Virtual Address Descriptor is a kernel structure used to give detailed information on the virtual address space of a given process. The VAD structures follows the same MMVAD structure found in the normal kernel.

The PEB structure is a datastructure that can be referenced from user mode containing information about the operating environment for the given process. The PEB follows the same structure as the PEB specified in ntdll meaning that this datastructure is already defined in the public symbols.

A final thing to keep in mind is that the secure kernel also has a process object of its own running in kernel mode. This object is not referenced in the *SkiProcessList*, but one can find the address for this object by looking at the *PsIumSystemProcess* variable. The process object follows the same structure, but several of the members are just set to null.

4.3.2 Virtual Address Descriptors

Virtual Address Descriptors (VADs) are kernel data structures that provide additional information about the organization of the virtual address space of the given trustlet. Every virtual address reservation is organized into an AVL-tree which can be traversed from the root node in order to enumerate all the virtual addresses currently reserved by the trustlet. VAD trees have been used in previous versions of Windows, but in Windows 10 the implementation is slightly different so this is why this section provides some further insight into VADs on Windows 10.

From a forensic perspective the VAD structure is useful since it greatly limits the address space that must be searched for forensic artifacts. For a user mode address space the available address space is 48 [1] bit, where half of the address space is reserved by the kernel. This means that there is still a 128TB address space where the trustlet could store information. The VAD tree stores every virtual address space reservation which are the only virtual addresses the trustlet actually uses. Since the reserved address space is often a small subset of the full virtual address space this greatly minimizes the relevant search area.

The root node address is found at offset 0x80 where every node is a node in a binary AVL tree with the following structure:

```
SKVAD:
+0x000 - Left node (Pointer, 8 byte)
+0x008 - Right node (Pointer, 8 byte)
+0x010 - Parent node (Pointer, 8 byte)
+0x018 - StartVPN (UINT, 4 byte)
+0x01C - EndVPN (UINT, 4 byte)
+0x020 - StartingVPNHigh (UChar, 1 byte)
+0x021 - EndingVpnHigh   (UChar,1 byte)
```

The StartVPN and EndVPN members represents the starting and ending virtual address of every section of virtual memory. By traversing the entire VAD tree one can obtain the full layout of the virtual address space of the trustlet. The SKFT features the built-in **!vad** command that can traverse the entire VAD tree to show the various sections. Below features example output of the virtual address space of the credential guard trustlet:

```
Vad:                    Start:          End:
FFFF9080000BEF60            7ff9905a0           7ff9905ab
FFFF9080000BEF60            7ff9905a0           7ff9905ab
FFFF90800003E60             0239da6a0           0239da6af
FFFF9080000B2C70            0239da390           0239da40f
FFFF90800009B270            0239da2f0           0239da302
FFFF908000092250            0239da2d0           0239da2e4
FFFF90800008F9D0            07ffe0          07ffe0
FFFF90800008A9A0            0239da320           0239da38f
FFFF908000056390            0239da310           0239da316
FFFF9080000B2DB0            0239da440           0239da53f
FFFF9080000C23A0            0239da410           0239da416
FFFF9080000C21C0            0239da540           0239da5bf
FFFF9080000C3EC0            0239da5c0           0239da63f
FFFF908000033C50            7ff65f240           7ff65f240
FFFF9080000C2240            7ff65f220           7ff65f221
FFFF9080000C4070            7ff65f200           7ff65f201
FFFF9080000C4030            0239da6b0           0239da72f
FFFF9080000C3F00            7ff65f210           7ff65f211
FFFF9080000B2CB0            7ff65f230           7ff65f231
FFFF9080000AD1F0            7ff990540           7ff990546
FFFF908000090390            7ff65fd90           7ff65fdd1
FFFF9080000C3DB0            7ff990530           7ff99053a
FFFF9080000C1230            7ff990570           7ff990584
FFFF9080000B0310            7ff990550           7ff990566
FFFF9080000AFC50            7ff990590           7ff990596
FFFF9080000AA840            7ff990e20           7ff991007
FFFF9080000C0B60            7ff990920           7ff990948
---TRUNCATED OUTPUT---
```

[1] Modern 64-bit processors limit the address space to 48 bits.

Remember to add the high bits to the VPNs to get the actual VPN. Also remember that the VPN only represent the Virtual Page Number and so has to be multiplied with the given pagesize to resolve into a valid address. As an example the starting VPN for the VAD located at 0xFFFF9080000BEF60 is 0x7ff9905a0 and so the virtual address is 0x7ff9905a0000.

4.3.3 Secure threads

As in the regular kernel, the secure kernel provides thread objects that perform the actual work done by the process. After the thread object is created it is added to the *SkiThreadTable* that contains pointers to the thread objects of all processes. Each thread object is 168 bytes consisting of a SKOB_HEADER and a SKTHREAD structure. The SKTHREAD structure has the following layout:

```
SKTHREAD:
+0x000 - Flink, next thread object (Pointer, 8 byte)
+0x008 - Blink, previous thread object (Pointer, 8 byte)
+0x010 - Owning process (Pointer, 8 byte)
+0x018 - Owning process (Pointer, 8 byte)
+0x020 - Thread ID (Integer, 8 byte)
+0x030 - Owning trustlet ID (Integer, 8 byte)
+0x070 - Thread Environment Block (Pointer, 8 byte)
```

By using the SKFT one can take a look at all active threads using the **!dv** command:

```
>!skvars
(Truncated Output)
SkiThreadTable: 0xFFFF900001102000

>!dv 0xFFFF900001102000

FFFF900001102000:  0000000000000000 FFFFF8024AE4F2E0  . . . . . J .
FFFF900001102010:  FFFF908000000890 FFFF908000000950  . . . . . P . . . .
FFFF900001102020:  FFFF908000000A10 FFFF908000000AD0  . . . . . .
FFFF900001102030:  FFFF908000000B90 FFFF908000000C50  . . . . . P . . . .
FFFF900001102040:  FFFF908000000D10 FFFF90800000B2D00  . . . . . -
FFFF900001102050:  FFFF90800000C3E10 FFFF90800000C3F50  . > . . . . P ? . . .
FFFF900001102060:  FFFF90800000C4260 00000000000000E0  ‘ B . . . . . .
FFFF900001102070:  00000000000000F0 0000000000000100  . . . . . .
```

All the valid thread objects begin with 0xFFFF908 addresses, and here we can currently see 11 objects which is the same number of thread objects found by the **!skob** command. If one wishes to find all the thread objects belonging to a particular process one can simply traverse the next or previous thread pointers until one returns to the starting object. Using the owning trustlet ID it is also easy to identify which process the thread belongs to.

5 Conclusion and summary

This conclusion is dedicated to discussing three important aspects regarding the secure kernel and device guard in regards to digital forensics. This means understanding the security, the forensic artifacts as well as how to best aquire forensic information from the secure kernel.

5.1 Security

What is apparent when studying how a secure implementation of Device Guard is implemented, is that Device Guard relies on several security components that has to work together to ensure system security. It has to rely on several hardware mechanisms such as Intel Vt-d and the UEFI BIOS, as well as relying on several software components both in the secure and normal kernel to ensure code integrity. This means that a single bug in one of these components might cause code verification to fail. One example can be found in CVE-2017-0007 where a bug in wintrust.dll caused a powershell script without a valid signature to still be executed by device guard [24]. Wintrust.dll is not part of device guard directly, but rather a dll that exists in the normal kernel. This means that an attacker that wishes to break device guard has a larger attack surface that just the device guard components, making it hard to ensure that every component is secure.

Another challenge comes from creating the CIP itself, since a CIP that is not restrictive enough could allow unintended code to be executed. Ultimately the system administrator needs to find a good tradeoff between how strict the policy should be, and convenience. Using the cryptographic hashes for every executable is obviously the most secure, but difficult to maintain and keep up to date. Even executables signed by Microsoft might be a problem since some verified programs allows for code execution within the running process. One such example is WinDbg or CDB which are powerful debuggers that also allows for executing code through the debugger [25]. There are also examples of malware such as stuxnet that steals digital certificates from otherwise trusted retailers so it is important that the administrator does not accept certificates from too many providers.

In theory the secure kernel design itself should be quite secure as it is a kernel greatly reduced in functionality. Reducing the functionality of the secure kernel means that the attack surface is smaller and even if the kernel is compromised it can not perform many actions anyway. The main challenges lies in making a secure CIP as well as making sure that all components necessary to realize Device Guard is up to date.

5.2 Forensic Artifacts

Since the secure kernel aims to keep it self as simple as possible it also greatly reduces the number of objects tracked by the kernel in comparison to the normal kernel. Another thing that makes tracking these objects easier is the fact that they are all tracked by a single ASCII tag, "BOKS", making it easy to spot where secure kernel objects are located. If one wanted to find all normal kernel objects one had to know the tag that belonged to the specific object, for example the "Proc" tag is used for the process objects. This also means that it is easy to track new additional objects if the secure kernel is updated to support them. The secure kernel toolkit allows for a brute force search of secure objects by using the **!skob** command.

Once a forensic researcher has an overview of all the secure kernel objects it becomes easier to see what kind of configuration the secure kernel is running. As soon as the secure objects are known, then the researcher has a full overview of the running processes, the loaded images, the active threads as well as the catalog files present. This means that if the secure kernel is running any malicious processes it should become apparent based on the secure objects present in the kernel. In the end a good start for any forensic analysis of IUM or the secure kernel is enumerating all the secure objects.

5.3 Live Forensics

The strategy used to obtain full memory dumps of the secure kernel was to use nested virtualization and then capture a full memory dump of the host. The virtual machine memory manager provided kernel structures that kept track of the pages used by the virtual machine, making it easy to extract all the pages from the virtual machine from the full memory dump of the host. Once the dump had been extracted the virtual address space could be extracted by finding the entry point of the secure kernel binary and then using the *ShvlpPageDirectoryBase* variable to obtain the physical address of the initial page directory.

When dealing with a physical machine that is using device guard as well as DMA remapping one likely need to find another solution. Further research needs to be done to see whether using a hardware based approach or a software approach is the best course if a full memory dump is desirable. As mentioned previously a full memory dump might not always be needed as most information is still located in the normal kernel.

6 Documenting the Secure Kernel Forensic Toolkit (SKFT)

The SKFT is a demonstration tool to perform memory forensics on memory dumps of machines running the secure kernel. The tool provides functionality to analyze secure kernel artifacts so that these artifacts can be explored for future reference. This section provides documentation on how to use the tool.

The first thing that happens when the tool executes is that it tries to open the memory dump files that is used for analysis. If this fails the tool will exit. If this succeeds, you are greeted by a command line that allows you to enter the desired command.

6.1 Commands

Each command supported by the tool is prefixed by "!". If the command takes any parameters they will be specified in brackets. The type of the parameters is either a number (NUM) or string (STRING), where the number type can be specified either as a hexadecimal or decimal number. Hexadecimal numbers must be prefixed by "0x" to be interpreted correctly. Some parameters are optional and will be marked as such. The tool can be used to perform memory forensics on both the normal and secure kernel. Since these two kernels differ in their structure certain commands are only meaningful in the context of a given kernel. If a command is not universal (working as intended for both types of kernels), this will be specified.

- **!quit** - (No parameters)
  Exits the program.
- **!dv** - (NUM VirtualAddress, NUM NumBytes (Optional))
  Display virtual memory. Will display the memory contents at the specified virtual address, the optional second parameter specifies how many bytes of memory to display.
- **!setcontext** - (NUM PhysicalAddress)
  Changes the current virtual memory context. The first parameter will be the physical address of the page directory for the new context. Setting the appropriate context is important for virtual memory translation to work. All commands that require virtual address translation will use the new context set by this command.
- **!dp** - (NUM PhysicalAddress, NUM NumBytes (Optional))
  Display contents of physical memory address.
- **!translate** - (NUM VirtualAddress)
  Translates the specified virtual address into a physical address.

- **!processlist** - (No parameters)
  List the currently running processes by traversing the ActiveProcessList. (Only used in the context of the normal kernel).
- **!peb** - (NUM VirtualAddress)
  Outputs a selection of the data in the process environment block (PEB), residing at the specified virtual address. Since the peb resides in user mode it is important to set the appropriate context before dumping the information.
- **!process** - (NUM VirtualAddress)
  Outputs a selection of the data in the EPROCESS block. Note that the EPROCESS block is only used by the normal kernel. The secure kernel uses a similar, but different structure for managing processes.
- **!skvars** - (No parameters)
  Outputs a selection of thel global variables used by the secure kernel. Since this is only meaningful for the secure kernel one should only use this command in this context.
- **!skvar** - (NUM Offset)
  Outputs the data of the global variable at the specified offset. This offset can be found in the symbol information for the secure kernel.
- **!add** - (NUM firstNumber, NUM secondNumber)
  Outputs the sum of two numbers.
- **!sub** - (NUM firstNumber, NUM secondNumber)
  Outputs the result of the operation $n1 - n2$.
- **!skob** - (NUM VirtAddress)
  Searches the entire memory space for secure kernel objects. This is done by searching for the secure kernel tag that identifies the allocation. The output is sorted into different known types of objects. The first parameter specifies the base virtual address of the secure kernel (SkImageBase).
- **!sklm** - (NUM VirtAddress)
  Outputs the currently loaded modules used by the secure kernel. The first parameter specifies the address where the list of loaded modules is found. This address is pointed to by the SkLoadedModuleList variable.
- **!vad** - (NUM VirtAddress)
  Traverses the vad structure and outputs the starting and ending address of every virtual address section. Requires a valid vad root address.

6.2 Architecture

The SKFT is written in C++ and compiled using the Visual Studio 2010 compiler. The program is written as a native CLI program using the Win32 API to interact with the operating system. The project is divided into 11 classes, this section gives a overview of how each class is partitioned. Each class is sectioned into a C++ header (.h) file and a C++ source file (.cpp), each using the same name.

Classname	Description
ConsoleFunc	Contains functionality related to implementing the commandline interface. This includes recording keystrokes, repainting the commandline window and interpreting commands.
DLIST	Implements the doubly linked list datastructure. Contains generic functions for creating and traversing double linked lists.
ForensicTool	Contains the main entry point for the program.
MemoryFunc	Contains functionality for reading memory from dump files files on disk. This includes virtual address translation and reading physical and virtual memory contents.
MiscFunc	Generic functionality related to reading and converting data. This includes reading and manipulating strings, converting raw data into well defined types and reading pointers.
PageEntry	Implements a PageEntry datastructure used in virtual address translation.
PEB	Implements the Process Environment Block datastructure defined in ntdll.
PoolHeader	Not used
SecureKernelFunc	Contains functionality that is related to reading information from the secure kernel memory area.
SupportFunc	Contains various functions related to various memory forensic functionality. This includes searching for patterns, reading data from memory dump files and traversing various structures in the memory dump.

6.3 The custom debugger extension

To dump the required pages from the memory dump a custom debugger extension was built. Microsoft provides a Debugger Engine and Extension API [1]that allows anyone to write their own extensions that extends the already existing functionality. The extension is written in C++ and compiled using Visual Studio 2010 where the finished extension results in a dll file. This extension was also built to explore features such as driverlocked pages which can be seen in action in section 3.3 on driverlocked memory.

The dll provides the following exports:

- driverlocked
- findlockedpages
- findfield
- findmdl
- writemdl
- writemem
- poolheader
- findmdlheaders

Some extensions are used just for testing purposes, the most relevant ones are writemem, findlockedpages and driverlocked. Writemem will write all memory from the virtual machine from the structures allocated by the Virtual Machine memory manager. See section 3.5 for more information. The writemem requires the virtual starting address for the VdMm structure, the size of the structure as well as a valid filename. The findlockedpages and driverlocked extensions are demonstrated in section 3.3. The dll can be loaded by the debugger by using the .load command.

[1]https://msdn.microsoft.com/en-us/library/windows/hardware/ff540525(v=vs.85).aspx

A An introduction to reverse engineering using WinDBG

Since a good amount of the information about the secure kernel is achieved by means of studying the program binaries, this section will provide some insight into how to go about obtaining information from these files. Since it is difficult to perform live debugging of the secure kernel most of this information has been obtained by static analysis. Since the program binaries are 64-bit there is a reduced set of static analysis tools to choose from. This appendix takes a look at how to best use WinDbg, a part of the Debugging Tools for Windows, to perform static analysis. To open a program binary without executing it one can start WinDbg using the -z switch:

```
windbg -z securekernel.exe
```

This will load the securekernel binary so that one can dissasemble program functions as well as study symbol information.

A.1 Using symbol information

Microsoft provides a symbol server[1] that provides public symbols for several Microsoft binaries. Public symbols provide useful information such as the names of global variables and functions as well as providing the structure of some of the data structures. Studying the symbols provides a good starting point when one wants to learn the capabilities of any program binary. To see the available symbols one can use the x command to dump all symbol information:

```
0:000> x securekernel!*
00000001`4003bf90 securekernel!ZwCreateTimer2 (<no parameter info>)
00000001`4000f11c securekernel!SkmiTallyBootPtes (<no parameter info>)
00000001`4003c2a0 securekernel!NkUnmapViewOfSection (<no parameter info>)
00000001`4000be24 securekernel!SkDecryptTrustletData (<no parameter info>)
00000001`40041bd0 securekernel! ?? ::FNODOBFM::`string (<no parameter info>)
00000001`40049830 securekernel!_newclmap = <no type information>
---Truncated---
```

The '*' symbol acts as a wildcard telling the system to print out all the available symbol information for the specific binary. The above is just a short list of the symbols provided for the secure kernel, showing both global variables and the available functions. It is easy to differentiate between a function and a variable as the function names will specify no parameter info, while the variables will specify no type information. The full list of symbols provides the names of a total of 1413 variables or functions so it is useful to know how to narrow down the information further.

The first trick is to sort symbols into groups by looking at the prefixes of the symbol names. Generally Microsoft will assign prefixes that designate the general component of the system that the variable or function operates under. For example the Mm prefix specifies the Memory manager meaning that any symbol that starts with Mm generally belongs under memory manager

[1] http://msdl.microsoft.com/download/symbols

functionality. To sort the output by symbol names one can use the /N switch which makes it easy to see groups of prefixes.

```
x /N securekernel!*
---TRUNCATED---
00000001`400293cc securekernel!HvlSetupLiveDumpBuffer (
00000001`40033928 securekernel!HviIsHypervisorMicrosoftCompatible (
00000001`400338c0 securekernel!HviGetHypervisorInterface (
00000001`40033968 securekernel!HviGetHypervisorFeatures (
---TRUNCATED---
```

Here one can see a group of functions using the Hv prefix, presumably related to the Windows hypervisor. One should also note that prefixes might come in several variants, for example using the "i" suffix to specify an internal function.

To further narrow down the information displayed one can use the /d and /f switches to only output symbol information related to variables or functions. Underneath one can see example output of a symbol search that finds all variables having the word "page" in the name.

```
0:000> x /d /N securekernel!*page*
00000001`400490c0 securekernel!_imp_SkciValidateDynamicCodePages
00000001`40058d10 securekernel!SkmiUnprotectedPageCount
00000001`40059030 securekernel!SkmiReservedPagesStart
00000001`40059028 securekernel!SkmiReservedPagesRemaining
00000001`40058ce8 securekernel!SkmiReservedImagePageLock
00000001`40058cf0 securekernel!SkmiReservedImagePage
00000001`40058d40 securekernel!SkmiPartialIntegrityPages
00000001`40058d60 securekernel!SkmiPageEncryptionContext
---TRUNCATED---
```

A.2 Reversing secure kernel objects

One of the important new artifacts created by the secure kernel are Secure Kernel Objects (SKOs) which are unique to the secure kernel. Since the symbol information for the secure kernel does not include any data structures, one needs to reconstruct the structures from scratch. This section will provide a description on how to use WinDbg to reconstruct these structures.

After constructing a simple version of the SKFT the ASCII string "BOKS" kept appearing in several places when investigating secure kernel memory. Since this was seen several times, it seemed evident that this was not due to random memory patterns, but a deliberate pattern created by the secure kernel. It appeared similar to how pool tagging works in the normal kernel. Pool tags are used to mark the start of kernel heap allocations that are less than the size of a page+pool header size. In this model the tags are 4 byte ASCII strings that identifies the component that allocated the memory. In WinDbg one can find more descriptive information for each pool tag in the pooltag.txt file. Since the "BOKS" tag was not found in the pooltag file it was presumably a new tag unique to the secure kernel.

Listing A.1: "Some example pooltags taken from pooltag.txt"

```
BTMO - bthmodem.sys - Bluetooth modem
CcBm - nt!cc        - Cache Manager Bitmap
PnpF - nt!pnp       - PNPMGR eject data
ObjT - nt!ob        - object type objects
```

If the ASCII string is indeed some sort of tag then judging by how pooltags are named it seems reasonable that the tag itself would be descriptive in some way. A decent guess is that the tag is in fact backwards and should be SKOB, meaning Secure Kernel Object. One possible reason for why the tag is backwards could be because the endianness of the x86 architecture. The x86 architecture is little-endian which means that the least significant bytes are stored first effectively reversing the order of the characters in a ASCII string. This confusion might have led this tag to be accidentally created backwards.

Given the hypothesis that the BOKS tagged allocation is created deliberately by the secure kernel, the next question is what function is responsible for the allocation. A simple way of seeing where this ASCII string is used (if at all) is by searching the program binary using the s command in Windbg:

```
0:000> s -a 0000000140000000 0000000014007b000  "BOKS"
00000001`4001d19a  42 4f 4b 53 c7 40 04 01-00 00 00 48 89 58 08 48  BOKS.@
```

The ASCII string is found exactly once at address 0x000000014001d19a, to see which function is closest to this address one can use the "ln" command.

```
0:000> ln 00000001`4001d19a
(00000001`4001d170)   securekernel!SkobCreateObject+0x2a
```

The address resides inside the space of *SkobCreateObject* strengtening the assertion that BOKS actually refers to secure kernel objects. To see where in the function the tag is used one can dissassemble the function by using the "uf" command:

```
securekernel!SkobCreateObject:
00000001`4001d170 48895c2408      mov     qword ptr [rsp+8],rbx
00000001`4001d175 57              push    rdi
00000001`4001d176 4883ec20        sub     rsp,20h
00000001`4001d17a 488bd9          mov     rbx,rcx
00000001`4001d17d 488bfa          mov     rdi,rdx
00000001`4001d180 8b4908          mov     ecx,dword ptr [rcx+8]
00000001`4001d183 4883c110        add     rcx,10h
00000001`4001d187 e888910000      call    securekernel!IumAllocateSystemHeap
00000001`4001d18c 4885c0          test    rax,rax
00000001`4001d18f 7507            jne     securekernel!SkobCreateObject+0x28

securekernel!SkobCreateObject+0x21:
00000001`4001d191 b89a0000c0      mov     eax,0C000009Ah
00000001`4001d196 eb1a            jmp     securekernel!SkobCreateObject+0x42

securekernel!SkobCreateObject+0x28:
00000001`4001d198 c700424f4b53    mov     dword ptr [rax],534B4F42h
00000001`4001d19e c7400401000000  mov     dword ptr [rax+4],1
00000001`4001d1a5 48895808        mov     qword ptr [rax+8],rbx
00000001`4001d1a9 4883c010        add     rax,10h
00000001`4001d1ad 488907          mov     qword ptr [rdi],rax
00000001`4001d1b0 33c0            xor     eax,eax
```

```
securekernel!SkobCreateObject+0x42:
00000001‘4001d1b2 488b5c2430      mov     rbx,qword ptr [rsp+30h]
00000001‘4001d1b7 4883c420        add     rsp,20h
00000001‘4001d1bb 5f              pop     rdi
00000001‘4001d1bc c3              ret
```

As can be seen from the disassembly the ASCII string "53 4b 4f 42" (BOKS) is moved into the memory address pointed to by rax. We can also see that the memory address originates from the call to IumAllocateSystemHeap which presumably allocates the memory. We can also see that the rax register is used in later instructions to add the hardcoded value 1 as well as some value passed by rbx. The rbx register gets its value from rcx which is used for passing the first parameter in the fastcall calling convention. Finally, the rax pointer is increased by 16 and is stored in the rdi pointer before zeroing eax.

To better keep track of what happens in SkobCreateObject+0x28 it can be useful to convert the assembly instruction into pseudocode based on the observations that can be made from the assembly code. If one can assume that IumAllocateSystemHeap should return a pointer into valid memory upon successful completion then it is reasonable that the rax variable is some sort of memorybuffer making the pseudocode like this:

```
returnvalue = 0;
pMemoryBuffer = IumAllocateSystemHeap(?)
if(*pMemoryBuffer != 0)
{
        pMemoryBuffer->tag = "BOKS"; (0x0)
        pMemoryBuffer->?1 = 1; (0x4)
        pMemoryBuffer->?2 = rbx; (0x8)
        pMemoryBuffer += 0x10;
}
else
{
        returnvalue = 0x0C000009A;
}
```

The returnvalue variable refers to the eax register which can be either 0 or 0x0C000009A upon returning. Since Windows functions often return error codes if anything goes wrong a decent guess would be that 0x0C000009A refers to some errorcode. It is simple to look up Windows error codes using the !error extension in WinDbg:

```
0:000> !error 0x0C000009A
Error code: (NTSTATUS) 0xc000009a (3221225626) - Insufficient system resources
    exist to complete the API.
```

It turns out that the error code is a NTSTATUS value that is returned whenever the system has insufficient resources to complete the request. This would be a reasonable error code to expect since we are trying to allocate memory using *IumAllocateSystemHeap*, and if this function returns 0 (a NULL pointer) we branch into the error braching code at SkobCreateObject+0x21. Based on these observations we can assume that *SkobCreateObject* returns a NTSTATUS code which returns 0 upon successful completion of the function. If *IumAllocateSystemHeap* returns a valid memory address one can see that the first four 4 bytes are tagged using the hardcoded "BOKS" tag. After tagging the memory allocation *SkobCreateObject* initializes two additional variables

using the memoryBuffer offset. In total the size of all three variables is 16 bytes, and we can see that memoryBuffer pointer is changed by the same amount. This suggest that the three variables might be part of a single datastructure having three members.

Given that we now have some notion of what the return type of the function is, it would be useful to understand what parameters are used when calling *SkobCreateObjec*t. In the fastcall calling convention the four first parameters are found in the processor registers (RCX,RDX,R8,R9). As seen in the function prologue only the RCX and RDX registers are referenced specifically, suggesting that *SkobCreateObject* has only two parameters. To understand what type of data is passed in the two parameters it is useful to know which functions calls *SkobCreateObject* directly. For this purpose WinDbg falls short as it has no functionality to cross reference which functions calls another function. Another useful thing by listing the cross references is that it also gives one an

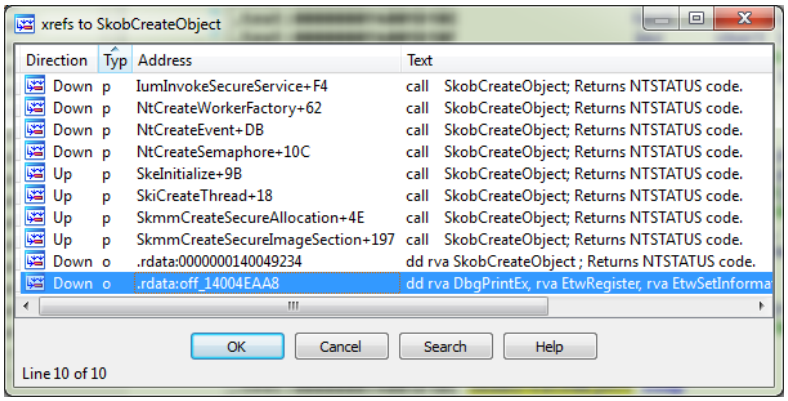

Figure 6: Showing all cross references to SkobCreateObject.

idea of what objects can be created. Selecting one of the functions at random one can see how rdx and rcx are initalized before calling *SkobCreateObject*:

```
securekernel!SkiCreateThread:
00000001‘40002860 48895c2408      mov     qword ptr [rsp+8],rbx
00000001‘40002865 57              push    rdi
00000001‘40002866 4883ec20        sub     rsp,20h
00000001‘4000286a 8bf9            mov     edi,ecx
00000001‘4000286c 488d542438      lea     rdx,[rsp+38h]
00000001‘40002871 488d0d60750400  lea     rcx,[securekernel!SkeThreadType
00000001‘40002878 e8f3a80100      call    securekernel!SkobCreateObject
```

The most notable thing is the rcx register which contains the address of the global variable *SkeThreadType*. Checking the other functions that calls *SkobCreateObject* one can see that rcx always contains the type of object that is created. One can infact use the x command in WinDbg to search for all Global Variables that end in type:

```
0:000> x /d /N securekernel!*Type
00000001‘40051000 securekernel!pctype
00000001‘40049530 securekernel!_newctype
00000001‘40049bd8 securekernel!SkmiSecureAllocationType
00000001‘40049be8 securekernel!SkmiImageType
00000001‘40049c60 securekernel!SkeWorkerFactoryObjectType
```

```
00000001‘40049dd8 securekernel!SkeThreadType
00000001‘4004a5f8 securekernel!SkeShadowSyncObjectType
00000001‘40049bc8 securekernel!SkeProcessType
```

Except for the first two variables, all variables on the list actually reference the type of object that can be created. One can check that this is indeed true by seeing where these variables are used and confirm that are always used to initialize the rcx register before calling *SkobCreateObject*. This means that the secure kernel can directly create 6 types of objects (Device Guard implements an additional catalog object), which reveals that the secure kernel only keeps track of a few objects. Compare this to the over 4000 types used by the normal kernel and it is clear that the secure kernel is a more bare bones implementation of the NT-kernel. With this information in place lets update our view of the *SkobCreateObject* function:

```
NTSTATUS SkobCreateObject(OBJECT_TYPE* objType, PVOID* buffer)
{
        NTSTATUS errorCode = 0;
        PVOID* pMemoryBuffer = IumAllocateSystemHeap(?)
        if(*pMemoryBuffer != 0)
        {
                OBJECT_HEADER *header = pMemoryBuffer;
                header->tag = "BOKS";
                header->unknownDword = 1;
                header->objectType = objType;
                pMemoryBuffer += 0x10;
                buffer = pMemoryBuffer;
        }
        else
        {
                errorCode = 0x0C000009A;
        }
        return errorCode;
}
```

Note that the second parameter is a pointer to the memory allocation allocated by *IumAllocateSystemHeap*. This becomes clear from the assembly where we can see that the rdi register holds the address pointed to by rdx. The next question unanswered question is how *IumAllocateSystemHeap* really works.

Looking at the diassembly of *IumAllocateSystemHeap* reveals that it only seems to refer to a single parameter in the rcx register. Let’s see how the rcx register is used before *IumAllocateSystemHeap* is called:

```
securekernel!SkobCreateObject+0x10
00000001‘4001d180 8b4908          mov     ecx,dword ptr [rcx+8]
00000001‘4001d183 4883c110        add     rcx,10h
00000001‘4001d187 e888910000      call    securekernel!IumAllocateSystemHeap
```

Remember that we found that the rcx register is in reality the object type specific to the object that is created. Since we do not know much about this datatype yet it can be useful to study these object type variables further.

```
0:000> dq securekernel!SkeThreadType
00000001`40049dd8  00000000`00000000 00000000`000000a8
00000001`40049de8  731a0100`13000016 e0b34782`89cf4f50
00000001`40049df8  0000ba76`04c9e8dc 00001000`31415352
00000001`40049e08  00000200`00000003 00000000`00000000
00000001`40049e18  1919ddcd`bf010001 ee5df445`b9df65b5
00000001`40049e28  7e90245b`727f4dc9 9d5475a4`ff40d430
00000001`40049e38  b1ffba4d`909a348c 08d76f8c`aa1b00d1
00000001`40049e48  73c45373`4bf372ac 446d3e40`38fb9cfb
0:000> dq securekernel!SkeShadowSyncObjectType
00000001`4004a5f8  00000000`00000000 00000000`00000010
00000001`4004a608  00000000`00000000 00000000`c0000002
00000001`4004a618  00000000`00000000 00000000`c0000002
00000001`4004a628  00000000`80000011 c0000002`c0000002
00000001`4004a638  00000000`80000011 37363534`33323130
00000001`4004a648  46454443`42413938 00000000`c0000002
00000001`4004a658  c000009a`c000009a 5632d174`00000000
00000001`4004a668  00000002`00000000 0004a828`00000029
0:000> dq securekernel!SkeProcessType
00000001`40049bc8  00000001`40001d30 00000001`00000140
00000001`40049bd8  00000001`4000e7b0 00000001`00000020
00000001`40049be8  00000001`4001b2b0 00000001`00000060
00000001`40049bf8  00000001`40027fd0 00000001`40020580
00000001`40049c08  00000000`00000000 00000001`40027dd0
00000001`40049c18  00000001`40027880 00000001`40027b40
00000001`40049c28  00000001`40027650 00000001`400277a0
00000001`40049c38  00000001`40027c70 00000001`40027c50
0:000> dq securekernel!SkmiSecureAllocationType
00000001`40049bd8  00000001`4000e7b0 00000001`00000020
00000001`40049be8  00000001`4001b2b0 00000001`00000060
00000001`40049bf8  00000001`40027fd0 00000001`40020580
00000001`40049c08  00000000`00000000 00000001`40027dd0
00000001`40049c18  00000001`40027880 00000001`40027b40
00000001`40049c28  00000001`40027650 00000001`400277a0
00000001`40049c38  00000001`40027c70 00000001`40027c50
00000001`40049c48  00000001`40027d70 00000001`40027990
0:000> dq securekernel!SkmiImageType
00000001`40049be8  00000001`4001b2b0 00000001`00000060
00000001`40049bf8  00000001`40027fd0 00000001`40020580
00000001`40049c08  00000000`00000000 00000001`40027dd0
00000001`40049c18  00000001`40027880 00000001`40027b40
00000001`40049c28  00000001`40027650 00000001`400277a0
00000001`40049c38  00000001`40027c70 00000001`40027c50
00000001`40049c48  00000001`40027d70 00000001`40027990
00000001`40049c58  00000001`40027c50 00000001`40028910
0:000> dq securekernel!SkeWorkerFactoryObjectType
00000001`40049c60  00000001`40028910 00000000`00000040
00000001`40049c70  00430067`00750042 006b0063`00650068
00000001`40049c80  0067006f`00720050 00730073`00650072
00000001`40049c90  00000000`00000000 00430067`00750042
00000001`40049ca0  006b0063`00650068 00650064`006f0043
00000001`40049cb0  00000000`00000000 00430067`00750042
00000001`40049cc0  006b0063`00650068 00610072`00610050
00000001`40049cd0  00650074`0065006d 00000000`00310072
```

After studying all the global variables certain things should be apparent. The first is that the OBJECT_TYPE for every object is a 16 byte structure since we can see some of the variables in sequential order meaning that the OBJECT_TYPE cannot be any larger than 16 bytes. The next thing is that the first 8 bytes either refers to another address in the secure kernel or is null. We can use the "ln" command once again to see what functions the first 8 bytes of the OBJECT_TYPE refers to:

- SkeProcessType points to SkiDeleteProcess
- SkeWorkerFactoryObjectType points to IumRemoveWorkerFactory
- SkmiImageType points to securekernel!SkmiDeleteImage
- SkmiSecureAllocationType points to SkmiDeleteSecureAllocation

The common theme here is that first 8 bytes appears to point to the destructor routine for the object given. *SkeThreadType* and *SkeShadowSyncObjectType* either do not have a destructor routine or it gets initialized during kernel startup. With the first 8 bytes accounted for one can now study the last 8 bytes of the OBJECT_TYPE structure. Looking specifically at the low 32 bits one can see that they are all integer values between 0x10 and 0x140. If we remember the previous disassembly of *SkobCreateObject* this was the parameter used to call *IumAllocateSystemHeap*. Given that we are allocating memory it would be reasonable to expect that this is the number of bytes needed for each object. Thus the second member of the OBJECT_TYPE structure is the objectSize. Given this new information we can once again update our pseudocode for *SkobCreateObject*:

```
NTSTATUS SkobCreateObject(OBJECT_TYPE* objType, PVOID* buffer)
{
        NTSTATUS errorCode = 0;
        PVOID* pMemoryBuffer = IumAllocateSystemHeap(objType->size + 0x10)
        if(*pMemoryBuffer != 0)
        {
                OBJECT_HEADER *header = pMemoryBuffer;
                header->tag = "BOKS";
                header->references = 1;
                header->objectType = objType;
                pMemoryBuffer += 0x10;
                buffer = pMemoryBuffer;
        }
        else
        {
                errorCode = 0x0C000009A;
        }
        return errorCode;
}
```

Note that we allocate an additional 16 bytes to make space for the OBJECT_HEADER. Also note that the mysterious second DWORD in the header is actually the number of references to the object. This reference count is implemented in the normal kernel and when this drops to 0 the object is deleted. It would make sense that when a object is created this reference is initialized to 1 as this is the minimum number of references for a valid object. To confirm this hypothesis one should see that the functions *SkobReferenceObject* and *SkobpDereferenceObject* actually increases and decreased this count as expected.

This means that we know the complete workings of *SkobCreateObject*, including knowing the structure of all the parameters and variables. This entire process was accomplished by mostly using WinDbg as well as IDA for cross referencing. This process can be followed for other parts of the secure kernel to uncover how they work. The key idea is to make educated guesses and then test these guesses to see if they make sense. For example the guess that one of the members in the OBJECT_HEADER structure is the number of references could be confirmed by looking if this counter was increased and decreased in the expected way. A good starting point when reversing functions in the secure kernel is to start with understanding what parameters are used by the function as well as understanding the return type. By knowing the function name, the parameters and the return type one should be able to have at least a general understanding of what the purpose of the function is. One should also notice that in this process of reversing *SkobCreateObject* it was not sufficient to study just the assembly of this specific function. Only by studying the assembly of the functions that referenced *SkobCreateObject* as well as studying the functions called by *SkobCreateObject* can you get a full understanding of how the function work.

B Glossary

CIP - Code Integrity Policy.
Specifies which code can be executed under UMCI. Also contains the UMCI settings.

Credential Guard
A special trustlet that leverages the new security features of IUM to keep the credentials and hashes of a given user safe.

Device Guard
A set of security features that is meant to guard against various attacks on a system.

HVCI
Hypervisor based Code integrity. Leveraging the secure kernel to perform code integrity checks.

Hyper-V
The hypervisor provided by Microsoft for use on Windows systems.

IOMMU - I/O Memory Management Unit
Used to control the memory access that DMA enabled devices has.

IUM - Isolated User Mode.
Refers to the new mode of execution that special applications known as trustlets can be executed in.

KMCI - Kernel Mode Code Integrity.
Refers to verifying kernel mode modules.

MDL - Memory Descriptor List
Allows Windows to describe a virtual memory buffer by way of physical memory. By using physical addresses this ensures that the memory contents are the same regardless of context.

PFN - Page Frame Number
Windows designates a page frame number for every page in physical memory. This allows Windows to keep track of every physical page by referring to its PFN.

Secure Kernel
A new minimal Windows kernel that runs in parallel with the normal kernel. The secure kernel is considered more secure because it does not load any third party modules.

SKFT - Secure Kernel Forensic Toolkit
Refers to the software made for this thesis to perform forensic analysis on the secue kernel.

SLAT - Second Level Address Translation
Generic name for technology that is meant to speed up address translation for virtual machines.

Trustlets
Software that execute in Isolated User Mode.

UMCI - User Mode Code Integrity
The user mode portion of device guard. Allows a administrator control over which software that can execute on a given machine.

VAD - Virtual Address Descriptor
VADs are used by Windows to give more information on the layout of the process virtual address space.

VBS - Virtualization Based Security
Using virtualization features on a CPU to provide new security contexts. This can be used to expand on the old two-ring security model introduced in Windows NT.

Bibliography

[1] Zachary, G. P. 1994. *Showstopper!: the breakneck race to create Windows NT and the next generation at Microsoft*. The Free Press.

[2] Intel Corporation. *i860 64-bit Microprocessor Programmer's Reference Manual*, 1990.

[3] AMD. *AMD I/O Virtualization Technology (IOMMU) Specification*, 2016.

[4] Lich, B. 2017. Device guard deployment guide. `https://docs.microsoft.com/en-us/windows/device-security/device-guard/device-guard-deployment-guide`.

[5] 2008. Code integrity. `https://technet.microsoft.com/library/dd348642.aspx`.

[6] Nichols, J. A., Taylor, B. A., & Curtis, L. 2016. Security resilience: Exploring windows domain-level defenses against post-exploitation authentication attacks. In *Proceedings of the 11th Annual Cyber and Information Security Research Conference*, 26. ACM.

[7] Lich, B. 2017. Credential guard protection limits. `https://docs.microsoft.com/en-us/windows/access-protection/credential-guard/credential-guard-protection-limits`.

[8] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2016.

[9] Jiewen, Y. & Zimmer, V. J. *A Tour Beyond BIOS Using Intel Vt-d for DMA protection in UEFI BIOS*. Intel Corporation, 2015.

[10] Witherden, F. 2010. Memory forensics over the ieee 1394 interface.

[11] Halderman, J. A., Schoen, S. D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J. A., Feldman, A. J., Appelbaum, J., & Felten, E. W. 2009. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5), 91–98.

[12] Carbone, R., Bean, C., & Salois, M. 2011. An in-depth analysis of the cold boot attack. *DRDC Valcartier, Defence Research and Development, Canada, Tech. Rep*.

[13] Crincoli, C. 2005. Smis are eeeevil (part 1). `https://blogs.msdn.microsoft.com/carmencr/2005/08/31/smis-are-eeeevil-part-1/`.

[14] Reina, A., Fattori, A., Pagani, F., Cavallaro, L., & Bruschi, D. 2012. When hardware meets software: A bulletproof solution to forensic memory acquisition. In *Proceedings of the 28th annual computer security applications conference*, 79–88. ACM.

[15] Thompson, T. & Cooley, S. Run hyper-v in a virtual machine with nested virtualization. Technical report, Microsoft, 2016.

[16] Cowen, D. 2013. *Computer Forensics: Infosec Pro Guide*. McGraw-Hill.

[17] Ionescu, A. 2015. Battle of skm and ium - how windows 10 rewrites os architecture. In *Blackhat 2015*.

[18] Oney, W. 2003. *Programming the Windows Driver Model*. Microsoft Press, second edition.

[19] Microsoft. 2017. Bug check 0xcb: Driver_left_locked_pages_in_process. `https://msdn.microsoft.com/en-us/library/windows/hardware/ff560212%28v=vs.85%29.aspx`.

[20] Ionescou, A. 2015. How control flow guard drastically caused windows 8.1 address space and behavior changes. `http://www.alex-ionescu.com/?m=201501`.

[21] Russinovich, M. & Margosis, A. 2016. *Troubleshooting with the Windows Sysinternals Tools*. Microsoft Press.

[22] Ligh, M. H., Case, A., Levy, J., & Walters, A. 2014. *The art of memory forensics: detecting malware and threats in windows, linux, and Mac memory*. John Wiley & Sons.

[23] Russinovich, M., Solomon, D., & Ionescu, A. 2012. *Windows Internals Part 1*. Microsoft Press, sixth edition.

[24] Nelson, M. 2017. Defeating device guard: A look into cve-2017-0007. `https://enigma0x3.net/2017/04/03/defeating-device-guard-a-look-into-cve-2017-0007/`.

[25] Graeber, M. 2017. Bypassing application whitelisting by using windbg/cdb as a shellcode runner. `http://www.exploit-monday.com/2016/08/windbg-cdb-shellcode-runner.html`.