BATTLE OF SKM AND IUM

HOW WINDOWS 10 REWRITES OS ARCHITECTURE

ALEX IONESCU
BLACKHAT 2015
@AIONESCU

WHO AMI I?

- Chief Architect at CrowdStrike, a security startup
- Previously worked at Apple on iOS Core Platform Team
- Co-author of *Windows Internals 5th and 6th Editions*
- Reverse engineering NT since 2000 – main kernel developer of ReactOS
- Instructor of worldwide Windows Internals classes
- Conference speaking:
  - Blackhat 2015, 2013, 2008
  - Recon 2015-2010, 2006
  - SyScan 2015-2012
  - NoSuchCon 2014-2013, Breakpoint 2012
- For more info, see www.alex-ionescu.com

PRESENTATION OVERVIEW

- Microsoft is investing heavily into Virtualization Based Security (VBS) in Windows 10 / Server 2016
  - Device Guard
  - Credential Guard
  - vTPM and Guarded Fabric / Shielded VM
- Scarce documentation and a few blog posts are starting to appear on these features
  - And there was a talk about Credential Guard yesterday!
- **This presentation is not about these features – Microsoft will do their own marketing of them**
- All of these features, however, leverage the same internal changes to the Microsoft Hypervisor (Hyper-V) and the NT kernel – a set of changes called **Virtual Secure Mode (VSM)**
- We will “go deep” in the architecture of these changes, and their impact on the platform for security
  - Microsoft official “Going Deep” session on Channel 9 with Dave Probert and Logan Gabriel is live as of 9 hours ago

OUTLINE

- Introduction
- VBS Feature Recap
- VSM Architecture Concepts
- VSM Bootstrap
- SKM and IUM Internals
- Trustlet Internals and Development
- Concluding Thoughts

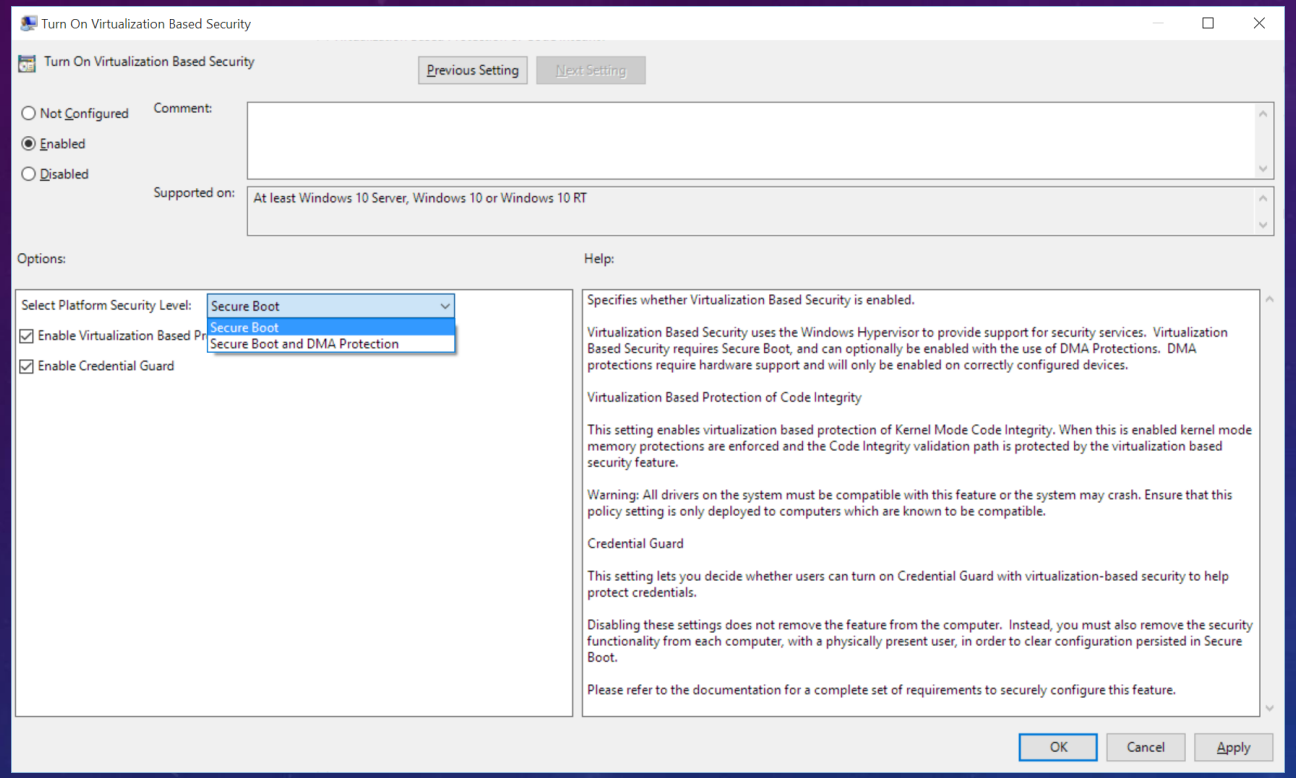
VBS FEATURES IN WINDOWS 10 / SERVER 2016

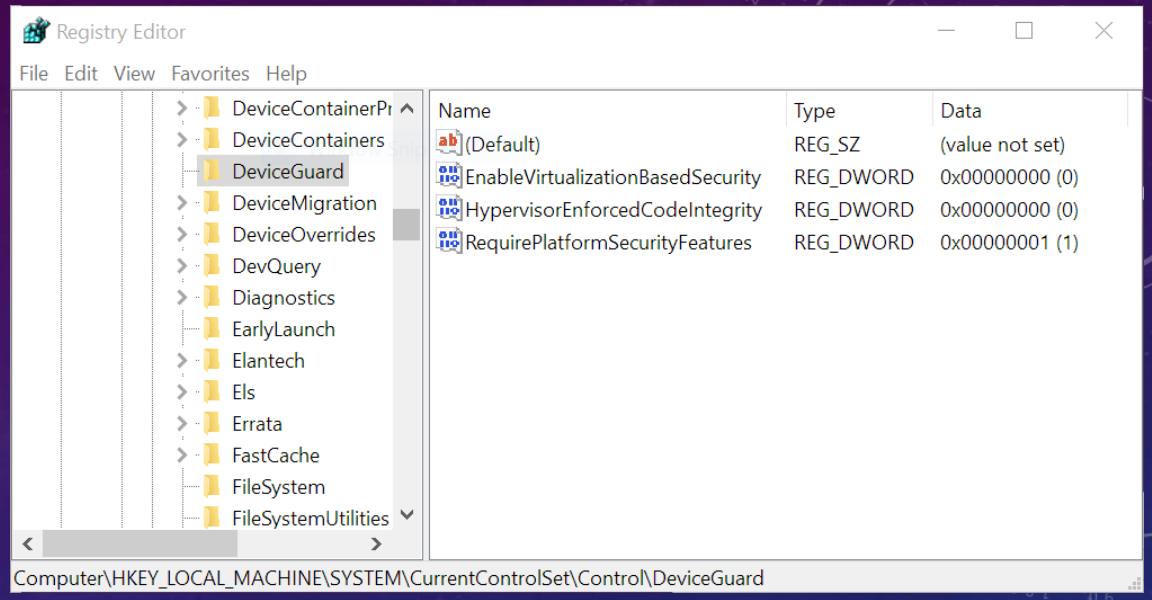
A BRIEF RECAP…

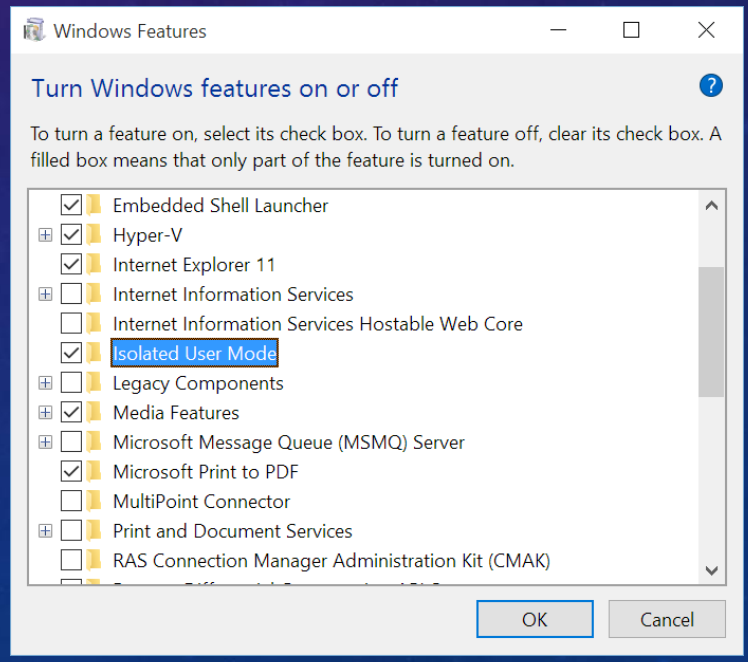
THREE KEY VBS FEATURES BEING INTRODUCED

- Device Guard *[ref: RSA 2015 Microsoft Presentation]*
  - Device Guard allows enterprises to strictly control the execution of everything from PowerShell scripts, to user-mode DLLs and executables, to kernel-mode drivers – can even include anonymous executable memory
- Credential Guard *[ref: BlackHat 2015 Microsoft Presentation]*
  - Credential Guard allows cryptographic secrets (normally owned by LSA, the Local Security Authority) to be kept in memory that cannot be read from anyone on the machine, regardless of privilege level or even hardware access
- Guarded Fabric and vTPM (Virtual TPM) *[ref: Microsoft Ignite 2015 Presentation]*
  - vTPM allows using BitLocker to function on Virtual Machines and to keep data encrypted at all times
  - Shielded VMs leverage vTPM and other security features to make memory owned by the VM invisible to Host

ENABLING VBS







VIRTUAL SECURE MODE (VSM)

ARCHITECTURAL CONCEPTS

HOW DOES IT ALL WORK?

- Hypervisor now associates a **Virtual Trust Level (VTL)** with each **Virtual Processor (VP)**
- Two VTLs defined today (higher is more privileged) – more are supported by the architecture
  - VTL 0, which is the **Normal World**
  - VTL 1, which is the **Secure World**
- Hypervisor uses Enhanced Page Tables (EPT) which now have essentially a “VTL” associated with them
  - VTL 0 access to VTL 1 pages can be controlled
- Blocking +R -> allows hiding cryptographic secrets (Credential Guard)
- Blocking +RX (or +RWX) -> allows preventing execution of code, or modification of code (Device Guard)
- Blocking +W -> allows preventing modification of executable pages shared with VTL 1

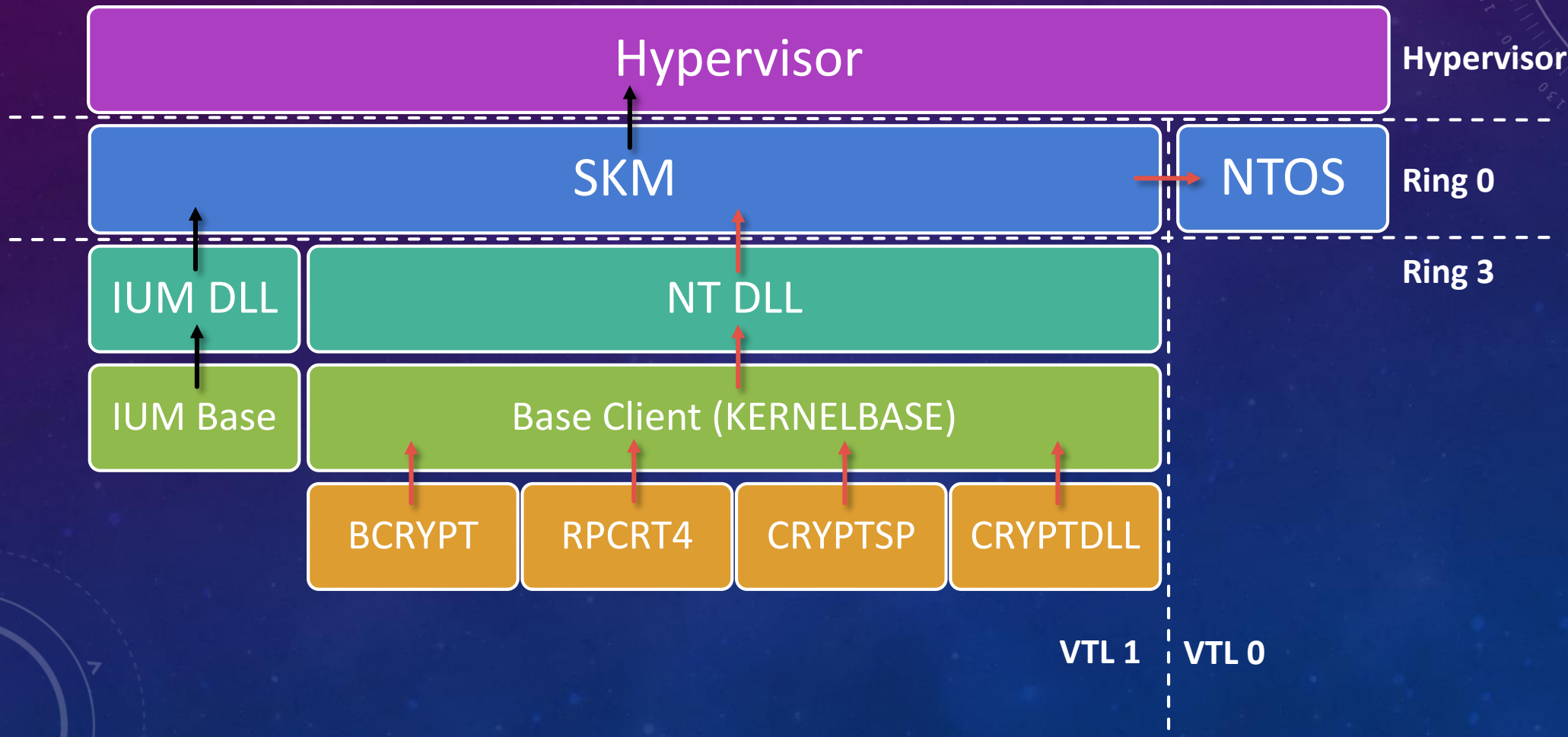
SEPARATION OF POWERS

- The hypervisor itself doesn't implement any code/logic that implements VBS features themselves
  - This reduces the complexity of the hypervisor and the increased attack surface
- A *partition* is an isolated boundary within the Hyper-V world (i.e.: an instance of a virtual machine)
  - Typical Hyper-V setup creates a **root** partition for the host OS, and **child** partition for the guest OS
  - Until Hyper-V Manager is used, only one partition exists (the root)
- Previously, Hyper-V fully trusts the root partition, and root partition has full control of child
- With VSM, the hypervisor will still create the root partition, but start it out in VTL 1
  - The code running is now in **Secure Mode**, and has full access to EPT mappings and can define access boundaries
  - Code then drops to VTL 0, and the rest of the execution continues
- With VSM, the hypervisor no longer implicitly trusts the root partition

SEPARATION OF ROLES

- Instead of loading the full kernel and environment into the secure partition (which would add significant attack surface), a different execution model is loaded
  - The **Secure Kernel Mode (SKM)** environment which runs at Ring 0 (VTL 1)
  - The **Isolated User Mode (IUM)** environment which runs at Ring 3 (VTL 1)
- Secure Kernel Mode includes a minimal kernel, and a few kernel modules (no Win32k.sys!)
  - Kernel -> Called the SMART Kernel or SK
  - Modules -> SKCI.DLL (Provides Code Integrity), CNG.SYS (Provides Cryptographic Services)
- Isolated User Mode includes specialized processes called **Trustlets**
  - Trustlets are isolated among each others and have special digital signature requirements
  - They are limited by the system calls provided through SKM, and do not have access to CSRSS
  - They can only load IUM-approved DLLs
- Everything else in VTL 0 remains as is, and is sometimes called the **High Level OS (HLOS)**
  - It is important to note that barring the IUM-signing restrictions, IUM shares the same binaries as Normal Mode

ARCHITECTURAL LAYER OVERVIEW



PLATFORM REQUIREMENTS

- The Boot Loader needs to trust that the platform it is running on is not running arbitrary firmware
  - This code could modify the bootstrap of the Hypervisor and affect all of VSM's guarantees
  - **Secure Boot** provides this guarantee
  - Also used to safely store protected, persistent variables that cannot be modified (to store policy/settings)
- The Hypervisor needs to trust that hardware is not performing DMA access to VTL 1 pages
  - Flashing a vulnerable device's firmware, or loading a vulnerable/custom driver could allow a VTL 0 administrator access to VTL 1 memory
  - **IOMMU / VT-d** provides this guarantee
- The Hypervisor needs to have a secure place to store and seal machine secrets and keys
  - Otherwise, persistent data such as the Machine Key could be stolen/replayed
  - **TPM** provides this guarantee
- VBS can be configured to require these platform features, or to work around their lack thereof
  - VSM will work regardless, and can be enabled on its own

HYPERVISOR-BASED CODE INTEGRITY (HVCI)

- Windows 10 introduces a new module called SKCI.DLL (Secure Kernel Code Integrity) or HVCI
  - It is a functionally identical, simpler form of CI.DLL, the standard Code Integrity library
  - It is enabled when the hypervisor is enabled, and the administrator configures Device Guard settings
  - It runs as a module part of SKM
- When Strong Code Guarantees are enabled, the kernel becomes unable to create kernel-mode executable page tables
  - Due to the Hypervisor enabling EPT/NPT/SLAT (i.e.: nested page tables)
  - With SLAT enabled, the permissions on the PTEs, even if +X, are further limited by those on EPT entries
  - With VSM enabled, only SKM has permissions to mark EPT entries as executable
- The SKM marks an EPT entry as executable only if HVCI confirms that the code pages are signed according to signing policy

HARD CODE GUARANTEES

- When *hard* code guarantees are enabled, user-mode PTEs cannot be executable either
  - All process execution, DLL loading (i.e.: section object mapping) is first validated through HVCI
- Possible vulnerabilities exist if stray EPT entries are executable
  - It no longer matters that stray kernel PTEs are executable!
  - Write-what-where to fixed PTE entry offsets becomes irrelevant
- Otherwise, vulnerabilities are possible in the Hypervisor/SKM
  - But IOMMU/VT-d is used to prevent against DMA/Hardware/RAM attacks against the hypervisor
  - SMM Mode vulnerabilities, however, could still affect Hypervisor
- Additionally, the hypervisor implements a type of SMEP, even on systems without true SMEP
  - User-mode executable pages are not allowed to be executed by the kernel or marked as kernel
- But, on a Secure Boot system, with TPM, IOMMU, SLAT and the Hypervisor all present, and hard code guarantees enabled…
  - IT IS IMPOSSIBLE TO EXECUTE ANY TYPE OF UNSIGNED CODE (without “Ring -1” vulnerabilities)

VOCABULARY REVIEW

- “Normal World” / “Secure World” – ARM/TrustZone-borrowed terms to describe secured secrets
- SMART/VSM – Secure Mode Application Run Time / Virtual Secure Mode – why we’re here today
  - SK – SMART Kernel, one of the core modules running inside SKM
- HLOS – High Level OS, an older name for “Normal World”
  - NTOS – NT OS, the “Normal World” kernel (Ring 0)
- IUM – Isolated User Mode, another name for “Secure World”, specifically Ring 3
  - SKM – Secure Kernel Mode, the “Secure World”, but running in Ring 0
- VTL – Virtual Trust Level, a synthetic “ring level” managed by the hypervisor (higher is more privileged)
  - VTL 0 -> Normal World/Mode
  - VTL 1 -> Secure World/Mode
- VBS – Virtualization Based Security, the set of features that VSM enables on Windows 10 / Server 2016

VIRTUAL SECURE MODE (VSM)

BOOTSTRAP AND INITIALIZATION

VSM / HYPERVISOR LAUNCH

- VSM is a built-in feature of the Hyper-V 5.0+ hypervisor (Windows 10 / Server 2016)
  - Enabling Hyper-V with the "*hypervisorlaunchtype*" BCD variable will enable VSM
  - But nothing executes at VTL 1 (yet)
- When the hypervisor is configured to launch, *OslArchHypervisorSetup* will eventually call *HvlpLaunchHvLoader*
  - This launches HVLOADER.EFI, which is the boot loader for Hyper-V itself
  - Later, in the second phase of the bootloader, *OslArchHypervisorSetup* runs again, and starts the hypervisor
- With the Hypervisor fully active and initialized, the rest of the boot loader's execution is itself running under the root partition
  - Note: this is a change from Windows 7, where the hypervisor was initialized much later through a boot driver

SKM LAUNCH

- To initialize VTL 1, and provide IUM capabilities, the Secure Kernel must be loaded
  - This will be configured by the various VBS configuration options that the Administrator / Group Policy enables
  - But can also be manually configured by using the "*vsmlaunchtype*" BCD variable
- Loading the Secure Kernel is done by the *OslVsmSetup* routine inside of the boot loader (WINLOAD.EFI)
  - This will create the VSM Loader Block, which contains all the key data structures for the Secure Kernel to initialize
  - Boot loader will provide an initial set of boot stack pages, shared system buffers, map the API set schema, map code integrity data, pass policy information, and more
  - Boot loader is also responsible for generating, provisioning, measuring and sealing the IDK (**Identification Key**) and the LKEY (**Local Key**) which are used to provide strong cryptographic guarantees by Credential Guard and vTPM
- Boot loader will then load the Secure Kernel itself (SECUREKERNEL.EXE), as well as its dependencies
  - Namely SKCI.DLL and CNG.SYS

BOOT VSM POLICY

- Implemented by *BlVsmSetSystemPolicy*, this checks which options are enabled for VBS
  - Policy options are read/stored from a Secure Boot Variable called “VbsPolicy” and locally cached in the *BlVsmpSystemPolicy* structure
  - Can be fully disabled with Secure Boot Variable “VbsPolicyDisabled”
- *BlVsmSetSystemPolicy* receives information from *OslSetVsmPolicy,* which parses the registry key **HKEY_LOCAL_MACHINE\SYSTEM\Control\DeviceGuard**
  - **EnableVirtualizationBasedSecurity** – Enables VSM
  - **RequirePlatformSecurityFeatures** – Forces SecureBoot and/or IOMMU (VT-d) Platform Support
  - **Mandatory** – Forces VSM to be on
  - **Unlocked** – Configures whether “VbsPolicy” Variable is written to force settings to be permanent
  - **HypervisorEnforcedCodeIntegrity** – Configures HVCI in either strong or hard (strict) mode

BCD VSM POLICY OPTIONS

- The policy options collected are then sent to *VsmpSetBootOptionsForPolicy* which populates the following BCD settings
  - 0x250000F0 (Unknown) – Root Partition Only
  - 0x25000142 BCDE_OSLOADER_TYPE_VSM_LAUNCH_TYPE (*vsmlaunchtype*)
  - 0x25000115 BCDE_POLICY_OSLOADER_TYPE_HYPERVISOR_IOMMU_POLICY (*hypervisoriommupolicy)*
  - 0x25000119 BCDE_OSLOADER_TYPE_HYPERVISOR_MMIO_NX_POLICY (*hypervisormmionxpolicy*)
  - 0x25000118 BCDE_OSLOADER_TYPE_HYPERVISOR_MSR_FILTER_POLICY (*hypervisormsrfilterpolicy*)
- These are then parsed by the Hypervisor Loader to enable particular features or requirements
  - Code inside *HvlpInitializeBootParameters* sets these in its internal boot flags structure

HYPERVISOR MSR FILTERING AND NX MMIO

- When HVCI is turned on, both MSR Filtering and NX MMIO features will also be turned on
  - They address highly specialized attacks against VSM
- Many drivers use device-mapped memory (MMIO) to access device registers or memory through virtual memory mappings accessible from Ring 0 (*MmMapIoSpace*)
  - Previous to Win 10, there was no API to request these pages to be non-executable – now *MmMapIoSpaceEx*
  - However, legacy drivers will still obtain/request executable MMIO pages – could be used to run arbitrary code
  - The NX MMIO policy will prevent this from happening at the Hypervisor level, using EPT to make the pages NX
- Secondly, certain MSRs could be used from a Ring 0 attacker to affect the platform and workaround HVCI guarantees
  - TLB attacks and split-TLB tricks, for example, can both be used to leak, hide, and execute arbitrary code
  - Microsoft really did a thorough job here. For example, even AMD's "Secret" MSRs are guarded, such as **C001_1023** (The Bus Unit Configuration Register, which mediates TLB, among other things)
  - This also protects Virtualization-related MSRs (Such as VM_CR on AMD Pacifica) or OS-sensitive MSRs (Such as LSTAR)

SKM INTERNALS

SKM FUNCTION LAYOUT

- Similar to NTOS, SKM has different modules that handle various parts of its mechanisms
  - *Ke (Ki), Etw, Rtl, Dbg, Se* appear as the usual suspects
  - *Ium* is the IUM secure system call interface
  - *Nt/Zw* is the SKM system call interface
  - *Nk* are the NTOS normal mode system call stubs
  - *Shvl, Skob, Skmm(Skmi), Ske(Ski), Skps* are the SKM counterparts of the respective NT interfaces
  - *Sk* are generic Secure Kernel helpers/functions used by various parts of the kernel (catchall prefix)
- It also exports certain functions mainly for CNG.SYS to load correctly
  - Most will immediately bugcheck if called, as CNG.SYS is not expected to call most of these
  - But, those that begin with *Sk* are exported for SKM Modules (Pool, Locks, Objects)
    - CNG.SYS detects if it's running in SKM, and will call these routines instead of the NTOS ones
  - A few Rtl, Etw, and CRT functions are exposed as well

SKM STRUCTURES

- SKM has a PCR at gs:[0], but with a different structure than NTOS
  - CurrentThread is at offset 0x08, CurrentProcess is at offset 0x48
  - Hypervisor data is at offset 0x10
  - We call this **SKPCR**
- The CurrentThread is similar to a KTHREAD structure, but again, specific to SKM
  - PreviousMode is at offset 0x30, for example, the TEB at offset 0x68
  - We call this **SKTHREAD**
- CurrentProcess is akin to KPROCESS
  - NTOS KTHREAD is at 0x58
  - PID at 0x48, PEB at 0xA0, Handle Table at 0xA8, Mailbox Slots at 0xD0, etc…
  - We call this **SKPROCESS**

MAILBOXES

- Secure Communications Channel between IUM and NTOS
- Each Secure Process can have up to 8 Mailbox Slots
  - Once a slot is ‘filled’, it must be retrieved before it can be used again
  - Each slot can contain up to 4092 bytes of data (4 bytes are used to store the size, for a total of 1 page)
- *PostMailbox* will fill a slot on the IUM side
- SKM side uses *SkRetrieveMailbox* to read the data, and then performs a Normal Mode Service Call to copy the mailbox data to NTOS
  - NTOS can access mailboxes by using the *VslRetrieveMailbox* API which is exported by the kernel

SKM CAPABILITIES

- The *SkCapabilities* array defines certain IUM System Calls that can be access controlled on an individual Trustlet basis using one of the following rules
  - AllowAll
  - CheckByTrustletId
  - CheckByTrustletInstanceGuid
- *SecureStorageGet* and *SecureStoragePut* are the two IUM System Calls that currently check for capabilities
  - The *Get* API only allows Trustlet ID 2 to call it
  - The *Set* API only allows Trustlet ID 3 to call it
- **This implies a certain hardcoded knowledge of Trustlets baked into SKM**

STORAGE BLOBS

- Secure Communications Channel between different IUM Trustlets with the same Instance GUID
- Each Trustlet Instance has 2 Storage IDs
- Each Storage ID can have an one of **Storage Blob** of up to 4088 bytes
  - The rest is used up by a header, for a total of 1 page
- A Trustlet can get only Storage Blobs that belong to its own Instance GUID
- All Storage Blobs are in a system-wide LIST_ENTRY called *BlobList*
- Given the SKM Capability model we just saw, this means that Trustlet 3 can push data for the Instance GUID it manages, and Trustlet 2 can read data
- We'll shortly see why mailboxes and storage blobs exist – they are specialized for a particular purpose

SECURE MODE CALLS

- **Secure Mode Calls** are services that SK provides to NTOS
- NTOS uses the *SkCallSecureMode* routine and passes a special structure that identifies
  - The Operation Code
  - The Service Code
- Three operations are possible
  - **Resume Thread [0]**
  - **Secure Service Call [1]**
  - **TLB Flush [2]**
- This puts the CPU into VTL 1 and executes the handler for VTL which will run in SKM
  - VTL 0->1 Switch is done with VMCALL instruction, RCX == 0x11 (RCX is saved into RAX)

SECURE MODE SERVICE CALLS

- NTOS needs to query SKM for protected secure policies, as well as ask it to take actions that it now owns depending on what VBS features are enabled
  - For example, allocating executable memory is not allowed in HVCI Strict Mode without code signing
    - Likewise, changing the permissions to make executable memory writable, or writable memory executable
  - Loading an IUM image and validating its signature and catalog data can only be done by SKM
  - Creating an IUM secure process, secure thread, secure PEB, secure TEB, and querying them
  - Debugging an IUM secure process, or setting/changing the context of an IUM secure thread
  - TLB flushing must be done with SKM, since it owns the EPTs
  - Hibernation, crashdump, shutdown, and kernel debugging must flow through SKM for encryption/security
- At boot, NTOS uses secure service calls to inform of its initialization stages and exchange entropy
- Mailbox data is retrieved through a secure service call as well

SPECIALIZED SECURE MORE SERVICE CALLS

- Because the secured partition really runs on the same processor as the root partition, if there are no entries into the secure world, the VP will remain in VTL 0 forever
  - However, the secured partition might have work items that need to be processed, such as deferred pool frees
  - A special periodic service call exists to force VTL 1 entry and process work items
- Because ETW tracing uses embedded checks for the tracing level in code, SKM needs to know if ETW was enabled for the IUM Provider
  - A special service call updates the ETW mask
- There is also a special service call to register ‘failure log pages’
  - These are then written into by any of the HVCI-related service calls in case of failure
  - Registered by *HvlRegisterLogPages* during *PspIumInitialize*, and stored in PspIumLogBuffer

NORMAL MODE CALLS

- **Normal Mode Calls** are the opposite of Secure Mode Calls – they are services provided by NTOS to SK
- SK uses the *SkCallNormalMode* routine by passing the same structure that's used for Secure Mode Calls
- Four operations are possible
  - **Normal Service Calls [0]**
  - **Normal System Calls from IUM (i.e.: Ring 3, Previous Mode == User) [2]**
  - **Normal System Calls from SKM (i.e.: Ring 0, Previous Mode == Kernel) [3]**
  - **Virtual Interrupt Assertions (VINA) [4]**
- This puts the CPU into VTL 0 and executes the handler for VTL return which will run in NTOS
  - VTL 1->0 Switch is done with VMCALL instruction, RCX == 0x12 (RCX is saved into RAX)
- Finally, when secure mode ultimately ***returns*** back to normal mode, operation 1 is used (IUM Return)

NORMAL MODE SERVICE CALLS

- SK needs access to certain resources and mechanisms that are only implemented in NTOS as part of its behavior
  - This is different from the Normal Mode System Calls which are needed by IUM – these calls are for SKM proper
- These are performed with the same secure call interface that's used to enter SKM, but this time using the VtlReturn instruction
  - Allocate/Free/Get/PhysicalPages [1, 2, 4]
  - TerminateThread/Process [7, 8]
  - AlertThreadByThreadIt, WaitForAlertByThreadId, CheckForApcDelivery [10, 11, 12]
  - Allocate/Free/Copy/VirtualMemory [15, 16, 13]
  - DebugForwardException/SuspendProcess/ResumeProcess/SendError [9, 18, 19, 20]
  - EtwRegister/SetInformation/WriteTransfer [21, 22, 23]
  - GetCurrentProcessorNumber [17], GetSecureImageHandle [14]
  - DbgPrint [3]!

UEFI RUNTIME CALLS

- The UEFI specification provides firmware with the ability to expose certain functions that remain resident in memory, and can be called from the live, running OS
  - A lot of complexity exists here around locking and memory map layouts, since there's no facility provided to synchronize with the OS
- NT uses certain key UEFI Runtime Calls, such as the ones that provide the ability to enumerate, get, and set EFI variables
  - System reset/shutdown also done through UEFI, as well as Capsules (firmware updates / crash dump data)
- Specifically, the UEFI Variable API exposed to user-mode (*Get/SetFirmwareEnvironmentVariable)* performs a system call which is ultimately handled by the HAL, which then calls UEFI
  - So what happens if an IUM application wants to access UEFI Runtime Calls?
- VSM can therefore be configured to virtualize UEFI Runtime Calls, and to run them in VTL 1
  - NTOS (or rather, HAL) therefore will perform a Secure Mode Service Call to access UEFI
- This means that malicious firmware could access VTL 1 data during this time!

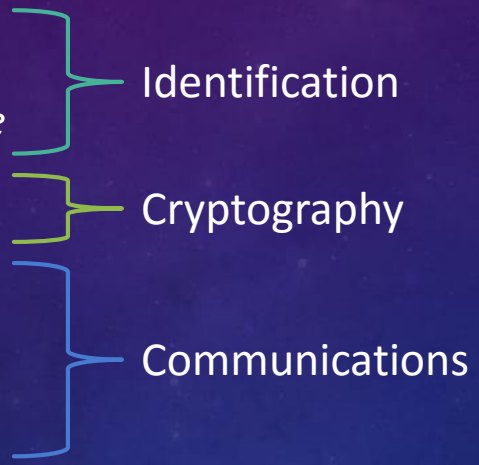
IUM INTERNALS

CORE IUM-EXPOSED SKM SERVICES

- SKM provides three basic services to Trustlets
  - Secured Identification (using the IDK and Trustlet Instance GUIDs)
  - Secured Communication (using Mailboxes and Secure Blobs)
  - Secured Cryptography
- These basic services are implemented through the system call interface
  - *syscall* instruction on x64
- Following the standard Windows NT model, Trustlets interact with a “base” (Subsystem) DLL, which in turn calls the “native” DLL

SECURE SYSTEM CALLS

- SKM exposes six system calls that have their high bit set
  - 0x80000000 – *IumGetIdk*
  - 0x80000001 – *IumSetTrustletInstance*
  - 0x80000003 – *IumCrypto*
  - 0x80000002 – *IumPostMailbox*
  - 0x80000004 – *IumStoragePut*
  - 0x80000005 – *IumStorageGet*



- The *IumCrypto* API internally exposes five different subcalls
- System Calls are exported by the Native DLL (IUMDLL.DLL)

CRYPTO SUBCALLS

- With *IumCrypto* the following services are exposed
  - Encrypt Data (0)
  - Decrypt Data (1)
  - Decrypt IDK-bound data (2)
  - Get SKM-generated seed (3)
  - Get SKM FIPS Mode (4)
- Each of them is passed in by using an internal structure (IUM_CRYPTO_PARAMETERS)
  - But we don’t have to worry about the internals of this…

SECURE BASE API

- Trustlets shouldn’t call the system calls on their own
  - Just like Win32 applications don’t call NTDLL.DLL
- Instead, the Base API provides us the following exports:
  - *DecryptData*
  - *DescryptISKBoundData*
  - *EncryptData*
  - *GetFipsModeFromIumKernelState*
  - *GetSecureIdentityKey*
  - *GetSeedFromIumKernelState*
  - *PostMailbox*
  - *SecureStorageGet*
  - *SecureStoragePut*
  - *SetTrustletInstance*
- They take care of any internal details and expose a simpler interface

IUM SYSTEM CALLS

- Only 48 system calls are currently allowed from IUM Trustlets, covering
  - Synchronization (Semaphores, Events and Alerts)
    - Allows Critical Sections and SRW Locks API to work
  - Worker Factories, Thread API, and Wait Completion Packets
    - Allows Thread Pool API to work (used by RPC)
  - ALPC
    - Allows RPC to work
  - Virtual Memory and Section Objects
    - Allows Heap API and Memory Mapping API to work
  - *NtRaiseException* and *NtContinue* for SEH Support
  - A few query APIs are partially supported (*NtQuerySystemInformation(Ex), NtQueryInformationToken, NtQueryVolumeInformationFile, NtQueryPerformanceCounter, NtQueryInformationThread, NtQueryVirtualMemory)*

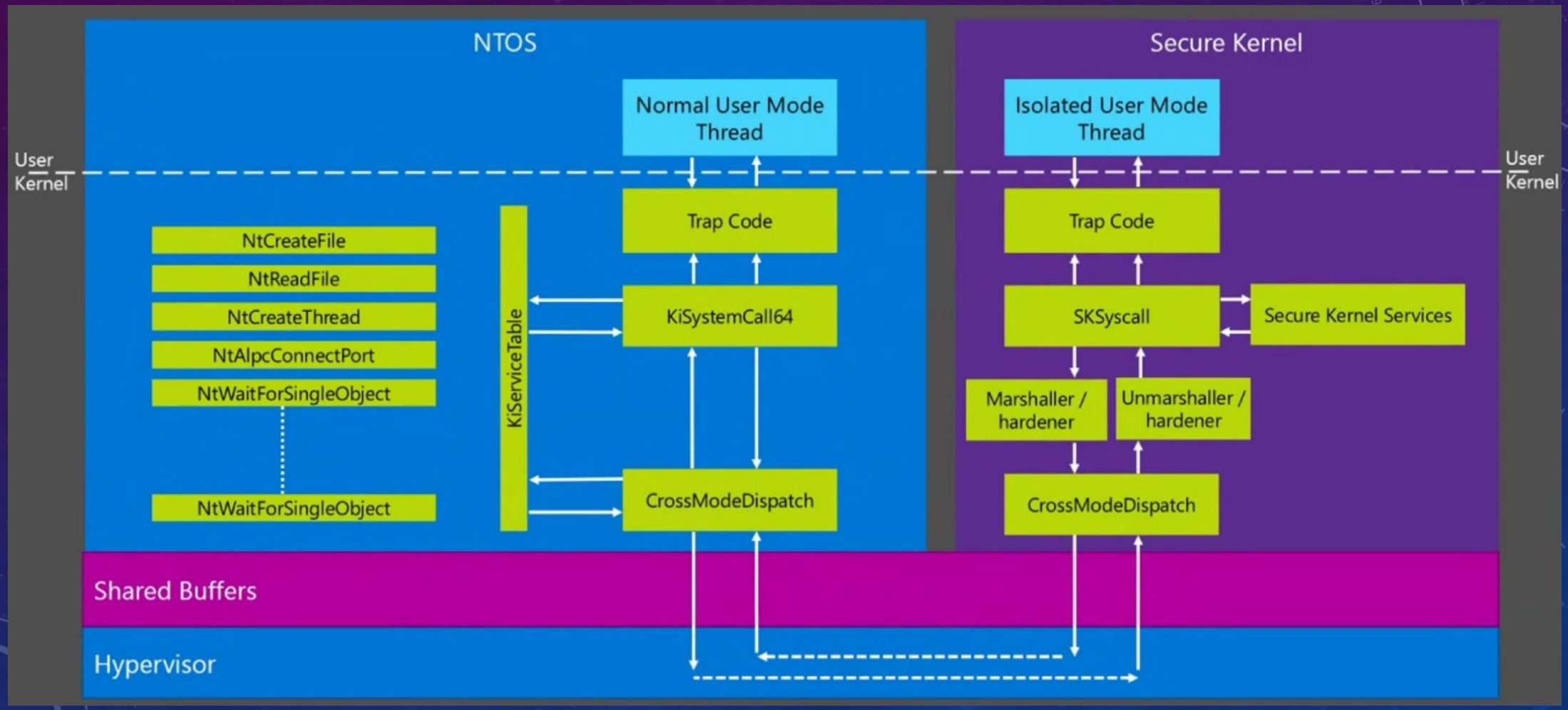
NOTABLY MISSING…

- Trustlets cannot do any registry I/O
  - *NtOpenKey(Ex)* are both implemented, and return STATUS_OBJECT_NAME_NOT_FOUND
- Trustlets cannot do any file I/O
  - Including device I/O (there are no devices to talk to anyway)
  - This obviously means no network, sound, video I/O either
- The Trustlet model therefore implies the existence of a Normal Mode “Agent” to perform these actions, if needed
  - Since RPC is permitted, this follows a similar model to the Broker/Renderer model of AppContainer/Chrome
- Also, the lack of certain basic things like mutexes means some standard user-mode libraries won’t work correctly or fail
  - The goal was to support vTPM and Credential Guard, so don’t expect to run arbitrary Trustlets

IUM SYSTEM CALL SECURITY

- All IUM system calls are subject to *marshalling* and *sanitization*
- *Marshalling* is done on a per-type basis for the following structures
  - ALPC_MESSAGE_ATTRIBUTES, OBJECT_ATTRIBUTES, PORT_MESSAGE, SID, UNICODE_STRING, WORKER_FACTORY_DEFERRED_WORK
  - GENERIC is used for simple, flat types
- *Marshalling* operations are done twice before the call (once to obtain size, once to actually marshal into the input buffer), and once after the call
  - *NtTraceEvent* has its own special marshalling logic
- *Marshalling* after the call involves *sanitization*
  - This is done in a generic fashion for most system calls using the types above, but some have their own logic
  - In some cases, *sanitization* will completely override any data returned from NTOS

NORMAL MODE SYSTEM CALL PROXYING



TRUSTLET INTERNALS

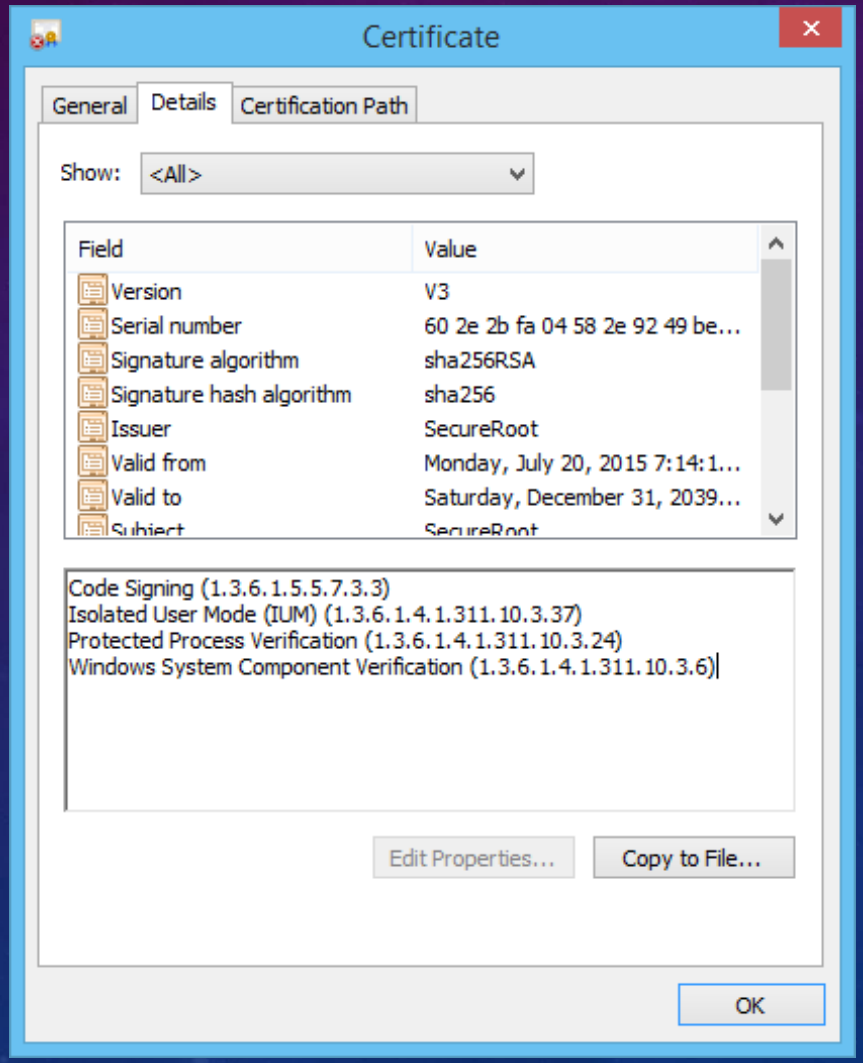
LAUNCHING A TRUSTLET

- Process Attributes were added in Windows Vista as part of a major refactoring to process creation, in part to support **Protected Processes**
  - The *CreateProcess* API can use a STARTUPINFOEX structure which contains a pointer to an **Attribute List**
  - Microsoft documents *Initialize/Update/DeleteProcThreadAttributeList* and a set of attributes to use
- However, the *CreateProcess* API accepts Win32 attributes that are then later converted to NT attributes
  - *NtCreateUserProcess* then uses the converted NT attributes
  - There is a much larger set of NT attributes than Win32 attributes, but they cannot be passed to *CreateProcess*
- To launch a Trustlet, a special **Process Attribute** must be used, but it is an NT attribute, not Win32
  - So we need to build our own Trustlet Launcher, using *NtCreateUserProcess*
  - *PsAttributeSecureProcess* can be used to define and use PS_CP_SECURE_PROCESS
- The attribute is a single 64-bit value: the Trustlet ID
  - Stored in SKPROCESS at offset 0x8

TRUSTLET CRYPTOGRAPHIC REQUIREMENTS

- As soon as IUM is enabled, HVCI is ultimately responsible for deciding if a binary will be permitted to actually run as a Trustlet
- Trustlets have two critical guarantees that must be met
  - They must have a Signature Level of 12 [ref: http://www.alex-ionescu.com/?p=146]
    - This means they must have the Windows System Component Verification EKU (**1.3.6.1.4.311.10.3.6**)
  - They must have the IUM EKU
    - **1.3.6.1.4.1.311.10.3.37**
- Optionally, if a Trustlet is meant to run as a Protected Process Light (PPL) as well, it should have the respective EKUs as well
  - i.e.: **1.3.6.1.4.1.311.10.3.22** or **1.3.6.1.4.1.311.10.3.24**
- The decision and logic to allow a binary to run as a Trustlet is done by *CipMincryptToSigningLevel*
  - **NOTE: System component checks are skipped if Test Signing is turned on and a Test Root was used**

EXAMPLE IUM-COMPLIANT SIGNED BINARY



TRUSTLET POLICY / IUM POLICY METADATA

- When a Trustlet loads, it must have a Trustlet Policy associated with it
  - Stored in a variable called **s_IumPolicyMetadata**, which must be exported
  - Must be located in a PE section called "**.tpolicy**", which must be **IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ**
- The Truslet Policy contains a version (must be 1) and the Trustlet ID of the process
  - This must match the Secure Process Process Attribute that was passed to *CreateProcess*
- The rest of the policy contains entries for different kinds of features that the Trustlet is allowed to use
  - EtwAllowed – Sets flag 0x20 in the SKPROCESS structure
  - Blob is saved at offset 0xC0 in SKPROCESS, to be checked later
  - Options include DebugEnabled, CrashdumpEnabled, CrashdumpKeyGuid, CrashdumpKey
- Checked at runtime by *SkpspFindPolicy*
  - *SkpsIsProcessDebuggingAllowed*, *SkpsIsProcessDumpEnabled*, *SkpsGetTrustletCrashdumpKey*

TRUSTLET INSTANCE GUID

- Trustlets can also associate an Instance GUID with their process
  - The same Instance GUID can be used by multiple Trustlets (with different IDs), including Trustlets that are running on different partitions
- Instance GUIDs are tied with Hyper-V Partition GUIDs
  - Allows a Trustlet on a child Hyper-V Partition to identify itself to SKM, as well as a Trustlet on the host partition to identify itself with SKM
  - Can now use the Secure Storage facility
- Instance GUID is set with *SetTrustletInstance*
  - This is then stored in the SKPROCESS structure at offset 0x10

INBOX TRUSTLET IDS AND INSTANCE GUIDS

- Trustlet ID 0 is the Secure Kernel Process, which hosts Device Guard
- Trustlet ID 1 is LSAISO.EXE, which hosts Credential Guard
- Trustlet ID 2 is VMSP.EXE which hosts the Virtual TPM (vTPM) on the Host Side
  - Uses the Instance GUID of the Hyper-V partition it is associated with to *pull* TPM secrets
- Trustlet ID 3 is the vTPM provisioning tool
  - Uses the Instance GUID of the Hyper-V partition it is associated with to *push* TPM secrets
  - Enables Guarded Fabric feature

LSA TRUSTLET

- One of the two inbox Trustlets on the system, LSAISO.EXE provides the *Credential Guard* feature of Windows 10
- Uses the *DecryptData* and *EncryptData* IUM Base API and nothing else
- Communicates to the outside world using RPC
  - Its agent is LSASS.EXE, which runs as usual in Normal Mode
- Provides support for encrypting Kerberos and NTLM secrets without visibility to an outside attacker without hypervisor/hardware exploits, as well as using Kerberos FAST to securely store and seal a Machine Key to prevent replay attacks and guarantee the provenance of the TGT

VIRTUAL MACHINE SECURE WORKER PROCESS

- Hyper-V architecture separates the work required to perform VM emulation for each partition into two categories
  - **Enlightened I/O** which is handled by VSP and VSC pairs (Virtualization Service Providers/Clients) that are typically kernel-mode drivers on both the host and the client
  - **Legacy I/O** which is handled by Virtual Machine Worker Processes (VMWP.EXE)
    - Emulates standard i440BX motherboard legacy devices
    - Also handles services such as RDP, clipboard, etc…
- With vTPM, a partition has a Truslet process as well – VMSP.EXE
  - Its agent is the VMWP.EXE process mentioned above
- VMSP Uses the Mailbox and Secure Storage interface to store and encrypt TPM secrets using the IDK
  - Mailbox provides things like a “Security Cookie” to authenticate communications
  - Secure Storage is used for Ingress, Egress and other cryptographic keys for vTPM

LOADING A TRUSTLET

- Ldr, or the **Windows Loader** (inside of NTDLL.DLL), will load a Trustlet just like any other process, with a few changes in behavior
  - These are detected through the PEB's Process Parameter Flags (0x80000000 == RTL_USER_PROC_SECURE) and stored in *LdrpIsSecureProcess*
  - No per-user **Application Verifier** support
    - System-wide settings still respected
  - No **Image File Execution Options (IFEO)** if queried by *LdrQueryImageFileExecutionOptions*
    - *RtlQueryImageFileExecutionOptions* still respected (but no registry APIs)
  - No **DLL Redirection (Side-by-Side Manifest File Support, SxS, Fusion)**
  - No communication with **CSRSS (Windows Subsystem)** allowed
  - No **Safer (Authz) / Software Restriction Policies** enforced
- This allows Trustlets not to have to share other data with Normal Mode (such as CSRSS data) or to have their behavior influenced by it
  - Like the Shim Engine

FAKE BASE SERVER CONNECTION

- Without talking to CSRSS, a special structure called BASE_STATIC_SERVER_DATA is never filled out
  - It is normally stored in the PEB's *ReadOnlyStaticServerData* field
  - It contains the Windows installation directory, Object Manager paths, Timezone information, and more
- *CsrpLocalSetupForSecureProcess* hacks around the connection requirement by manually filling out these fields locally
  - Some fields are never filled out (probably not used by current Trustlets)
- The data comes from a heap allocation local to the process
  - Therefore, instead of an unmodifiable, read-only shared memory section, Trustlets trust their own read-write data on the heap
- Attempts to connect to servers other than Server 1 (Base Server) will fail
  - /SUBSYSTEM:CONSOLE images will attempt to connect to the Console Server (or Console Driver) and **cannot be loaded as Trustlets**

TRUSTLET TO NORMAL WORLD COMMUNICATIONS

- Trustlet communication to the outside world can be achieved through a variety of means
  - ETW Tracing, as long as the Trustlet Policy allows it (including the ability to use WPP)
  - The *DbgPrint* API, albeit only on checked builds (the *IumDebugPrintNt* API is empty in release builds)
  - Local RPC by using the ncalrpc transport (which uses NTOS' ALPC infrastructure)
- Out of these, Local RPC is the only non-debugging, fully supported solution (works regardless of Trustlet Policy)
- It also means that ALPC marshalling vulnerabilities, or LRPC runtime/marshalling vulnerabilities, or RPC message parsing vulnerabilities, will be the likely key ways of breaking into a Trustlet from HLOS
  - However, you would then also need the ability to execute code 'remotely' in IUM, and bypass any HVCI requirements depending on configuration
  - And then you would need an IUM->SKM vulnerability to be able to attack arbitrary Trustlets (if the goal was to attack Shielded VMs, for example)

TRUSTLET ALPC ENDPOINT CONNECTIONS

- Each Trustlet can create its own ALPC / RPC Endpoints
  - For example, LSA Trustlet (LSAISO.EXE) creates **LSA_ISO_RPC_SERVER**
- However, when a Trustlet crashes, the Windows Error Reporting Service (WER) creates a special on-demand ALPC port **\IUM_TRUSTLET_DUMP_SERVER**
  - The SK will connect to this (*SkpAlpcConnectWerPort*) whenever it crashes, and send the encrypted pages to it
  - WER will write the encrypted pages to disk (IUM-name-date-time-pid.dmp) that are sent via *SkpAlpcSendReceiveWer*
    - One start message
    - N chunk messages
    - One end message
- One could arguably use this interface from a Trustlet as a way to write arbitrary files to disk, but the location/name is somewhat limited ☺

CAN WE BUILD OUR OWN TRUSTLETS?

- Microsoft has done a phenomenal job to provide new hypervisor-based guarantees to certain key mechanisms and features
- Architecturally speaking, the security model is very strong
- But could it be used against itself? Can we create malicious Trustlets that are protected?
  - We'll call these ***Malwarelets***
- There are a number of protections and checks in place to prevent this from happening
  - We need a correctly-formatted policy with our own Trustlet ID, exported with the right name in a section with the right name and attributes
  - We need a correctly-signed Trustlet image
  - We must link with /SUBSYSTEM:WINDOWS (but we still need to tell CRT to give us a wmain, not WinMain)
  - We must be careful what import libraries we use (and hence what DLLs we import)
- And ultimately, a Trustlet is isolated from other Trustlets, and cannot access HLOS in any privileged way

DEMO

WORLD'S FIRST 3RD PARTY TRUSTLET

CONCLUSION

RISKS AND THOUGHTS

IS IUM/VSM SECURE…? YES!

- There do not appear to be any design flaws in the implementation of IUM/VSM
- Good decisions made all around regarding separation of privileges, roles, and powers
  - Good analogy was made yesterday in the Microsoft presentation with government judicial, executive, and legislative branches
- Extremely limited attack surface
  - Hypervisor is already a very thin piece of code. VTL has added complexity, but is manageable
    - No true hypervisor non-DoS bugs have ever been found (only virtualization component vulnerabilities)
  - SKM is ~350KB and provides limited system calls and accessible interfaces from VTL 0
  - SKCI does do ASN.1 parsing, but code has been tested for decades and is a minimalistic implementation
- However, true VSM security relies on platform features to be present, and well-implemented

SECURE KERNEL COMPLEXITY / ATTACK SURFACE

- Note that outside of the Secure Mode Calls described, the Secure Kernel does not do much on its own
  - All scheduling, memory management (paging/etc) and other resource management is done by NTOS
  - No hardware drivers / devices implemented
  - System calls are “proxied” through Normal Mode Calls
- SKM doesn’t “trust” NTOS, but it does depend on it
  - So Denial-of-Service attacks are possible, but information disclosure would require a pretty severe vulnerability
- You also cannot “nop out” calls to SKM in order to attack the system
  - For example, since SKM owns the EPT mappings and secure image relocations, if you try to force the system to load an unsigned driver (bypassing Device Guard), the code will simply not execute, since the EPTs will not have the executable bit set
  - If you force loading an unsigned Trustlet, default settings will allow it to run (since it’s user-mode code), but it won’t actually have access to any SKM facilities (SKM will not have a secure image object associated with it)

COMPROMISING VBS / MISUSING VSM

- VBS' key weakness is its reliance on Secure Boot
  - Many vendor implementation and design bugs *[ref: Sith Strike]*
  - Many flash/SPI/chipset related issues *[ref: Legbacore/Corey/Xeno's 100 talks in the last decade]*
  - Option ROMs *[ref: Thunderstrike]*
  - Secure Boot vulnerabilities can bring down the entire security model
- Even worse, a compromised VSM can be used against the user
  - A compromised VSM can allow Malwarelets to execute, which can now provide easier access to all of the isolated secrets, but additionally can also use their own side-channel and secrets now, against the admin itself
  - Combined with protected processes, can create non-debuggable, inaccessible, unkillable implants protected by the hypervisor itself
  - **This is not an inherent/new risk to VSM! Implants today can load their own hypervisor and build their own VSM – but this now makes it easier to hide within Microsoft's own infrastructure**
  - **Recall: Trustlets cannot access files, or the registry, or NTOS/HLOS data**

VSM WITHOUT SECUREBOOT

- What's worse, however, is that VSM can be configured to work without Secure Boot
  - As evidenced on my laptop, even if Secure Boot is part of the platform requirements, VSM still loaded SKM and IUM even with it turned off
- HVCI does not perform and self-integrity checks and there is no "Hyper Guard"
  - I am booted with an SKCI.DLL binary that has a single byte patch – machine has been up for weeks
  - The boot loader allows corrupt / self-signed SKM binaries if test mode is turned on
    - Unlike SKCI which does not allow IUM binaries signed by non-Microsoft roots, even in test mode
- SKM does not have a hard-coded list of which Trustlet IDs are valid, or what their hashes should be
  - Arbitrary Trustlet Identities are allowed, once code signing checks are authorized
- Again, without Secure Boot, an attacker can build their own VSM-like hypervisor and own the machine, so these are not VSM "flaws"
  - But they make it a lot easier to abuse VSM on what is likely to be still a large number of in-the-wild machines

RECOMMENDATIONS

- Microsoft should at the very least make HVCI (SKCI.DLL) verify its own integrity
  - CI.DLL does this, and is additionally protected by PEAUTH.SYS, PatchGuard, and Warbird
- Even better, the Hypervisor (or an SKM component) should ensure that code pages are signed according to expected rules, and crash the machine if not
  - For example, IUM code should never be associated with a non-Microsoft root
- SKM should disallow launching Trustlet IDs that it doesn't know about
  - It already hardcodes Trustlet IDs in the SKM Capability Definitions
  - Microsoft won't be releasing Trustlets on a frequent basis – and when it does, it can update SKM
- Better yet, actually *enforce* Secure Boot to be enabled, turned on, and with the expected certificates loaded, before allowing VSM to enable itself
  - Force the attacker to write their own hypervisor, not use yours
- These are all defense-in-depth measures – obviously no way to truly protect against a physical attacker without Secure Boot

YOU HAVE QUESTIONS?

I HOPEFULLY HAVE SOME ANSWERS

Sincere thanks to the following people, without whom this presentation would not have been possible:

- Arun Kishan, Dave Probert, Alain Gefflaut, David Hepkin, Seth Moore, Baris Saydag, Bruce Dang, Dave Mitduri, Jason Shirk