

Redefining Security Boundaries: Unveiling Hypervisor-Backed Security Features For Windows Security

CONNOR MCGARR

SANS HACKFEST HOLLYWOOD 2024

About Me

- Security Researcher @ Prelude, a security startup
- Windows Internals, C, ASM, insert more fluff here...
- [@33y0re](#) && <https://connormcgarr.github.io>

Agenda

- Legacy Exploits and Techniques
- Another Security Boundary – Virtual Trust Levels
- Hypervisor-Backed Security Features

The slide features a dark blue background with decorative teal lines. On the left, several parallel lines extend vertically. On the bottom right, several parallel lines extend diagonally.

Legacy Exploits and Techniques

SANS HACKFEST HOLLYWOOD 2024

- Kernel exploits/vulnerabilities/etc. are old as time
 - Well, maybe not *that* old – but still pretty old!
- At one point this topic was not as documented or popular as it is now
 - To-date there are copious amounts of talks, articles, exploits, POCs available online
- Vulnerabilities reside in various kernel components
 - I/O handlers (many times in 3rd party driver IOCTL handlers)
 - Native OS components which process input
 - System calls, network components, file system filters, etc.
 - Etc.

- With legacy exploitation things used to be crazy!
 - Executable pool memory
 - No enforcement of execution environment
 - E.g., CPL is kernel mode, but instructions being executed can reside in user-mode
 - Little (if any, in some cases) address space randomization
 - No control-flow validation
 - E.g., are you sure “call qword ptr [rax]” is a legitimate target?
 - No enforcement of memory permission boundaries
 - What’s stopping an attacker with a kernel vulnerability from making writable kernel memory executable?

- It turns out defending kernel-mode is hard!
 - Using user-mode as an example – a mitigation like DEP can be enforced and protected by the user-mode/kernel-mode boundary
 - You can circumvent DEP by calling into VirtualProtect (as an example) but you still must ask the kernel to do this for you – which means you have some execution primitive
- Kernel-mode mitigations are a different story
 - If an attacker has a read/write primitive into kernel-mode, there is no other boundary which an attacker must “ask” to manipulate memory
 - Attackers in kernel-mode have access directly to the page tables and can manipulate memory any way they see fit with just a read/write primitive
 - There is nothing to enforce that the memory *must* be read-only because a kernel-mode attacker doesn’t have to ask for memory to be manipulated – it can be done directly without an API!

- First attempts at hardware-based protection – Supervisor Mode Execution Prevention (SMEP)
 - CPU-enforced mitigation to validate execution environment
 - Common technique at the time is:
 1. Allocate and store kernel-mode shellcode in user-mode
 2. Corrupt kernel-mode function pointer/return address using an exploit
 3. Call the corrupted address, redirecting execution to the user-mode allocation
 - Privilege Level of the instruction pointer is kernel mode, but instructions are in user-mode
 - Code is executed as kernel-mode, giving access to privileged instructions, etc.

Shellcode

0x1ad00000

User Mode

Kernel Mode

0xffffffff76011223300

0x1ad00000

0xffffffff76011223300

0xffffffff76011223300

Function Pointer Array

RIP:

call function_pointer_array[1]


- SMEP is a CPU feature which detects when a user-mode PTE is accessed while the current privilege level (CPL) of the CPU is operating in kernel-mode
 - This mitigates the previous scenario!
 - ...well, not entirely
 - Easily circumvented in a few ways
 - SMEP enablement is managed by a bit in a control register which can be cleared
 - PTE corruption
 - SMEP implements a special PTE bit (User/Supervisor)
 - Nothing stops an attacker from flipping the bit in a user-mode PTE to supervisor *before* redirecting execution – effectively “tricking” the CPU into executing user-mode code in a privileged fashion
 - Although SMEP was an improvement – it quickly proved to not be a robust solution

- Defenders learned that it basically “all boils down” to this:
 - If any kernel-mode mitigation is implemented in kernel-mode it assumes the integrity of the kernel is in-tact
 - However, mitigations are normally threat modeled assuming an attacker already has kernel-mode read/write access
 - If an attacker has kernel-mode read/write access, and a mitigation trying to defend against this attacker is ALSO in kernel-mode you have a situation now where the kernel is “defending against itself”
 - Answer: We need a more trusted entity than the kernel to guarantee it's integrity!

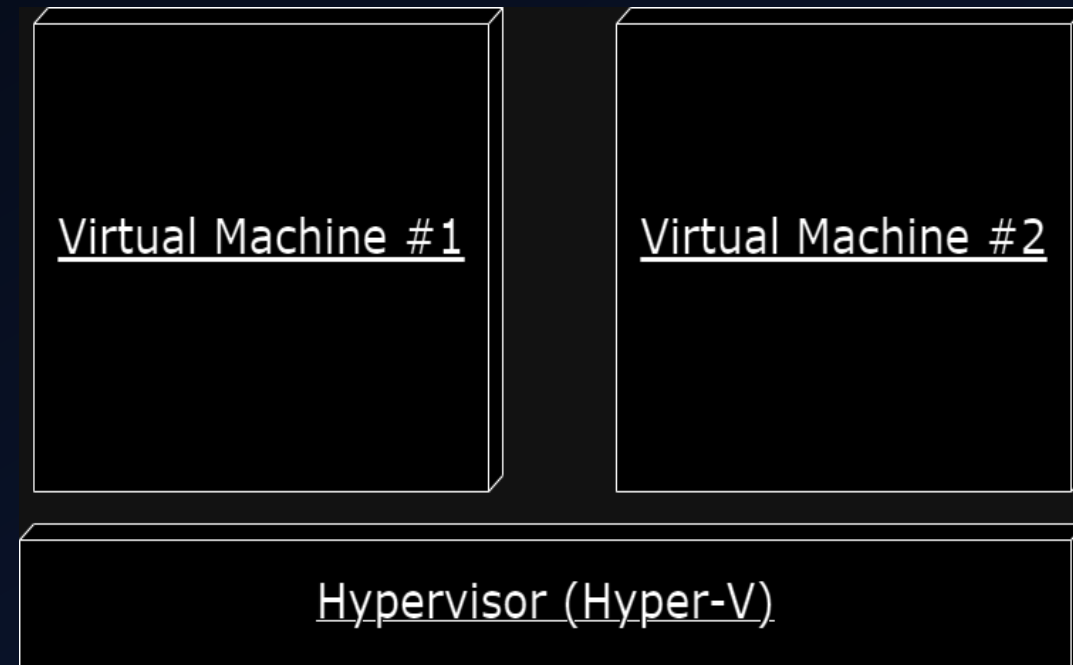


Another Security Boundary – Virtual Trust Levels

SANS HACKFEST HOLLYWOOD 2024

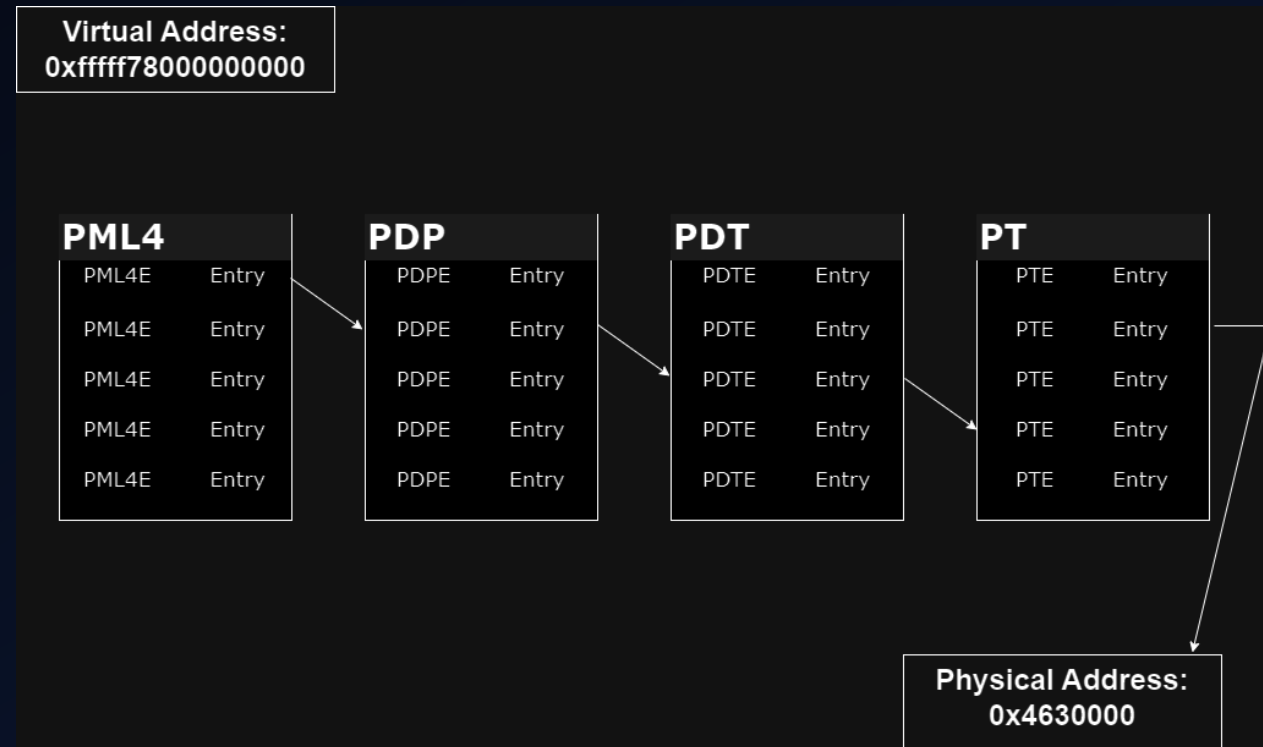
- 
- Microsoft (very cleverly!) leverages existing virtualization technologies to instrument a higher security boundary than the kernel!
 - Let's talk briefly about some of the technologies which help facilitate this!

- Virtual machine isolation
 - Virtual machines really are nothing more than an isolated set of resources!
 - VMs are not “isolated” because of security-related features – they are isolated because of how a hypervisor can *fundamentally* configure memory!
 - The hypervisor (Hyper-V in this case) can partition memory for each VM and configure it in such a way that the memory is only valid “in context” of the target VM
 - This technology is known as Second Layer Address Translation (SLAT)
 - *Intel’s implementation of SLAT is known as Intel Extended Page Table (EPT)



- To grasp how hypervisors can isolate memory its worth first mentioning virtual memory in general, as well as how the user-mode address space itself is isolated on Windows
 - Modern operating systems implement *virtual memory*
 - This allows the OS to work under the “illusion” that much more memory is available for use than is physically installed because memory can be flushed out to disk if not in use with the implementation of virtual memory
- At a high level:
 - A set of *page tables* are used to track which virtual memory is mapped to which physical memory
 - Specifically, the page tables are made up of *page table entries* (PTEs) which are used to perform translation and enforce various information about the target memory
 - PTEs denote if the memory is even valid, if its readable, writable, and even which physical memory backs the page

- Example: Consider a write operation to a virtual memory address
 - Virtual memory is effectively just an OS-created “abstraction”
 - The contents of the write need to eventually be committed to *physical* memory
 - As part of the operation, the MMU needs to translate the virtual address to its backing physical address
 - This translation is completed by leveraging the *page tables* (*although a cache exists which is not currently relevant to us)



- User-mode virtual address space isolation is facilitated using nothing more than the page tables!
 - Each process receives a set of page tables which contain mappings specific to that process. Attempts to access memory in a *different* process will fail simply because the page tables don't contain the mappings!
 - This allows, for example, virtual address **0x1ad00000** in **ProcessA.exe** to be either invalid in **ProcessB.exe** or to be mapped to entirely different physical memory!

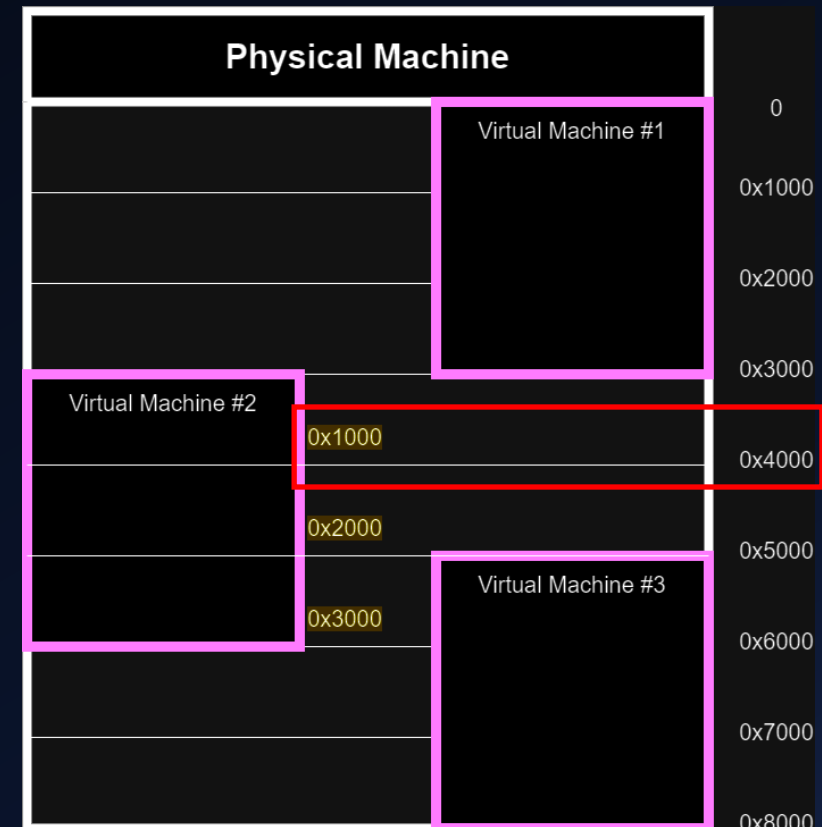
```

Command X
0: kd> dx @$curprocess.Name
@$curprocess.Name : services.exe
Length : 0xc
0: kd> r cr3
cr3=00000000ce5c6000
0: kd> !vtop 0xc5c6000 0x7ff7228a0000
Amd64VtoP: Virt 00007ff7228a0000, pagedir 00000000ce5c6000
Amd64VtoP: PML4E 00000000ce5c67f8
Amd64VtoP: PDPE 00000000cc1d2ee0
Amd64VtoP: PDE 00000000ce5d38a0
Amd64VtoP: PTE 00000000c9fd4500
Amd64VtoP: Mapped phys 00000000cc480000
Virtual address 7ff7228a0000 translates to physical address cc480000.
  
```

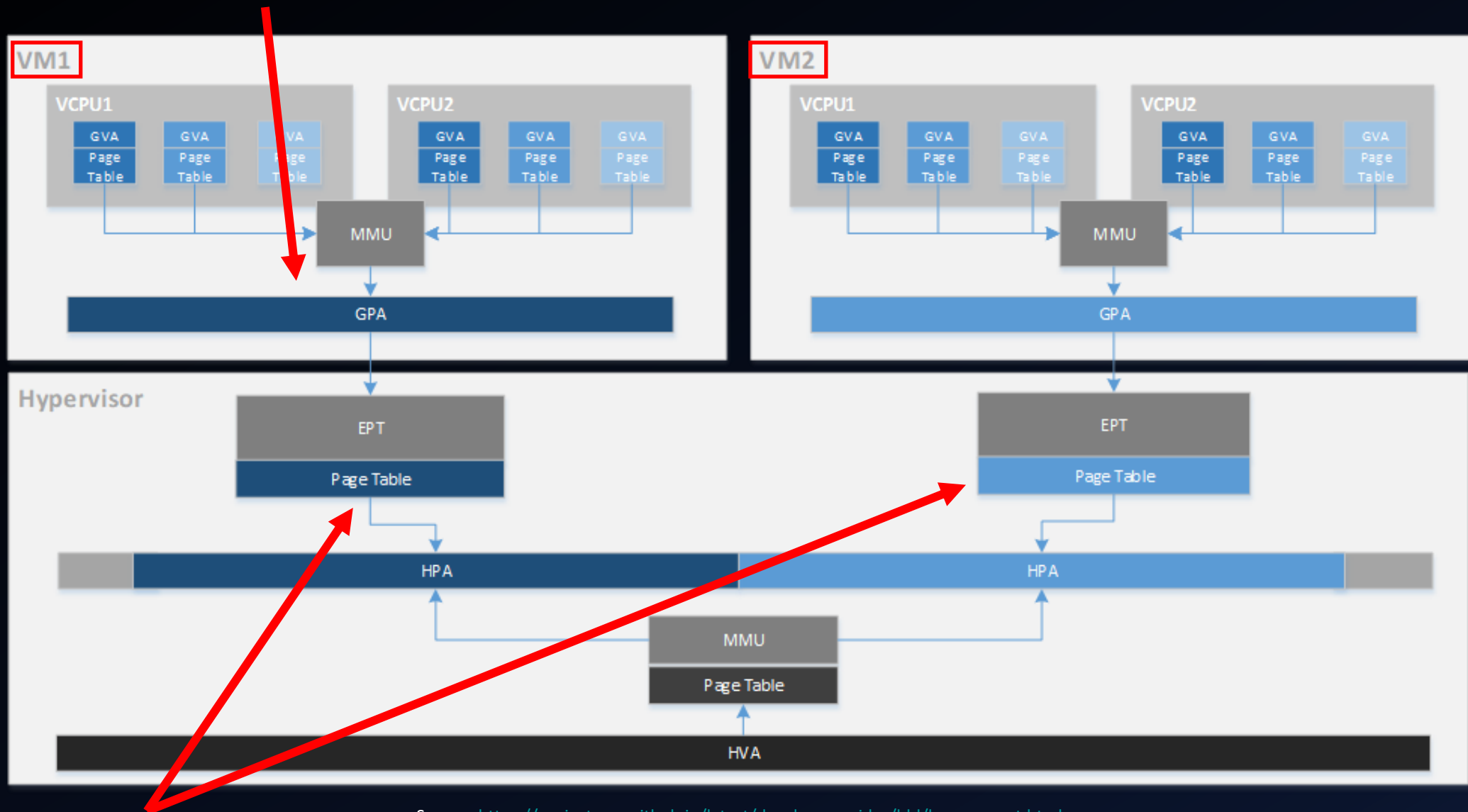
```

Command X
0: kd> dx @$curprocess.Processes.Where(p => p.Name == "SearchIndexer.exe").Select(p => p.KernelObject.Pcb.DirectoryTableBase)
@$curprocess.Processes.Where(p => p.Name == "SearchIndexer.exe").Select(p => p.KernelObject.Pcb.DirectoryTableBase)
[0x14d0] : 0x95426000 Type: unsigned __int64
0: kd> !vtop 0x95426000 0x7ff7228a0000
Amd64VtoP: Virt 00007ff7228a0000, pagedir 0000000095426000
Amd64VtoP: PML4E 00000000954267f8
Amd64VtoP: PDPE 0000000095432ee0
Amd64VtoP: zero PDPE
Virtual address 7ff7228a0000 translation fails, error 0xD0000147.
  
```

- Analogy: Just as per-process page tables (KPROCESS->DirectoryTableBase) give us UM virtual address space isolation, per-VM page tables give us *VM* address space isolation!
 - Virtualization gives us two sets of page tables – the “system” page tables and the “extended page tables”
 - VMs produce an “intermediary” address known as a “guest physical page” which is translated into *actual* physical memory.
 - The extended page tables (EPTs) contain the necessary information to turn a guest page into real physical memory
 - Each VM has a set of EPTs



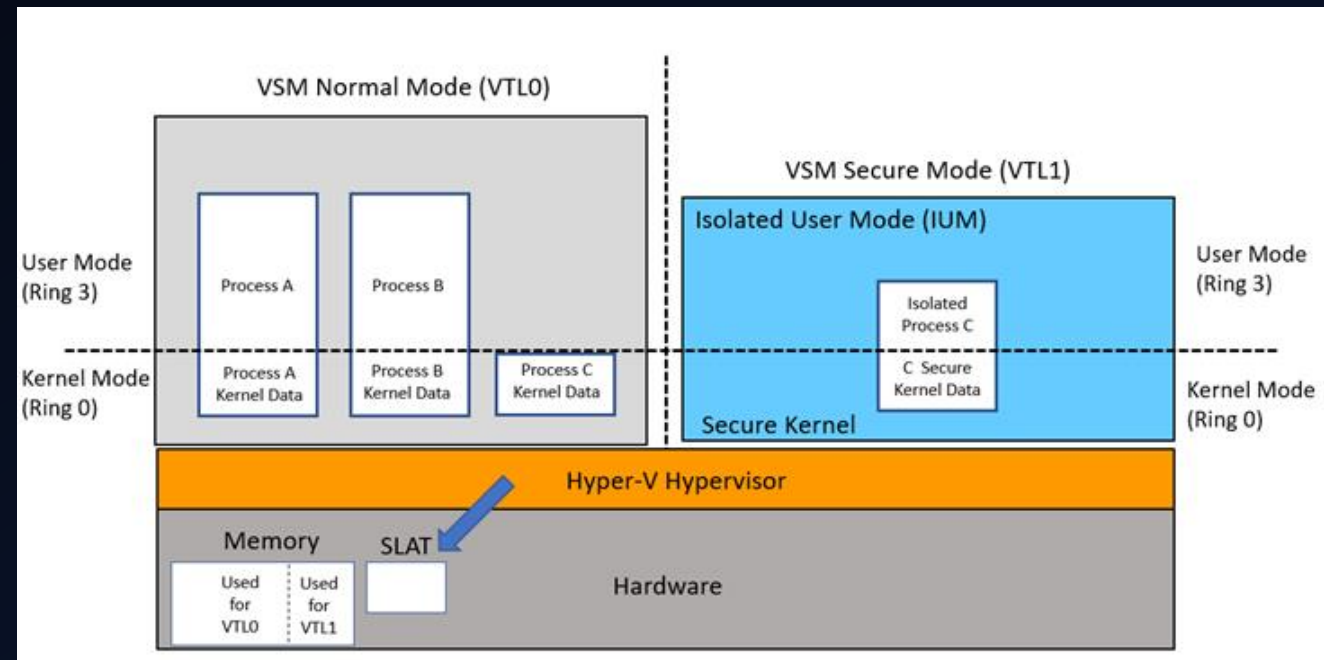
VA access in a VM results in a GPA



Per-VM EPTs are used to fetch SPA
(HPA in the diagram)

Source: <https://projectacrn.github.io/latest/developer-guides/hld/hv-memmgr.html>

- Why do we care about any of this?
 - This same technology is what facilitates Virtualization Based Security (VBS) on Windows!
- VBS instruments *Virtual Trust Levels* (VTLs) which effectively act as a sort of “VM” in the sense that a VTL is nothing more than an isolated address space (like a VM!)
 - VTL 1 (Secure) is more privileged than VTL 0 (normal) and VTL 1 can “impose its will” on VTL 0



- VTL 1 and VTL 0 each have a set of EPTs (as they are our “VMs”*) – this allows them to remain isolated, just as two VMs are isolated
 - ...but the actual EPTs (memory) in each VTL are mapped to the *actual* physical frames on the machine (known as identity mapping)
 - E.g., When VTL 0 emits GPA 0x1000 it is located at the *real physical address* 0x1000
- If GPAs just “forward” to real physical memory, why do we need the EPTs if they aren’t being used for “translation” in the “traditional sense”?
 - With VBS the EPTs are used to configure *two different* access rights on physical memory! Here is how this works.

- VTL 1 is the address space which is treated as “Secure world”
 - It is treated as a higher security boundary than VTL 0
- VTL 0 is where users operate (traditional user-mode/kernel-mode)
- VTL 1 is allowed to configure the EPTs for VTL 0 as it sees fit by asking the hypervisor to do so through a *hypercall*
 - This results in VTL 1 being able to set any desired access rights on the EPTs associated with VTL 0 (remember page tables are made up of PTEs, which denote if memory is readable, writable, etc.)
 - In addition, VTL 1 can ask the hypervisor to mark some regions of memory as totally *invalid* within the context of VTL 0 via the EPTs
 - These configuration options allow VTL 1 to apply immutable access rights on VTL 0 and to completely isolate sensitive regions of memory away from VTL 0. **THIS IS THE BASIS FOR MUCH OF VBS AND ITS FEATURES!**

- Quick example to “put it all together”
 - HVCI – Hypervisor Protected Code Integrity
 - W^X (write XOR execute) implementation in kernel-mode
 - Memory effectively can never be both writable and executable (pre-requisites for unsigned code execution) nor can RWX memory be dynamically generated!
- How does it work?
 - The EPTs for VTL 0 are configured (by VTL 1) to never allow PTEs that are both W and X!
 - Consider when a kernel exploit corrupts a writable page’s PTE to mark it as read/write/execute (RW -> RWX).
 - With HVCI:
 - CPU accesses memory to execute. It checks that the PTE has the eXecute bit set
 - Since SLAT is present, that memory is treated as a guest page. The extended page table entry (EPTE) for the guest page is then fetched and checked.
 - The EPTE was originally configured by VTL 1 to only be RW. It never contained the executable bit **EVEN THOUGH THE VTL 0 PTE SAYS IT IS EXECUTABLE!**
 - Even though the VTL 0 PTE says eXecute, the EPTE takes precedence, which says the memory is **NOT executable**. The memory therefore won’t be executed!

- This makes unsigned code execution effectively impossible with HVCI enabled!



- VTL Communication Channels
 - VTL 0 can request the services of VTL 1, and vice-versa
 - VTL 0 -> VTL 1 – Secure System Call (vmcall opcode)
 - VTL 1 -> VTL 0 – Normal Call (not talking about this today!)
 - Secure System Calls
 - A particular secure system call number (SSCN), like a traditional system call number, is provided to denote what resource is being requested
 - Additionally, arguments are packed in a particular structured format
 - Secure Kernel receives the request in its handler function and determines how to handle the call (securekernel!lumInvokeSecureService) and then safely parses and copies the arguments
 - Calls are made for various reasons, especially for letting VTL 1 know about something happening in VTL 0

- securekernel!IumInvokeSecureService

```
switch (secureSystemCallNumber)
{
case 1u:
    v37 = __rdtsc();
    if ( _InterlockedCompareExchange(&g_ExpectedIumInitializationPhase, 3, 2) != 2 )
    {
        initied = -1073741790;
        goto LABEL_277;
    }
    SkmmInitializeUserSharedData();
    initied = SkInitSystem(4LL, 0LL);
    if ( initied < 0 )
        goto LABEL_277;
    *(a1 + 16) = SkCheckHibernationSupport();
    _InterlockedIncrement(&g_ExpectedIumInitializationPhase);
    SkWritePerfTraceEntry(0LL, 4LL, __rdtsc() - v37);
    goto LABEL_53;
case 2u:
    started = IumpStartProcessor(a1 + 8);
    goto LABEL_267;
case 3u:
    started = IumpFinishStartProcessor(a1 + 8);
    goto LABEL_267;
case 4u:
    initied = SkUpdateUserSharedDataSystemRoot();
    if ( initied < 0 )
        goto LABEL_53;
    initied = SkpgConnect(0, 0, 1, 0, 0LL);
    if ( initied < 0 )
        goto LABEL_53;
    started = SkpsRegisterSystemDlls(*(a1 + 8), *(a1 + 16));
    goto LABEL_267;
case 5u:
    if ( _InterlockedCompareExchange(&IumpSystemProcessRegistered, 0, 1) )
        goto LABEL_255;
    SkeLockProcessorStartup();
    started = SkpsRegisterSystemProcess(*(a1 + 8), *(a1 + 16), a1 + 24);
    goto LABEL_267;
case 6u:
    initied = SkpsCreateProcess(*(a1 + 8), *(a1 + 16), &v67);
    if ( initied >= 0 )
        *(a1 + 16) = v67;
    goto LABEL_53;
case 7u:
    started = IumpInitializeProcess(a1 + 8);
    goto LABEL_267;
```



Hypervisor-Backed Security Features

SANS HACKFEST HOLLYWOOD 2024

- Using the hypervisor as a higher security boundary than the kernel, we can do some cool things
- Example #1: Kernel Data Protection (KDP)
 - Microsoft blog by Andrea Allievi: <https://www.microsoft.com/en-us/security/blog/2020/07/08/introducing-kernel-data-protection-a-new-platform-security-technology-for-preventing-data-corruption/>
 - Static KDP
 - Instrumented through the NT function nt!MmProtectDriverSection
 - Allows a kernel-mode driver to guarantee read-only access to a particular memory region
 - Sets a VTL 0 read-only SLAT entry which is managed by the hypervisor (not accessible in NT of VTL 0)

```
NTSTATUS MmProtectDriverSection(  
    [in] PVOID AddressWithinSection,  
    [in] SIZE_T Size,  
    [in] ULONG Flags  
);
```

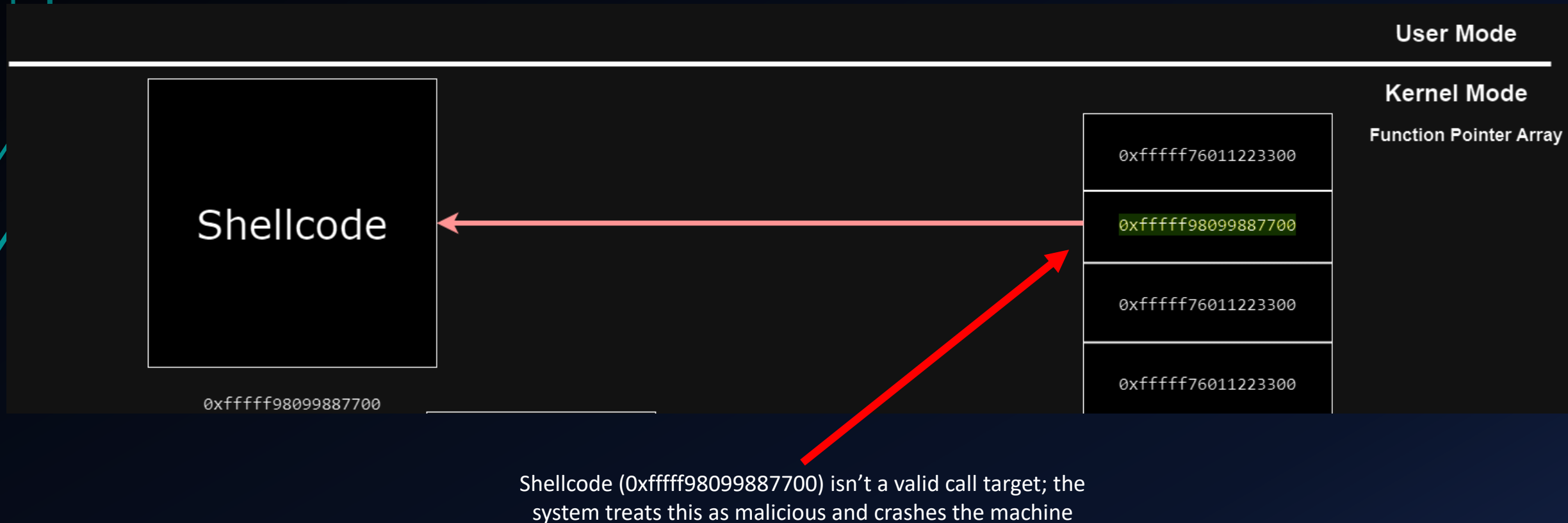
- Static KDP (Cont'd)
 - An example of this in action is protection of the global variable `CI!g_CiOptions` which is commonly abused to disable signing requirements to load arbitrary unsigned-drivers
 - Static KDP is used to protect this global variable, making it read-only at the EPT level!

```
g_CiCurrentBootId = bootId;
status = CiBlackBoxInitialize();
if ( status >= 0 )
{
    CipInitializeRevocationListInfo(a1);
    if ( (g_CiOptions & 0x10) != 0 || KdDebuggerEnabled != 1 || KdDebuggerNotPresent )

        // nt!MmProtectDriverSection(&g_CiOptions, 0, 1);
        MEMORY[0xFFFFF807D22279A0](&g_CiOptions, 0LL, 1LL);
    *ProtectedContent = &g_CiProtectedContent;
}
```

- Dynamic KDP (Secure Pool)
 - Effectively extends (literally, using extended parameters) KDP to dynamic memory allocations
 - Instead of just static data inside a compiled image being protected by the hypervisor, dynamically-allocated pool memory can also reap the benefits by allocating read-only memory protected by SLAT!
 - Some slight nuance since pool memory is constantly allocated/freed – but you get the point!
 - Yarden Shafir's blog: <https://windows-internals.com/secure-pool/>
 - **ExCreatePool** -> **ExAllocatePool3** -> **ExSecurePoolUpdate** are the pertinent APIs

- Kernel Control Flow Guard (KCFG)
 - Kernel-mode implementation of CFG – an older technology at this point (shipped as optional on Windows 8.1)
 - Indirect function calls (a call to a target determined at runtime) are verified that they were known about at compile time using a special bitmap



- Off when HVCI is disabled as the bitmap, which is the source of truth, is susceptible to an existing attacker in the kernel corrupting it (like the PTEs before HVCI)
 - “Kernel defending against itself” scenario
 - Technically KCFG still acts as a type of software SMEP even with HVCI disabled to verify at least a kernel call isn’t calling out to user-mode code
- VTL 1 marks the bitmap (located in VTL 0) as read-only in the EPTes. This means even if the bitmap is made read/write in the VTL 0 PTE, the bitmap is STILL read-only in the eyes of the hypervisor

```
// extendedPageProtectionFlags & 0x80 == HVCI
if ( (extendedPageProtectionFlags & 0x80u) != 0LL )
    // MI_FLAGS.SecureKernelCfgEnabled = 1
    MiFlags = (miFlags | 0x40000);
return extendedPageProtectionFlags;
```

Kernel CFG is enabled only if
HVCI is supported and running

- The hypervisor (once again) provides another security boundary which allows KCFG to actually be effective when it otherwise wouldn't

```
Command X
0: kd> u nt!guard_dispatch_icall
nt!guard_dispatch_icall:
fffff806`46234de0 4c8b1d191b9e00 mov     r11,qword ptr [nt!guard_icall_bitmap (fffff806`46c16900)]
fffff806`46234de7 4885c0      test    rax,rax
fffff806`46234dea 0f8d7a000000 jge     nt!guard_dispatch_icall+0x8a (fffff806`46234e6a)
fffff806`46234df0 4d85db      test    r11,r11
fffff806`46234df3 741c       je      nt!guard_dispatch_icall+0x31 (fffff806`46234e11)
fffff806`46234df5 4c8bd0      mov     r10,rax
fffff806`46234df8 49c1ea09    shr     r10,9
fffff806`46234dfc 4f8b1cd3    mov     r11,qword ptr [r11+r10*8]
0: kd> !pte nt!guard_icall_bitmap
                                VA fffff80646c16900
PXE at FFFF9C4E27138F80  PPE at FFFF9C4E271F00C8  PDE at FFFF9C4E3E0191B0  PTE at FFFF9C7C032360B0
contains 000000000118B063 contains 000000000118C063 contains 00000000016A0063 contains 8900000005401963
pfn 118b      ---DA--KWEV pfn 118c      ---DA--KWEV pfn 16a0      ---DA--KWEV pfn 5401      -G-DA--KW-V
0: kd> ep nt!guard_icall_bitmap 9090909090909090
^ Memory access error in 'ep nt!guard_icall_bitmap 9090909090909090'
```

Kernel CFG bitmap cannot be written to from VTL 0 even if the VTL 0 PTE is corrupted to say it's writable!

- Kernel CET
 - Kernel-mode implementation of Intel CET
 - Intel CET (the shadow stack feature, Indirect Branch Tracking is not used by Windows) has a hardware-based shadow stack which is used to validate return addresses on the stack have not been corrupted
- Although CET is a hardware mitigation, the kernel-mode implementation still relies on VTL 1 (which is why it is worth talking about)

To use Kernel-mode Hardware-enforced Stack Protection, you must have **Memory Integrity** enabled, and you must be running a CPU that supports **Intel Control-Flow Enforcement Technology** or **AMD Shadow Stack**.

- Shadow stacks, even though they are hardware-protected, are still allocated by software
 - For kernel-mode CET shadow stacks are allocated using the KernelStackPteInfo system PTE region on thread creation (and some other circumstances, like stack expansion)
 - KernelStackPteInfo[0] = “normal stack” region
 - KernelStackPteInfo[1] = “shadow stack” region

```
Command X
kd> dx @$miState->SystemPtes->KernelStackPteInfo[1]
@$miState->SystemPtes->KernelStackPteInfo[1] [Type: _MI_SYSTEM_PTE_TYPE]
[+0x000] Bitmap [Type: _RTL_BITMAP_EX]
[+0x010] TrackingBitmap [Type: _RTL_BITMAP_EX]
[+0x020] BasePte : 0xffffffff7e40000000 [Type: _MMPTE *]
[+0x028] Flags : 0x2 [Type: unsigned long]
[+0x02c] VaType : MiVaKernelShadowStacks (15) [Type: _MI_SYSTEM_VA_TYPE]
[+0x030] PteFailures : 0x0 [Type: unsigned long]
[+0x034] SpinLock : 0 [Type: long]
[+0x038] TotalSystemPtes : 0x200 [Type: unsigned __int64]
[+0x040] Hint : 0x673a12 [Type: unsigned __int64]
[+0x048] LowestBitEverAllocated : 0x673a00 [Type: unsigned __int64]
[+0x050] CachedPtes : 0xfffffd981800002a8 [Type: _MI_CACHED_PTES *]
[+0x058] TotalFreeSystemPtes : 0x1ee [Type: unsigned __int64]
```

- To protect modification from VTL 0, a secure system call is made upon shadow stack allocation which marks the shadow stack as read-only in VTL 0 to prevent unauthorized modification
 - Legitimate modification can occur, as an example, on longjmp or thread context update
 - NT delegates these situations to the Secure Kernel through an “assist” function
 - **The basic premise is that only hardware or VTL 1 should manipulate the shadow stack!**

```
bt cs:MiFlags, 29h  
jnb short loc_14027400B  
mov edx, [rsi+10h]  
lea rax, [rsi+40h]  
mov r8d, [rsi+34h]  
mov r9, rcx  
mov [rsp+0D0h+var_A8], rax  
mov rcx, rbx  
shl edx, 0Ch  
mov dword ptr [rsp+0D0h+var_B0], r15d  
call VslAllocateKernelShadowStack
```

MI_FLAGS.StrongPageIdentity (HVCI) is needed

```
SkmiProtectSinglePage(*ShadowStackPhysicalFrame, 3LL);
```

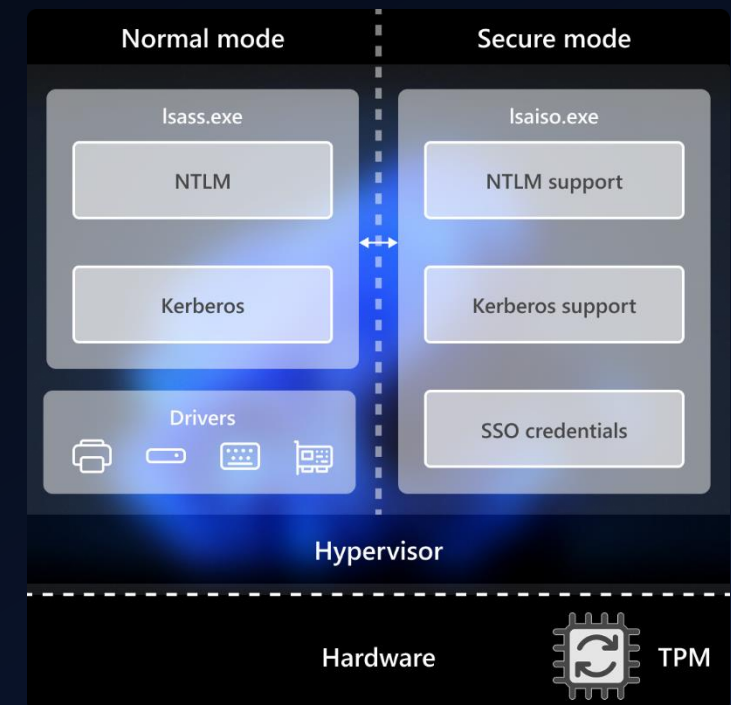
SkmiProtectSinglePage in VTL 1 marks the VTL 0 EPTE for the shadow stack as read-only

- Secure Kernel Patch Guard (SKPG), or HyperGuard
 - Yarden Shafir's multi-part blog series: <https://windows-internals.com/hyperguard-secure-kernel-patch-guard-part-1-skpg-initialization/>
 - HyperGuard addresses the fact that PatchGuard is code written in the kernel that is used to defend against modification against the kernel (kernel defending against itself)
 - Uses the concept of “SKPG extents” which describe target components/memory regions of interest to monitor
 - Intercepts (actions HyperGuard can literally intercept) are registered based on these extents in order to intercept certain items like MSRs, control registers, instructions, etc.
 - Instead of taking a “PatchGuard approach” and periodically checking for MSR manipulation, as an example, HyperGuard can intercept these operations
- Example: CET MSR access can be intercepted by HyperGuard (another reason kernel CET relies on VTL 1) to avoid modification

- Credential Guard
 - Windows Enterprise-specific
 - Leverages VTL 1 (Lsalso) as a place to store secrets/credential material previously stored in LSASS (VTL 0) – VBS makes this possible!
 - Lsalso.exe exposes RPC with which LSASS communicates with
- Traditional LSASS credential dumping becomes obsolete!
 - Kerberos TGTs are also protected
 - “TGT delegation trick” + TGT dumping

```
if ( g_fCredGuardEnabled )
{
    CredManIumRpcInit(&LsaIsoFunctions);
    NtlmIumRpcInit(&LsaIsoFunctions);
    KerbClientIumRpcInit(&LsaIsoFunctions);
    ProvIumRpcInit(&LsaIsoFunctions);
    CloudApIumRpcInit(&LsaIsoFunctions);
}
```

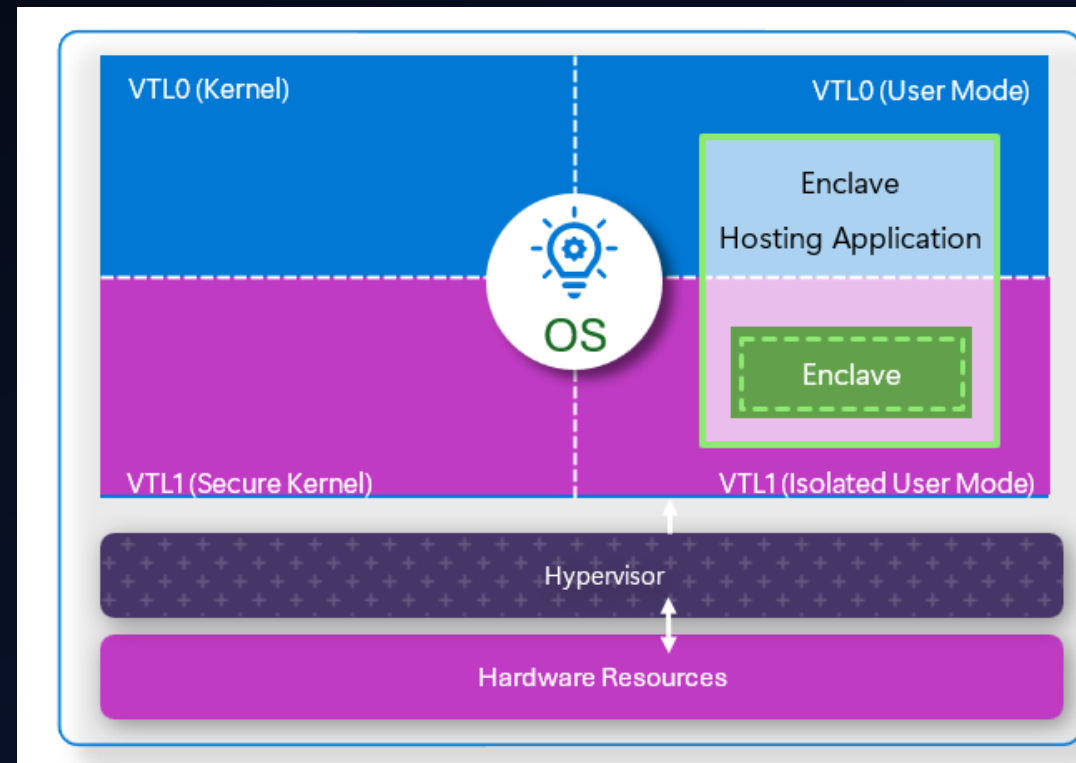
RPC initialization routines for supported authentication providers from Lsalso.exe



From Microsoft Docs (formerly MSDN): <https://learn.microsoft.com/en-us/windows/security/identity-protection/credential-guard/how-it-works>

- VBS Enclaves

- An isolated region of memory within an application which is protected by VBS
 - More specifically allows developers to protect parts of their applications with VBS
 - Now becoming accessible to third-party developers!
- Previously a type of hardware enclave (Intel SGX) existed, and the same basic premise has been extended by VBS (with obvious technical and implementation differences)



- Enclaves are manifested as DLLs
 - These DLLs are loaded by an untrusted “container” process
 - Enclave regions cannot be accessed even in kernel-mode of VTL 0
 - The address space of an enclave in VTL 0 has read/write access to the hosting process of the enclave, but the hosting process has *NO* access to *ANY* memory within the enclave
- Enclaves also have a limited API set to help reduce attack surface
- Developers can now seal sensitive data which is guaranteed by Virtual Secure Mode (VSM)!
 - And finally, as of Windows 11 && Server 2025 they are available to third-party developers using a *Trusted Signing VBS Enclave Certificate Profile*

APIs available in VBS enclaves

Article • 01/31/2024 • 1 contributor

[Feedback](#)

Enclaves are used to create trusted execution environments. These APIs are available to developers in VBS enclaves.

- Bcrypt APIs available in VBS enclaves
- UCRT APIs available in VBS enclaves
- Vertdll APIs available in VBS enclaves
- libvcruntime APIs available in VBS enclaves

- Conclusion

- It is self-evident Microsoft are making heavy investments into features and protections that have extremely strong guarantees (many of them, as we have seen, being backed by the hypervisor)
 - Some techniques/procedures are not just more expensive for attackers but outright *OBSOLETE* in many cases
- These are not features which require some special subscription model, they are baked into the OS and are ready for usage
 - We should start using them if they are available and encouraging others to use them!
- Some features which are dependent on VBS (like Kernel CET) require specific hardware
 - As this hardware starts to become more and more adopted it will be very interesting to see how attackers adapt/change over time in the same way legacy mitigations like DEP/ASLR forced the “OG’s” to adapt
- Thank you for coming to my talk!
 - I can’t guarantee I can answer everything, but happy to answer questions!