ANALYSIS OF THE ATTACK SURFACE OF WINDOWS 10 VIRTUALIZATION-BASED SECURITY

Rafal Wojtczuk <rafal@bromium.com>, 31 July 2016

Abstract

In Windows 10, Microsoft introduced virtualization-based security (VBS), the set of security solutions based on a hypervisor. In this paper, we will talk about details of VBS implementation and assess the attack surface - it is very different from other virtualization solutions. We will focus on the potential issues resulting from the underlying platform complexity (UEFI firmware being a primary example).

Note that most of the research described in this paper was done with Windows 10 Enterprise 1511. Judging by the preview Windows 10 versions, VBS is actively being worked on and enhanced; therefore it is possible that the behavior of later Windows 10 releases might differ in some aspects.

We focus on the Intel hardware throughout this paper.

Architecture and features

High-level overview of Windows 10 virtualization-based security (VBS) can be found at [dgo]. In this section we shortly summarize its architecture and provide details on the implementation. For more in-depth information, see [bsi] [sop].

Note that below we describe the operation and properties of VBS services under assumption that the hypervisor has not been compromised. Attacks against the hypervisor are discussed in “Hypervisor security” section.

When Hyper-V is present, it has control over the root partition. Therefore it is capable of implementing extra restrictions and providing secure services to the root partition. With VBS, the threat model is that the root partition has been compromised by malicious entity, up to the point of malware possessing kernel privileges. Note that VBS provides mitigation only for very specific attack cases, described below. Many actions typical for malware, e.g. exfiltration of confidential documents, keyloggers, or access to the internal (or corporate) network are not obstructed in any way.

When VBS is enabled, Hyper-V creates a special-purpose virtual machine and assigns it higher trust level (Virtual Trust Level 1, VTL1). Code running in this VM is responsible for providing security services. Unlike other VMs, VTL1 VM is protected from the root partition.

Credential Guard

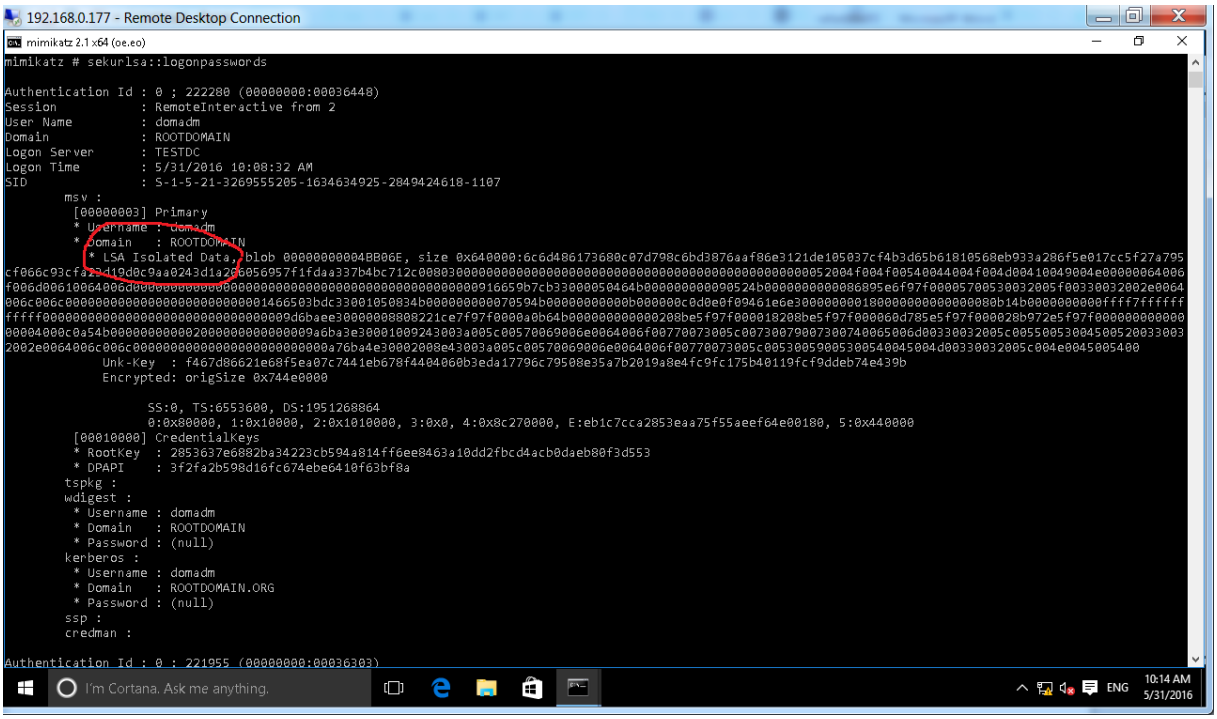
The best description of Credential Guard (CG) is [sop]. It is advertised as a method to prevent pass-the-hash attacks. Note that after KB2871997, lsass never keeps in memory the hashes of logged-out users – so all we can get is the hashes of a currently logged-in user. Also, in this paper we are not interested in Credential Manager, just logon credentials.

As described in [sop], there are two different types of configuration possible.

Scenario 1: Just enable Credential Guard in Group Policy

Without CG, lsass.exe keeps the hashes of passwords in its memory, in order to be able to provide SSO to remote servers. Many public tools, particularly mimikatz [mim], are capable of extracting these hashes. Afterwards, attacker can use these hashes to launch pass-the-hash attack from any machine, anytime (until the password is changed).

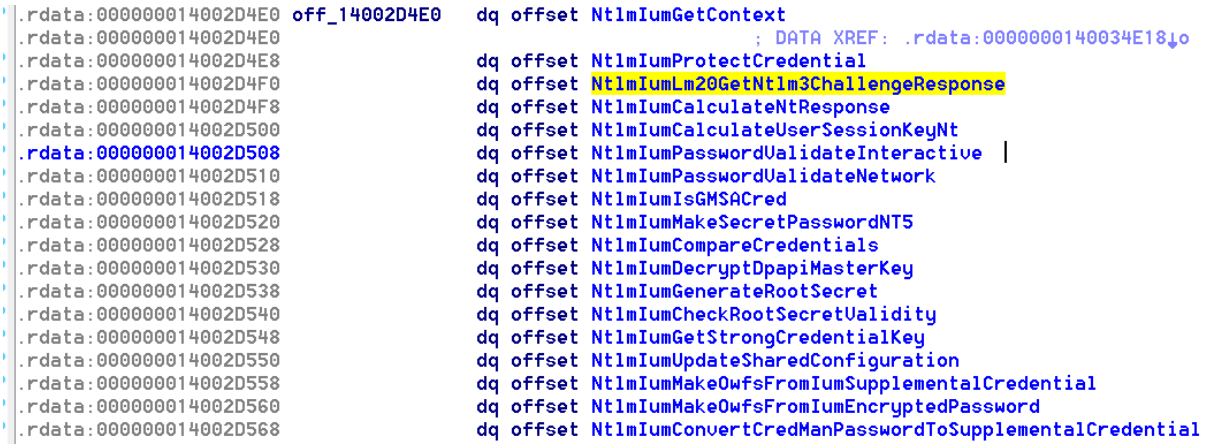
If CG is enabled, then lsass.exe no longer keeps hashes of passwords in its memory. As seen in the screenshot below, mimikatz[1] is not able to obtain the hashes (only encrypted opaque blobs are available):



How is SSO implemented in such case, particularly, the management of password hashes?

It is the LsaIso trustlet, running in VTL1 that keeps the hashes for the logged-in user. LsaIso exposes RPC interface over ALPC port \RPC Control\LSA_ISO_RPC_SERVER. The interface consists of the following functions:

[1] In fact, the unmodified mimikatz malfunctioned on w10 1511 system with CG enabled; I needed to patch out the attempt to print 0x640000 zeroes.



The workflow is as follows:

1) On startup, lsass calls NtlmIumGetContext, to obtain the secret nounce. This function can be called only once during system lifetime. All other functions require the correct nounce as the argument.
2) When user logs in, lsass creates a structure named _MSV1_0_SECRETS_WRAPPER and stores authentication material (likely, NTOWF) there. Then lsass calls NtlmIumProtectCredential. This function encrypts the authentication material, and passes the encrypted blob back to lsass (in the same _MSV1_0_SECRETS_WRAPPER structure). From this moment on, lsass no longer keeps any unencrypted authentication material in its memory (keeps the encrypted blob only).
3) Whenever actual authentication needs to happen (say, ntlm response needs to be computed), lsass calls the relevant LsaIso function (say, NtlmIumLm20GetNtlm3ChallengeResponse), providing as input the encrypted blob obtained in step 2. LsaIso is able to decrypt the blob, use the resulting authentication material to perform the required computation and pass the results back to lsass.

This design has the following properties:

1) After step 2, lsass no longer keeps any unencrypted authentication material in its memory – so it cannot be stolen and used for pass-the-hash attack.
2) While the user is logged in, if attacker can execute code as this user, then attacker can freely authenticate to other servers as the user, without any trickery needed. Attacker's actions are indistinguishable from the user using SSO legally.
3) If attacker collects the encrypted blob from lsass memory[2] while the user is logged in, then even after the user logs out, attacker can still call LsaIso functions, pass the encrypted blob and authenticate as the user.

   In the proof-of-concept code implementing the above, the rpc call to NtlmIumLm20GetNtlm3ChallengeResponse is hooked so that a saved encrypted blob is passed instead of the current credentials. Then any attacker action (say, access to SMB share) gets automatically authenticated, even after user logout.

[2] Attacker needs SYSTEM privileges for this

The situation is still improved in comparison to the case without CG: attacker needs to launch the lateral movement from the same machine that the encrypted blob was collected on. Also, after reboot, the encrypted blob is no longer valid – if we pass the saved encrypted blob to NtlmIumLm20GetNtlm3ChallengeResponse then it returns STATUS_DECRYPTION_FAILED.

4) There is still a problem with how the unencrypted credentials are initially delivered to VTL1 (which happens during logon). Attacker can entice the user to perform logon again, while the machine is under attacker's control, e.g. by locking the workstation (just run "rundll32.exe user32.dll,LockWorkStation"). If not using smart-card based authentication, then the plaintext credentials can be captured by keylogger and used anywhere, anytime. In case of smart-card based authentication, the NTOWF hashes sent by KDC can be captured and reused. This problem is solved below – but it requires much more configuration, and limiting flexibility.

Scenario 2: Credential Guard with armor key protection and smartcard-based authentication

As described in [sop], it is possible to configure an account so that it can authenticate only from a single machine. Another key (machine key, specific to the machine) is needed in order to decrypt the authentication material received from KDC. With CG enabled, the machine key is available only in VTL1 (protected against rogue root partition), so only CG running on a particular machine can decrypt the authentication material.

In such case, the above problem no 4 goes away. Still, same as before, the scenarios 2 and 3 are possible: until reboot, attacker can interact with CG and have it perform all necessary authentications (supported by SSO) for remote resources; again, all attacker's actions must originate from the initially-compromised machine. There is no reliable way to deliver "user has logged out, refuse SSO" message to VTL1 – it would have to be produced by the root partition. The threat model it that it has been compromised, so even if such a message was supported and recognized by VTL1, compromised lsass could just elect to not send it.

One thing worth noting is that the only secret needed permanently by CG is the machine key, and it is why TPM is required in order to protect it reliably against rogue root partition. Without TPM, there is no way to store the machine key permanently in a manner preventing the root partition from reading it[3].

VBS-enforced code integrity

One can configure Windows 10 to enforce code integrity of usermode binaries, usermode scripts[4] and kernelmode code. In this paper, we will talk about details of implementation of VBS-enforced kernelmode code integrity.

The goal is ambitious – not allow execution of any unsigned code in kernel context, even if the kernel has been compromised[5]. The basic idea is that the trusted code (running in VTL1) agrees to grant

[3] See also the below discussion on protecting keys used during S4.
[4] Judging by [pow], powershell scripts integrity is enforced in usermode and can be easily bypassed by attaching a debugger to the powershell process.

execute rights in EPT tables of the root partition only for pages storing signed code. As the root partition does not have direct access to its EPT table, then even malware having kernel privileges cannot alter these permissions.

The problem arises when the configuration allows unsigned usermode code and only signed kernelmode code. Intel CPUs use same EPT table regardless whether running VM usermode or kernelmode. The problematic scenario is:

1) Usermode tries to execute unsigned code stored in physical page X. DG needs to approve it, by marking X as executable in EPT.
2) VM changes to kernelmode, and attempts to execute same code as above. Because X was just marked as executable, this operation will succeed, violating the policy.

This problem was considered in secvisor paper [scv], and although it is not stated explicitly in any official MS document, VBS-enforced KMCI uses very similar approach. The scenario below starts with usermode running unsigned code:

1) The root partition is configured so that each attempt to transition to kernelmode causes vmexit, namely:
    a. Root partition usermode runs with both IDT limit and GDT limit set to 0. Thus, any action involving reloading CS must throw #GP exception. VMCS exception bitmap (when running usermode) is all 1.
    b. IA32_EFER.SCE is cleared (so "syscall" instruction throws #UD) and IA32_SYSENTER_CS is zeroed (so that "sysenter" instruction throws #GP).
2) Upon detecting attempt to transition to kernelmode, hypervisor changes EPT to a version (let's name it sEPT) suitable for kernel – it has execute permission bit set only for pages containing signed code. IDT and GDT limits are restored to sanity, context is manually switched to kernelmode, and execution is resumed.
3) When kernelmode attempts to return to usermode, then either:
    a. RIP points to unsigned usermode code. Page storing such code is never marked as executable in the EPT used to run kernelmode. So, vmexit (EPT violation) happens, DG detects that the root partition attempts to return to usermode, so it emulates just this, additionally flipping EPT pointer back to the EPT tables specific for unsigned usermode (let's name it uEPT).
    b. RIP points to signed usermode code. No vmexit happens, VM continues to run with sEPT, until it tries to execute unsigned usermode code (and then vmexit and switch to uEPT happens).
    c. RIP points to kernel code. Sane kernel never returns to usermode with RIP pointing to kernel code, but if it happens, such scenario does not allow breaking the policy (because all kernel code is signed). Depending on the page tables used and whether SMEP is available, this might cause an exception in kernelmode.

Naturally, in case of unsigned usermode code, such approach results in many additional vmexits, and this impacts performance. In the worst case scenario involving Hyper-V running in VM and the

[5] Actually, the last statement is a conjecture - the author has not found any official explicit description on what precisely VBS KMCI is supposed to ensure. Perhaps, the name is self-descriptive, at least to some.

workload consisting of executing nonexistent syscall in a tight loop (from a location in unsigned usermode page), the performance hit was x200. Yes, 20000% slowdown. Again, signed usermode code performance is not impacted (because there are no uEPT->eEPT transitions).

With this approach, there are effectively separate EPTs for unsigned and signed code, managed by VTL1:

1) The root partition can request DG to make a set of pages to be executable (in sEPT). If the pages form a signed executable, the request is approved - pages are marked as executable and not writable.
2) The root partition can request DG to make a set of pages to be nonexecutable (e.g. on kernel driver unload). Then pages are marked as not executable and writable.

One interesting effect of enabling VBS-enforced KMCI is that it makes writing exploits for kernel vulnerabilities more difficult. Usually, the goal of an exploit is to run arbitrary code. Introduction of SMEP made it more challenging, but the workarounds were natural – first, gain ROP capability, then the ROP chain can either turn off CR4.SMEP [smp], or change page tables so that some usermode page become kernelmode (S bit cleared). None of this approach beats or can be adapted to beat KMCI, because it is enforced by EPT, and the root partition does not have access to it. This does not mean that kernel exploits are impossible – one can either use write-what-where primitive to corrupt kernel memory (e.g. change the token for the current process to be SYSTEM's), or perform all required actions in a ROP chain (but this is painful, and not everything is possible). Some actions become problematic, though – e.g. one cannot easily hook kernel code (because it is not writable[6]) and moreover one cannot have an arbitrary unsigned hook code.

A vigilant reader should observe that it is crucial to maintain the W^X invariant on all signed pages – otherwise, attacker could overwrite the legal signed executable code with unsigned code, without DG noticing. The algorithm described above seems to achieve this.

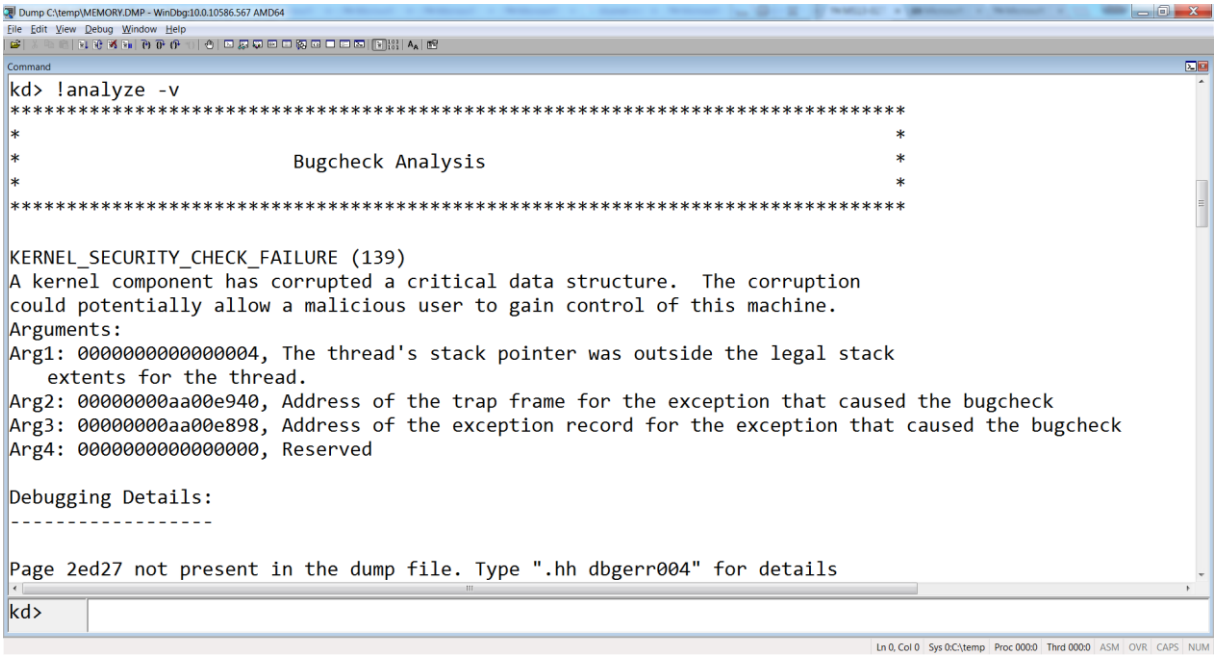
However, when analyzing EPT dumps of a W10 1511 system with KMCI enabled, it turned out that there are pages marked RWX. At the moment of inspection they seem to store non-executable usermode content; it is unclear what they were used for initially. The best hypothesis it that they are the pages originally storing early boot code, UEFI-related possibly. For whatever reason they were marked RWX, and then returned to the kernel as free pages, with the EPT permissions not corrected.

The impact is that attacker capable of corrupting kernel memory can execute arbitrary, unsigned code in kernel context. This vulnerability has been reported to Microsoft by the author and fixed in MS16-066 bulletin.

A technical problem remains – how an attacker in the root partition can determine which pages have RWX permission in EPT. Of course root partition does not have any access to pages storing its EPT, so attacker cannot search EPT directly. The idea is to "probe" each page in kernelmode – try writing to it and then executing. A failed probe will result in exception injected by the hypervisor. The probe code needs to catch this exception and continue probing. Catching an exception is straightforward if we can load a kernel module, and use normal try/except construct. The more interesting case is when

[6] If attacker has ability to create executable pages with arbitrary content, then possibly hooking can be done with the help of permanently changing page table entries pointing to the relevant pages.

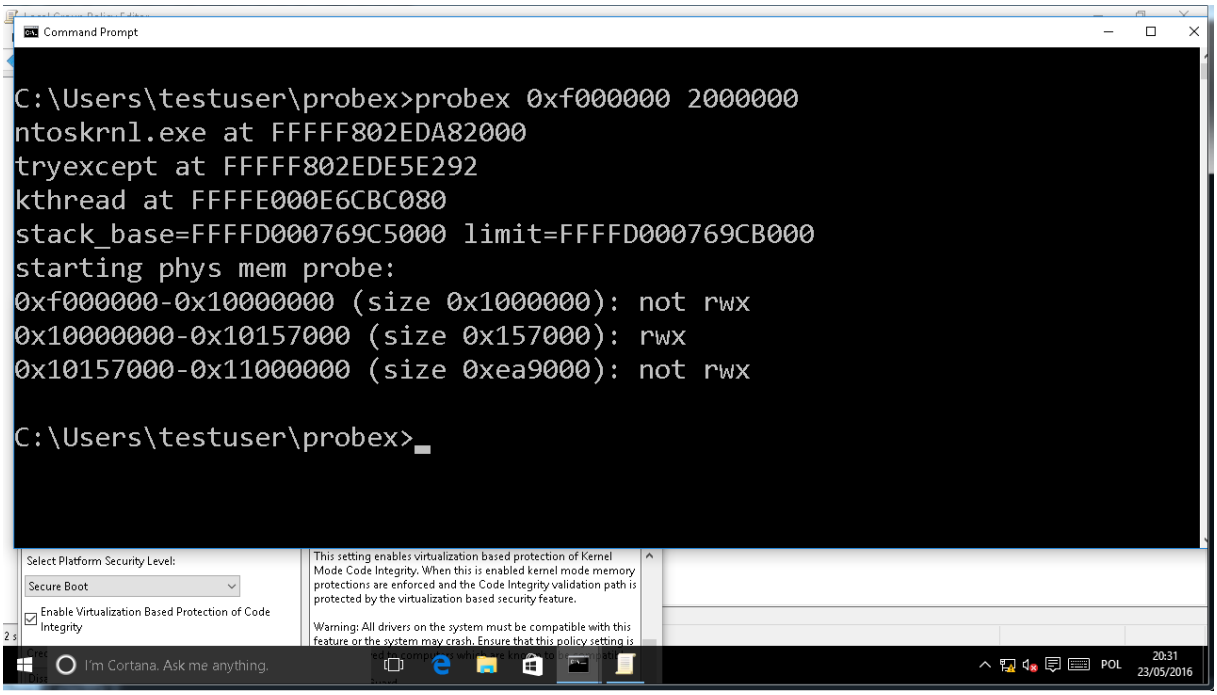
attacker does not have ability to load a custom signed module[7]. Let's assume attacker can launch a ROP chain in kernel mode, e.g. by exploiting a kernel vulnerability. Then we can use a gadget that resides in legal code wrapped in an exception handler. A suitable gadget is "call r10" instruction in nt!ObQueryNameStringMode function – if code executed by it throws an exception, it is gracefully handled and the function returns (to another ROP gadget) with exception code in RAX register. One catch is that apparently during exception handling, kernel does a security check – it verifies that the current stack pointer is within [stack_base, stack_limit] range defined by KTHREAD fields, and if not, it bugchecks in nt!RtlpGetStackLimits via int 29:



This means that the probe ROP chain must execute with RSP being within the legal bounds. Normally, for exploit purposes, the ROP chain is stored in usermode pages. Therefore, the ROP chain needs to relocate itself to the legal thread stack on startup.

The proof-of-concept ROP chain is capable of all the above; on the test W10 1511 system, it produces the following output:

[7] In fact, this is the case when attacker is most interested in achieving execution of arbitrary unsigned code in the kernel context.



On the test system, there were other, smaller RWX regions as well, scattered around physical address 0xa00000.

Bioiso, vtpm trustlets

In post-1511 Windows 10 builds there is another trustlet, BioIso. We did not investigate its properties. Similarly, we did not research vtpm trustlet functionality.

VTL1 security

The code running in VTL1 is privileged, and its compromise could be fatal to security functions implemented in VTL1. The attack surface consists of:

1) Services exposed by VTL1
   a) rpc services implemented in LsaIso. That also means the whole rpc demarshalling code. All that runs in VTL1 usermode.
   b) 48 services implemented in securekernel!IumInvokeSecureService (called by nt! HvlpEnterIumSecureMode), so, in VTL1 kernelmode.
2) VTL1 extensively calls into VTL0 (the root partition) to use some services. VTL1 must be careful to sanitize all responses received from VTL0.

This topic has been well discussed [bsi], and Microsoft apparently has devoted a lot of effort to audit all the interactions. Still, there is potential for vulnerabilities there.

Hypervisor security

The whole concept of VBS hinges on the ability to run code in VTL1 partition securely, even if the root partition is compromised. This protection is implemented by the hypervisor; if the latter can be compromised, all security guarantees are gone.

The topic of hypervisor security has been discussed on many occasions. However, usually the threat model is: a rogue, unprivileged, worker VM that has no access to hardware. In our case, we try to secure the hypervisor from an attacker in the root partition. Note that without VBS enabled, the root partition can compromise Hyper-V in a number of ways:

1) There is HvCallDisableHypervisor hypercall, that does what its name suggests, in runtime
2) hypervisor pages are in clear in hiberfile. We have verified that hooking the hibernation code in the kernel (and trojaning the hypervisor pages before they are written to the hiberfile) results in a trojaned hypervisor after restore from hibernation.
3) VTd not enabled, so it is possible to overwrite hypervisor body via DMA.

All these above known issues are prevented when VBS is enabled[8]. However, note that VBS can be enabled on a system without VTd support (or with VTd disabled)[9]. In such case, attacker can use the demonstrated DMA attacks against the hypervisor [sxh] and hijack the hypervisor in runtime. It is worth repeating one more time – VTd support is necessary in order to secure Hyper-V against the root partition. If available, Hyper-V enables both DMA and interrupt remapping – both are necessary [svt].

Another crucial feature is secureboot. It is required in order to enable VBS[10]. There were numerous vulnerabilities allowing breaking secureboot protection [sb1] [sb2], and they could be exploited from within the root partition[11]. The impact of such an exploit depends on the configuration:

1) Credential Guard with armor key protection and smartcard-based authentication.
   Secureboot vulnerability does not break credential guard in this case, because the change in the boot sequence will result in change of TPM measurements, which will result with inability to unseal the secret machine key[12]. All other features (e.g. code integrity) could be bypassed, though.
2) The remaining cases
   All VBS protection can be bypassed by altering the boot sequence so that a trojaned hypervisor is loaded.

It is worth mentioning that secureboot (even if it works as designed) is a weak concept, because it does not guarantee that the expected environment has been launched – just one of myriad of signed ones. TPM provides much better assurance, if used properly.

From now on, let's assume there is a system with unbreakable secureboot and VTd enabled. Again, any exploitable hypervisor vulnerability can be used to break VBS. Hyper-V has a very good security history – the only Microsoft security bulletins related to vulnerabilities allowing escaping from a VM

[8] Additionally, some extra features are enabled, e.g. vmexit on sidt (and related) instruction is enabled, thus preventing malicious usermode from leaking IDT/GDT addresses. Another features are preventing CR0.WP clear and similar. Nice !

[9] Still, Microsoft documentation clearly recommends enabling VTd.

[10] Interestingly, after VBS has been enabled, it is possible to reboot and disable secureboot in UEFI configuration (which requires physical presence), and VBS is still running.

[11] Note unlike any other VMs, root partition has access to the boot and system device, as well to UEFI variables related to boot, and thus can alter the boot sequence.

[12] Assuming that the implementation is correct (relevant secrets are sealed to the proper PCRs); not verified.

are [vms] [ms1]. However, most of attention has been devoted to attacks originating from normal, unprivileged VMs. What are the attack vectors specific to an attacker in the root partition[13]?

In comparison to usual VMs, the root partition has the following extra capabilities:

1) Privileged (available only to the root partition) hypercalls
   Hypervisor Top-Level Functional Specification [tls] mentions 14 hypercalls usable by nonprivileged VM, and 67 privileged hypercalls. More hypercalls exist, entirely undocumented. It should be apparent that this kind of attack surface exposed to the root partition is much larger.
2) Access to the most of the physical memory address space
   Besides pages devoted exclusively for hypervisor and VTL1, the root partition can access almost whole physical address space, including MMIO, exceptions being LAPIC and VTd bars whose addresses are routed to a dummy page in the root partition's EPT. Some pages are shared with VTL1 (e.g. libraries, like iumdll.dll, are mapped both in VTL1 and the root partition to the same physical address), but they are marked as read-only in the root partition's EPT.
   With HVCI enabled, the pages storing UEFI runtime are marked as read-only, and UEFI runtime executes in VTL1 context. In such case, it is imaginable to deny the root partition access to SMM services[14], which would be a significant improvement. Without HVCI, the pages storing UEFI runtime are marked as writable, and UEFI runtime executes in the root partition kernel context.
3) IO ports
   All are directly accessible, with the following exceptions (configured in the root partition's VMCS IO bitmap):
   a) 32, 33 (PCH interrupt controller), 160, 161 (same). Thus, the root partition cannot interfere with physical interrupts directly.
   b) 0x64, lpc microcontroller, keyboard - possibly trapped to control A20 gate setting
   c) 0xcf8, 0xcfc-0xcff. Hyper-V apparently wants to have ability to inspect attempts to alter PCIe config space.
   d) 0x1804. It is PMBASE+4 == PM1_CNT, it holds the SLP_EN bit, that triggers S3 sleep. This prevents the root partition from entering S3 sleep directly. More on S3 later.
4) MSR
   Only three SYSENTER MSRS and fs/gs/shadow gs base are allowed direct access; write to the rest result in vmexit. Hypervisor Top-Level Functional Specification states: "The default behavior for MSRs not listed here is pass through to hardware for root, and #GP for non-root", but it might not apply to VBS case. It would be interesting to reverse-engineer Hyper-V handling of such writes to verify that writes to some crucial ones, like IA32_APIC_BASE MSR, are handled properly [tms].

[13] Note the root partition does not need virtual devices (or it implements them by itself), so vulnerabilities in the device emulator (e.g. virtual disk) are not applicable to the case of an attacker in root partition, while a significant problem in case of a rogue normal VM. Similarly, denial-of-attacks against hypervisor are entirely not interesting in case of the root partition – it can simply do "shutdown /p" etc.

[14] UEFI runtime SetVariable must invoke SMM, as the flash storage is writable only in SMM mode. Unfortunately, on some systems, SMM services are used for other vital purposes, and denying the root partition access to SMM is problematic.

Utilizing the availability of the above resources, the following attacks are possible.

Unrestricted PCIe config space access via MMCFG

On Intel chipsets, it is possible to reserve a region of physical address space for memory-mapped PCIe config space. Most firmware enable such region, by writing the region base to PCIEXBAR register (in device 0:0.0 PCIe config space). Access to such region results in PCIe config cycles, the device and offset determined by the offset of MMCFG region being accessed.

Hyper-V does not restrict access to MMCFG. Therefore, unlike access via ioports cf8/cfc case, the root partition can access PCIe config space via MMCFG without Hyper-V knowing about it.

It is not known whether and how precisely such access could be abused. There are some interesting registers in chipset PCIe config space whose altering can result in behavior breaking hypervisor security. E.g. REMAP_LIMIT/REMAP_BASE registers (that reside in device 0:0.0 PCIe config space), can be used to create physical memory aliases and provide access to hypervisor memory [rbl]. REMAP_LIMIT/REMAP_BASE registers are expected to be locked by firmware, though.

Note that a naïve attack based on configuring PCIe memory bar address so that it overlaps (and covers, takes priority when routing memory access) some sensitive RAM locations (e.g. used by hypervisor) will not work[15]. Intel specifications state clearly that only addresses in [TOLUD, 4G] and above TOUUD are decoded to DMI and can be claimed by a PCIe device. TOLUD and TOUUD registers are locked by any sane firmware. Other types of resource overlaps might be more dangerous. Particularly, it is possible to cover the VTd bars with PCIe memory bar, and apparently read access is routed to PCIe memory, at least on the tested Haswell and Skylake systems:

```
[root@haswell bh16]# cp /sys/firmware/acpi/tables/DMAR .
[root@haswell bh16]# iasl -d  DMAR >/dev/null
Loading Acpi table from file DMAR
Acpi Data Table [DMAR] decoded
Formatted output:  DMAR.dsl - 5488 bytes
[root@haswell bh16]# grep -i register DMAR.dsl
[038h 0056   8]        Register Base Address : 00000000FED90000
[050h 0080   8]        Register Base Address : 00000000FED91000
[root@haswell bh16]# ./rdmem 0xfed90000
phys memory at 0xfed90000: 0x00000010 0x00000000 0x20660462 0x00c00000
[root@haswell bh16]# ./rdmem 0xfed91000
phys memory at 0xfed91000: 0x00000010 0x00000000 0x20660462 0x00d20080
[root@haswell bh16]# setpci -s 0:2.0 0x10.l
f5800004
[root@haswell bh16]# setpci -s 0:2.0 0x10.l=0xfed90004; ./rdmem 0xfed90000; setpci -s 0:2.0 0x10.l=0xf5800004
phys memory at 0xfed90000: 0x00000000 0x00000000 0x00000000 0x00000000
[root@haswell bh16]# setpci -s 0:2.0 0x10.l=0xfed91004; ./rdmem 0xfed91000; setpci -s 0:2.0 0x10.l=0xf5800004
phys memory at 0xfed91000: 0x00000000 0x00000000 0x00000000 0x00000000
[root@haswell bh16]#
```

The plausible attack is to cover VTd bar range, which will result in Hyper-V being unable to invalidate cached VTd translations (which is necessary for security when changing ownership of a page from root domain to HyperV, e.g. during HvDepositMemory hypercall). However, on the tested systems, the write access to the overlapped region resulted in platform hang (again, despite read access apparently working), so the above attack would not work on the tested systems[16].

[15] Overlapping physical memory ranges are briefly mentioned in [gd1], however, without any description of an actually reproduced attack. Either this is the "naïve" scenario described here, or something not applicable to more-or-less recent Intel chipsets.

[16] Most likely, other generations of chipsets behave the same; not verified.

Other chipset registers access

Full semantics of all chipset registers is not documented publicly by Intel. Some memory-mapped regions[17], e.g. ones in MCHBAR , have thousands of registers, most of them undocumented at all. We can only hope that all sensitive ones (example: VTd bars addresses) are properly lockable by chipset (quite likely) and that firmware actually locks them. Intel has implemented Hardware Security Test Interface to help with verifying that the known crucial locks have been applied.

Another little-known example of a scary feature of the chipset is the ability to overwrite the SPD configuration of the DRAM chips, stored in DRAM EEPROM [spd]. It is imaginable that altering SPD data can result in creating physical memory aliases. SPD EEPROM can be accessed using SMBUS, via IO ports 0x50–0x57. Again, firmware is expected to disable this feature, by writing to the "SPD Write disable" bit in (recent) PCH D31:F4 config space at offset 0x40.

S3 sleep

S3 sleep/resume cycle is interesting from security point of view. During this cycle, hypervisor loses control, and needs to trust firmware to maintain its integrity and properly return to the hypervisor.

The boot script hijack vulnerability [bt1] [bt2] rootcause was that the crucial data structure (boot script) controlling S3 resume was kept in normal RAM. This attack can be launched from the root partition and provide arbitrary asmcode execution before Hyper-V is woken up. Such asmcode can trojan the hypervisor and then complete S3 resume, maintaining the stability of the system. This vulnerability is expected to be fixed in up-to-date firmware releases.

The above vulnerability was interesting even outside of hypervisor security context, because it allowed for code execution in environment where not all hardware locks have yet been applied by firmware. But if we are focused on attacking the hypervisor, we do not care about these locks, all we need is to get arbitrary code execution before the hypervisor. One possibility is to introduce mismatch between where hypervisor stores the waking vector, and what firmware actually uses. Hyper-V follows the specification, and just before triggering S3 sleep, it stores the waking vector in ACPI FACS table. Consider the following scenario:

1) Assumption: firmware caches the location of FACS table in its private data structures
2) Then, assuming the cached pointer is in ACPI NVS region, root partition can corrupt the cached pointer before asking Hyper-V to initiate S3 sleep.
3) During S3 resume, firmware will read the waking vector not from the legal ACPI FACS table, but from the location controlled by the root partition, and thus arbitrary code will be executed instead of Hyper-V's waking vector.

Assumption 1 (and the following scenario) was confirmed with an old version of AMI firmware; current versions are not vulnerable.

Another potential problem is that the ACPI FACS table is writable by the root partition. One should consider the following attack:

1) Write asmcode address to ACPI FACS
2) trigger S3 sleep directly, without using hypercalls, by writing to the PM1_CNT io port

[17] Memory-mapped, thus accessible to the root partition without any Hyper-V mediation.

In order to prevent this, both the following conditions must hold:

1) hypervisor does not grant root partition direct access to PM1_CNT io port
2) root partition cannot change the value of PMBASE io port range (that includes PM1_CNT) in chipset registers

Hyper-V takes care of 1); tested chipsets seem to lock register PMBASE (in device 0:31.0, offset 0x40 on recent Intel chipsets) relevant to 2).

It would be interesting to investigate whether any of the below is possible:

a) S3 can be triggered by something else than write to PM1_CNT
b) Attacker can warm-reboot and fool firmware into believing that it is resume from S3, not a warm reboot

Either would bypass Hyper-V safeguards.

S4 sleep

Similarly to S3 case, during S4, hypervisor loses control. This time, the information on the system state is stored in hiberfile that is written to by the root partition. Therefore, the contents of hiberfile must be protected against tampering.

The natural idea is that VTL1 should be tasked with hiberfile generation and encrypt it before asking root partition to write it to disk. The problem is how to manage encryption keys. The keys must

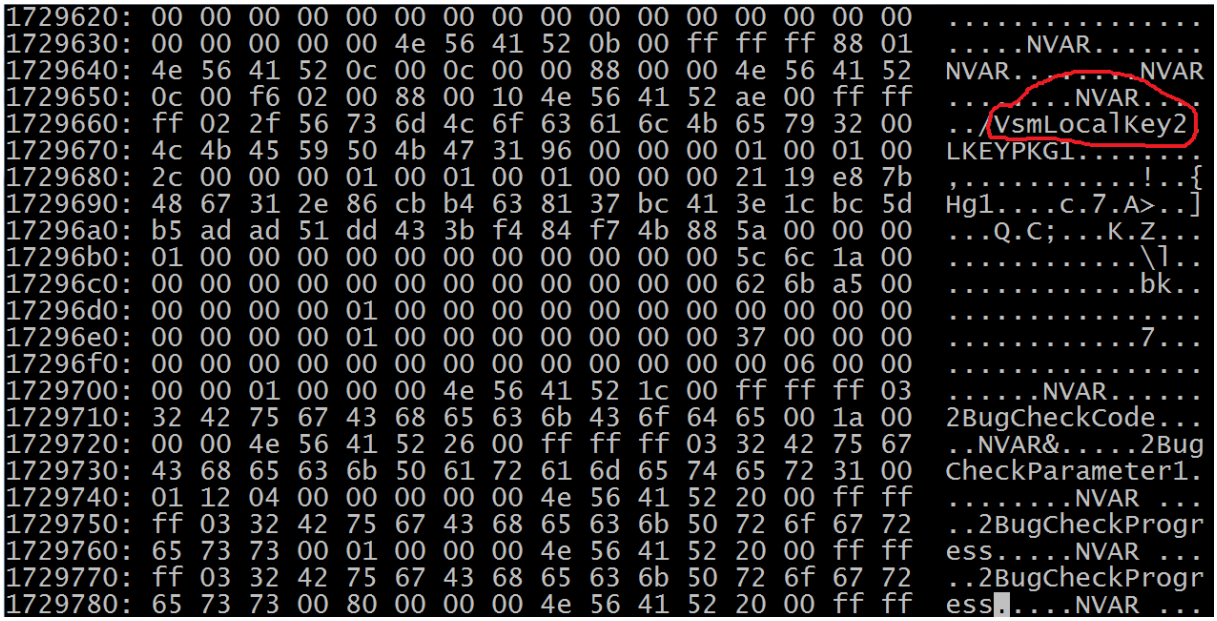
1) be available to VTL1
2) survive cold reboot (so permanent storage is needed)
3) be inaccessible to any other OS that can be booted in the meantime
4) winresume.efi must be able to read them

The combination of 3 and 4 means the permanent storage mechanism must distinguish between "the expected winresume.efi is running" and "some other OS, potentially booted for malicious purposes is running"[18], and release keys only in the former case. The only way to implement this is to use TPM. If it is available, the key package is sealed to particular PCR values[19] and the encrypted blob is saved to EFI environment variable "VsmLocalKey2" (prepended by the 0x44454C4145534C42 constant). Winresume! BlVsmKeysReadAndUnsealLKeyPkg function is able to retrieve the cleartext keys via TPM unseal operation.

If TPM is not available, then there is no hope. "VsmLocalKey2" is created with attributes indicating that only boot services can access it (without EFI_VARIABLE_RUNTIME_ACCESS), and indeed after OS is started, the documented UEFI runtime functions do not expose it. However, the storage for UEFI variables (SPI flash) is accessible directly by any entity capable of reading physical memory below 4G (where SPI flash is mapped) – root partition is capable of it.

[18] Or, at least, are we before ExitBootServices(). This could be done with help of ME (as it has its own flash partition); such "key escrow" is not available, though.
[19] Meaning, TPM will release the cleartext only if the platform is in the expected state (read: winload.efi is running)

With TPM unavailable[20], the key package is stored unencrypted in “VsmLocalKey2” variable, this time content prepended by 0x31474B5059454B4C constant. It can be found in physical memory dump:



Therefore, without TPM, the keys used to encrypt hiberfile can be determined by root partition, and the contents of the hiberfile changed on the fly during hibernation so that the hypervisor is trojaned[21].

SMM

SMM is highly-privileged mode of execution of x86 CPU. Compromised SMM code can compromise a hypervisor[22].

There was a number of SMM vulnerabilities discovered in the past. Before ca 2011, most of the vulnerabilities were at the platform level, caused by insufficient protection of SMM code against hostile ring0 code. Recently, all newly discovered vulnerabilities are in the SMM code itself, in the code providing services to the operating system. Such services can be invoked by OS via write to IO port 0xb2.

The crucial observation is that VBS allows root partition to access IO port 0xb2, thus allowing it to invoke SMM services directly. Another issue is that VBS allows root partition to corrupt ACPI NVS memory[23] – many SMM vulnerabilities were caused by SMM code trusting data (e.g. code pointer) in ACPI NVS. Therefore, it is expected that most of SMM vulnerabilities are exploitable from root partition.

The quality of SMM code is perceived as poor, and more such vulnerabilities are expected in the future. For a quite comprehensive overview of the exploitation process of particular recent (from 2016) vulnerabilities and the impact, see the recent [do1] [bp1], and especially [do2].

---

[20] Note Microsoft documentation recommends enabling TPM if possible.
[21] This has not been reproduced, though – some nontrivial code is required to actually perform necessary crypto operations.
[22] Unless STM is in use; but, no major firmware vendor support it
[23] Again, denying root partition access to ACPI NVS results in compatibility problems on some systems.

The author used a runtime vulnerability in an obsolete AMI firmware to achieve convenient, repeatable code execution in SMM context. Such capability allows for e.g. scanning the whole physical memory for VMCS pages[24]:

```
testuser@DESKTOP-W10 /c/Users/testuser/projects/bh16
$ tasklist | grep -i iso
LsaIso.exe                    812 Services                   0      3,108 K

testuser@DESKTOP-W10 /c/Users/testuser/projects/bh16
$ ./smm31.exe scan 0 2>/dev/null >scan.txt

testuser@DESKTOP-W10 /c/Users/testuser/projects/bh16
$ tail -4 scan.txt
VMCS at 0x8d4d000: host:  rip=FFFFF8000809011E cr3=00000000007C9000 ept=0000000006A7201E
                   guest: rip=FFFFFFFFFFD05000 cr3=00000000001AB000 cr4=00000000001526F8
VMCS at 0x8d4f000: host:  rip=FFFFF8000809011E cr3=00000000007C9000 ept=0000000006A7901E
                   guest: rip=FFFFF78000003035 cr3=000000000121B000 cr4=00000000000026F8

testuser@DESKTOP-W10 /c/Users/testuser/projects/bh16
$ wc -l scan.txt
32 scan.txt
```

With hypervisor's cr3 and vmexit handler address known (retrieved from VMCS), we can hook the vmexit handler, thus gaining full control over hypervisor's behavior. The framework was created facilitating compiling C code into dll and calling it in the vmexit handler context; sample code (shown without supporting includes and runtime) below:

```
testuser@DESKTOP-W10 /c/Users/testuser/projects/bh16
$ cat hyperhook_cpuid.c
#include <stdint.h>
#include "regsh.h"
#include "vmx.h"

void hook_c(struct regs * regs)
{
        if (do_vmread(VM_EXIT_REASON) == EXIT_REASON_CPUID &&
                regs->r13 == 0xAABBCCDDAABBCCDDULL)
                regs->r13 = 0x1122334411223344ULL;
}
```

In the example below, we show that a hooked vmexit handler can recognize that a CPUID instruction was executed with a magic value in R13 register, and respond with altering R13 value[25]:

[24] Other frameworks, notably Chipsec [chi], facilitate scanning for VMCS as well.
[25] Some readers might recognize similarities with „bpknock" operation on a Bluepill-ed system

```
testuser@DESKTOP-W10 /c/Users/testuser/projects/bh16
$ head -1 scan.txt
VMCS at 0x01fb000: host:  rip=FFFFF8000809011E cr3=00000000007C9000 ept=0000000006A7201E

testuser@DESKTOP-W10 /c/Users/testuser/projects/bh16
$ ./smm31.exe readv FFFFF8000809011E 00000000007C9000 2>/dev/null
vwalk: level 0 ptbase 00000000007C9000 pte 00000000007CD023
vwalk: level 1 ptbase 00000000007CD000 pte 00000000007CF023
vwalk: level 2 ptbase 00000000007CF000 pte 00000000040001A1
va FFFFF8000809011E -> pa 000000000409011E
4C8B4828244C8948

testuser@DESKTOP-W10 /c/Users/testuser/projects/bh16
$ ./magic_cpuid.exe
before cpuid: r13=AABBCCDDAABBCCDD; after cpuid: r13=AABBCCDDAABBCCDD

testuser@DESKTOP-W10 /c/Users/testuser/projects/bh16
$ ./smm31.exe hyperhook FFFFF8000809011E 000000000409011E 2>/dev/null
now=4C8B4828244C8948 write 4C8B48FFF6FEDDE9 pbase=0000000004000000
shsize=0x2000
shellcode written to phys 0000000004000000

testuser@DESKTOP-W10 /c/Users/testuser/projects/bh16
$ ./magic_cpuid.exe
before cpuid: r13=AABBCCDDAABBCCDD; after cpuid: r13=1122334411223344
```

Similarly, another proof-of-concept code used SMM privileges to hook syscall handler of VTL1 kernel (securekernel.exe), so that the syscalls invoked by LsaIso.exe could be monitored.

There are reports of successful use of firmware vulnerability (allowing to escalate to SMM) to extract secrets from Credential Guard [yb1], apparently by just scanning the whole physical memory for buffers looking like credentials.

WSMT ACPI table

In April 2016, Microsoft has released the specification of Windows SMM Security Mitigations Table [wsm]. Many of the above potential attacks are described there; firmware vendors should follow these guidelines in order to help secure VBS against attacks. Microsoft claims that firmware shipped with their hardware is hardened against these types of vulnerabilities.

Other non-VBS-specific attack vectors

CPU erratas

If CPU behaves in a way not matching the specification, then it is possible that the security measures implemented in a hypervisor can be bypassed.

The author is not aware of any case in the past when CPU errata was successfully exploited to escape from VTx VM confinement.

The most interesting cases were CVE-2015-5307 and CVE-2015-8104[26]. VM could misconfigure its #AC or #DB exception handling so that CPU entered infinite loop when handling the relevant exception. However, this was DoS only.

This paper focuses on Intel's hardware. In case of AMD, there is a report about a microcode bug at least potentially allowing escape from VM [rs1].

Rowhammer

Similarly, if DRAM is not working properly, then it can be abused to corrupt memory used by a privileged entity, including the hypervisor.

Root partition can control (to some extent) which pages are used by Hyper-V (via HvDepositMemory hypercall); this might make the exploitation reasonably reliable (at least in case of repeatable bitflips).

Malicious discrete device firmware

If there is a discrete device (say, discrete GPU card) with flashable firmware, then the root partition is capable of overwriting this device's firmware, by using the legal interfaces exposed by the device. If the attacker can create their own malicious firmware (which is highly nontrivial), then there are well-known additional attack vectors. For example, malicious firmware could use DMA to hijack early UEFI boot phase (before hypervisor configures VTd protection). Such an attack is believed to be very difficult, as well as specific to the discrete device and used UEFI firmware.

Summary

Despite its limited scope, VBS is useful – it prevents certain attacks that are straightforward without it. The security posture of VBS looks good, and it improves the security of a system – certainly it requires additional highly nontrivial effort to find suitable vulnerability allowing the bypass.

VBS depends on availability and security of quite a few features. Looking at the official documentation, one may infer that secureboot is a strict requirement, and VTd and TPM are recommended but optional enhancements. In fact, both VTd and TPM are necessary in order to protect hypervisor against compromised root partition. Similarly, just enabling Credential Guard provides little protection – more configuration is needed in order to ensure that the credentials never show up in the clear in the root partition. In any case, until reboot, a compromised user process can still use SSO to authenticate to other servers in order to perform lateral movement.

In this paper, many mechanisms are assessed only at design/architecture level – and almost all look sound. More problems are possible at the implementation level. The amount of relevant, security-critical code in the hypervisor and VTL1 is significantly smaller than in case of e.g. all Windows ring0 code, yet non-negligible.

Not surprisingly, in case of a well-configured system, the biggest threat comes from vulnerabilities in SMM code. There are many examples of such bugs, and more are expected in the future.

[26] Unlike most of the erratas, they were easily exploitable.

Acknowledgements

The author would like to thank Alex Ionescu for reviewing the paper.

The author would like to thank Microsoft for reviewing the paper.

The potential issues with altering SPD data on DRAM chips were brought to my attention by James McKenzie at bromium com.

Tomasz Wróblewski at bromium com participated in testing of VTd bar covering.

Bibliography

[dgo] Brian Lich, *Device Guard overview*, https://technet.microsoft.com/en-us/itpro/windows/whats-new/device-guard-overview

[bsi] Alex Ionescu, *Battle of the skm and ium: how windows 10 rewrites os architecture*, https://www.blackhat.com/us-15/briefings.html#battle-of-the-skm-and-ium-how-windows-10-rewrites-os-architecture

[sop] Seth Moore , Baris Saydag, *Defeating pass-the-hash: Separation of powers*, https://www.blackhat.com/us-15/briefings.html#defeating-pass-the-hash-separation-of-powers

[mim] Benjamin Delpy, *Mimikatz*, https://github.com/gentilkiwi/mimikatz

[pow] Binni Shah, *To Disarm Device Guard - Bring a Debugger to the Fight*, http://subt0x10.blogspot.in/2016/05/to-disarm-device-guard-bring-debugger.html

[scv] Arvind Seshadri, Mark Luk, Ning Qu, Adrian Perrig, *SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes,* http://www.cs.cmu.edu/~./arvinds/pubs/secvisor.pdf

[smp] Mateusz 'j00ru' Jurczyk, Gynvael Coldwind, *SMEP: What is it, and how to beat it on Windows*, http://j00ru.vexillium.org/?p=783

[sxh] Rafal Wojtczuk, *Subverting the Xen Hypervisor*, https://www.blackhat.com/presentations/bh-usa-08/Wojtczuk/BH_US_08_Wojtczuk_Subverting_the_Xen_Hypervisor.pdf

[svt] Rafal Wojtczuk. Joanna Rutkowska, *Software attacks against Intel(R) VT-d technology*, http://www.invisiblethingslab.com/resources/2011/Software%20Attacks%20on%20Intel%20VT-d.pdf

[vms] Kostya Kortchinsky, Thomas Garnier, *Hyper-V vmswitch.sys Guest-to-Host kernel pool overflow and whatnot*, https://technet.microsoft.com/library/security/MS16-045

[ms1], Thomas Garnier, *Vulnerabilities in Windows Hyper-V Could Allow Remote Code Execution*, https://technet.microsoft.com/en-us/library/security/ms15-068.aspx

[tls] Microsoft, *Hypervisor Top-Level Functional Specification*, https://github.com/Microsoft/Virtualization-

Documentation/raw/master/tlfs/Hypervisor%20Top%20Level%20Functional%20Specification%20v4.0b.pdf

[tms] Christopher Domas , *The Memory Sinkhole*, https://www.blackhat.com/docs/us-15/materials/us-15-Domas-The-Memory-Sinkhole-Unleashing-An-x86-Design-Flaw-Allowing-Universal-Privilege-Escalation-wp.pdf

[rbl] Joanna Rutkowska, Rafał Wojtczuk, *Preventing and Detecting Xen Hypervisor Subversions*, http://invisiblethingslab.com/resources/bh08/part2-full.pdf

[spd] Wikipedia, *Serial Presence Detect*, https://en.wikipedia.org/wiki/Serial_presence_detect

[bt1] Rafal Wojtczuk, Corey Kallenberg, *Attacks on UEFI security*, https://cansecwest.com/slides/2015/AttacksOnUEFI_Rafal.pptx

[bt2], Intel ATR, *Technical Details of the S3 Resume Boot Script Vulnerability*, http://www.intelsecurity.com/advanced-threat-research/content/WP_Intel_ATR_S3_ResBS_Vuln.pdf

[do1] Dmytro Oleksiuk, *Exploiting SMM callout vulnerabilities in Lenovo firmware*, http://blog.cr4.sh/2016/02/exploiting-smm-callout-vulnerabilities.html

[bp1] Bruno Pujos, *SMM unchecked pointer vulnerability*, http://esec-lab.sogeti.com/posts/2016/05/30/smm-unchecked-pointer-vulnerability.html,

[yb1], Intel FOCUS conference presentation (starting at 31m30), http://focus.intelsecurity.com/focus2015/player.html?xml=02-DAY2-SG_SAKW.xml

[wsm] Microsoft, *Windows SMM Security Mitigations Table*, http://download.microsoft.com/download/1/8/A/18A21244-EB67-4538-BAA2-1A54E0E490B6/WSMT.docx

[sb1] Yuriy Bulygin, Andrew Furtak, Oleksandr Bazhaniuk, *A Tale of One Software Bypass of Windows 8 Secure Boot*, http://www.c7zero.info/stuff/Windows8SecureBoot_Bulygin-Furtak-Bazhniuk_BHUSA2013.pdf

[sb2] Yuriy Bulygin, Andrew Furtak, Oleksandr Bazhaniuk, John Loucaides, *All Your Boot Are Belong To Us*, https://cansecwest.com/slides/2014/AllYourBoot_csw14-intel-final.pdf

[do2] Dmytro Oleksiuk, *Exploring and exploiting Lenovo firmware secrets*, http://blog.cr4.sh/2016/06/exploring-and-exploiting-lenovo.html

[gd1] Gal Diskin, *Virtually Impossible: The Reality Of Virtualization Security*, http://2013.zeronights.org/includes/docs/Gal_Diskin_-_Virtually_Impossible_-_ZeroNights_release_version.pdf

[chi] Intel, *CHIPSEC: Platform Security Assessment Framework*, http://www.intelsecurity.com/advanced-threat-research/chipsec.html

[rs1] Robert Święcki, *AMD newest ucode 0x06000832 for Piledriver-based CPUs seems to behave in a problematic way*, http://seclists.org/oss-sec/2016/q1/450