AYI NEDJIMI Consultants

HVCI Deep Dive : Intégrité du Code Protégée par l Hyperviseur

Catégorie : Deep Dive Lecture : 28 min Publié le : 30/03/2026 Auteur : Ayi NEDJIMI

Deep dive HVCI : architecture VTL0/VTL1, SLAT, CI.dll sécurisé, politiques signature kernel, bypasses documentés et mitigations. Analyse interne Windows.

HVCI — Hypervisor-Protected Code Integrity — constitue l'une des avancées les plus significatives en matière de sécurité kernel sous Windows depuis l'introduction de PatchGuard. En déplaçant la validation de l'intégrité du code depuis le noyau VTL0, intrinsèquement compromettable, vers un environnement isolé par l'hyperviseur en VTL1, Microsoft a fondamentalement redéfini la surface d'attaque disponible pour les exploits kernel. Ce deep dive technique s'adresse aux chercheurs en sécurité, développeurs kernel et red teamers qui souhaitent comprendre en profondeur les mécanismes internes de HVCI : l'architecture Virtualization-Based Security, le rôle de la Second Level Address Translation dans l'enforcement W^X, le flux de validation des signatures de code, les contraintes imposées aux drivers, et les techniques de contournement documentées dans la littérature académique et offensive. Nous examinerons les structures de données internes, les hypercalls, les implications sur les performances, et les configurations de déploiement en environnement d'entreprise. Chaque section s'appuie sur l'analyse du code désassemblé, les publications de Project Zero, et les recherches présentées à Black Hat, SSTIC et OffensiveCon entre 2019 et 2026.

Pourquoi HVCI existe — le problème de l'intégrité kernel

Pour comprendre pourquoi Microsoft a investi massivement dans HVCI, il faut revenir au problème fondamental de l'intégrité du code kernel dans un système d'exploitation monolithique. Historiquement, Windows s'appuyait sur Code Integrity (CI.dll) chargé dans le noyau lui-même — c'est-à-dire au même niveau de privilège que le code qu'il était censé protéger. Cette architecture créait un paradoxe de sécurité fondamental : le gardien résidait dans la forteresse qu'il était censé défendre.

L'héritage de CI.dll en Ring 0

Avant VBS, le module CI.dll s'exécutait en Ring 0 dans l'espace d'adressage du noyau NT. Lorsqu'un driver devait être chargé, ntoskrnl.exe invoquait les routines de CI.dll pour valider la signature Authenticode. Le problème est évident : un attaquant disposant d'une primitive d'écriture kernel arbitraire pouvait simplement patcher CI.dll en mémoire. Les fonctions `CiCheckSignedFile` et `CiValidateImageHeader` pouvaient être hookées ou neutralisées. Plus subtil encore, un attaquant pouvait modifier les structures de données internes utilisées par CI.dll — par exemple les listes de certificats de confiance ou les pointeurs de callback — sans toucher au code lui-même.

PatchGuard (Kernel Patch Protection) offrait une certaine protection en vérifiant périodiquement l'intégrité de structures kernel critiques, mais PatchGuard lui-même s'exécutait en Ring 0 et pouvait être contourné. Des techniques documentées — comme celles de Satoshi Tanda et Alex Ionescu — montraient comment désactiver PatchGuard en manipulant les timers kernel ou en exploitant les fenêtres de vérification. L'intégrité du code reposait donc sur un modèle de sécurité par l'obscurité plutôt que sur une isolation architecturale.

BYOVD : le cauchemar des défenseurs

La technique BYOVD (Bring Your Own Vulnerable Driver) a démontré de manière spectaculaire la faiblesse du modèle pré-HVCI. Les attaquants n'avaient même pas besoin de trouver un 0-day kernel : il suffisait de charger un driver légitime mais vulnérable, signé par un éditeur de confiance, pour obtenir une primitive de lecture/écriture kernel. Des drivers comme `RTCore64.sys` (MSI Afterburner), `dbutil_2_3.sys` (Dell BIOS Utility), ou `gdrv.sys` (GIGABYTE) étaient utilisés en production par des groupes APT. Une fois le driver vulnérable chargé, l'attaquant pouvait :

- Lire et écrire la mémoire kernel arbitrairement
- Désactiver les callbacks de sécurité (PsSetCreateProcessNotifyRoutine, ObRegisterCallbacks)
- Modifier les tokens de processus pour l'élévation de privilèges
- Patcher CI.dll pour autoriser le chargement de drivers non signés
- Désactiver DSE (Driver Signature Enforcement) en modifiant `g_CiOptions`

Le problème fondamental était clair : tant que le mécanisme d'enforcement de l'intégrité résidait au même niveau de privilège que le code malveillant, aucune protection logicielle ne pouvait offrir de garanties solides. La solution exigeait un changement architectural radical — déplacer la frontière de confiance vers un niveau de privilège inaccessible depuis le noyau. C'est exactement ce que fait HVCI en s'appuyant sur l'hyperviseur.

Architecture VBS — le socle de HVCI

HVCI ne fonctionne pas de manière isolée. Il s'inscrit dans le cadre plus large de la Virtualization-Based Security (VBS), une architecture de sécurité qui exploite les extensions de virtualisation matérielle (Intel VT-x, AMD-V) pour créer des environnements d'exécution isolés. Comprendre HVCI nécessite de comprendre VBS dans son intégralité.

Architecture VBS avec VTL0, VTL1, hyperviseur et SLAT enforcement

Virtual Trust Levels : VTL0 et VTL1

VBS introduit le concept de Virtual Trust Levels (VTL). Il s'agit de niveaux d'isolation matérielle gérés par l'hyperviseur Hyper-V. Actuellement, deux VTL sont utilisés :

- **VTL0 (Normal World)** : c'est l'environnement classique où s'exécutent le noyau NT (ntoskrnl.exe), tous les drivers, les processus utilisateurs, et l'ensemble de l'écosystème Windows traditionnel. Même si VTL0 contient du code Ring 0 (kernel), ce code est considéré comme non-trusted dans le modèle VBS.

- **VTL1 (Secure World)** : c'est l'environnement protégé par l'hyperviseur. Il exécute le Secure Kernel (securekernel.exe) et les processus IUM (Isolated User Mode). Les trustlets VTL1 — comme LsaIso.exe (Credential Guard) et CI.dll (HVCI) — bénéficient d'une isolation mémoire matérielle. Le code de VTL0, même en Ring 0, ne peut pas accéder à la mémoire de VTL1.

La hiérarchie de confiance est donc : Hyperviseur > VTL1 > VTL0. Le noyau NT en VTL0, traditionnellement le composant le plus privilégié du système, est relégué à un niveau de confiance inférieur à celui du Secure Kernel en VTL1. C'est un renversement complet du modèle de sécurité historique.

Hyper-V comme Root of Trust

L'hyperviseur Hyper-V (hvix64.exe pour Intel, hvax64.exe pour AMD) s'exécute au niveau le plus privilégié — VMX Root Mode sur Intel, ou l'équivalent Host Mode sur AMD. Il est chargé très tôt dans le processus de boot, avant le noyau NT lui-même. L'hyperviseur est responsable de :

- La création et l'isolation des VTL via les structures VMCS (Virtual Machine Control Structure)
- La gestion des tables SLAT/EPT (Second Level Address Translation / Extended Page Tables)
- Le dispatch des hypercalls depuis VTL0 et VTL1
- L'enforcement des permissions mémoire — c'est le mécanisme central de HVCI
- L'interception des violations EPT (EPT violations) lorsque VTL0 tente un accès interdit

Le point essentiel est que l'hyperviseur contrôle les mappings mémoire de VTL0. VTL0 ne peut pas modifier ses propres permissions de page au niveau SLAT. Si VTL0 veut qu'une page devienne exécutable, il doit en faire la demande via un hypercall — et c'est l'hyperviseur (en consultation avec VTL1) qui décide d'accorder ou non cette permission.

Secure Kernel et processus IUM

Le Secure Kernel (securekernel.exe) est un micro-noyau minimaliste qui s'exécute en VTL1. Il est considérablement plus petit que ntoskrnl.exe — environ 700 Ko contre plus de 10 Mo — et offre une surface d'attaque réduite. Il fournit les services essentiels aux trustlets IUM :

- Gestion de la mémoire sécurisée (allocation dans l'espace VTL1)
- Scheduling des threads IUM (en coordination avec le scheduler VTL0)
- Communication sécurisée avec l'hyperviseur via les hypercalls VTL1
- Gestion des Secure Interrupts

Les trustlets IUM incluent LsaIso.exe (qui isole les secrets d'authentification pour Credential Guard et les Secured-core PCs), le composant CI.dll de VTL1 (HVCI engine), et d'autres enclaves comme bioiso.exe (Windows Hello) ou KeyIso.exe (CNG Key Isolation).

Communication inter-VTL : les hypercalls

La communication entre VTL0 et VTL1 se fait exclusivement via des hypercalls — des appels de procédure qui transitent par l'hyperviseur. Il n'existe aucun canal de mémoire partagée non contrôlé. Les hypercalls pertinents pour HVCI incluent :

```
// Hypercalls clés pour HVCI
HvCallModifyVtlProtectionMask     // Modifier les permissions SLAT d'une page
HvCallQueryVtlProtectionMask      // Interroger les permissions actuelles
HvCallEnableVpVtl                 // Activer un VTL pour un processeur virtuel
HvCallVtlReturn                   // Retour d'une exécution VTL1 vers VTL0
HvCallVtlCall                     // Appel depuis VTL0 vers code VTL1
```

Lorsque le noyau VTL0 a besoin de charger un driver, il invoque `HvCallVtlCall` pour transférer l'exécution au Secure Kernel en VTL1. Celui-ci dispatch la requête vers CI.dll (VTL1) pour la validation. Le résultat est communiqué via la mémoire partagée mailbox, et l'hyperviseur met à jour les entrées SLAT en conséquence. Ce mécanisme assure que même un noyau VTL0 entièrement compromis ne peut pas contourner la validation — il ne peut tout simplement pas modifier les permissions SLAT sans l'accord de VTL1.

CI.dll dans VTL1 — le gardien des pages de code

Le composant CI.dll existe en deux versions sous HVCI : une version legacy en VTL0 (qui reste pour la compatibilité) et une version sécurisée en VTL1 (qui effectue la véritable validation). C'est cette instance VTL1 qui constitue le coeur de HVCI.

Flux de validation de code HVCI - du chargement driver à la vérification de signature

Le flux de validation détaillé

Quand le noyau NT reçoit une demande de chargement de driver (via NtLoadDriver, ou implicitement lors du boot via le Service Control Manager), le flux suivant se déroule :

Étape 1 — Mapping du PE en mémoire. Le noyau VTL0 lit le fichier PE (Portable Executable) du driver depuis le disque et le mappe dans l'espace d'adressage kernel. À ce stade, les pages contenant le code sont marquées comme Read-Write dans les tables de pages VTL0, mais l'hyperviseur maintient le bit Execute à 0 dans les entrées SLAT correspondantes. Le code est donc physiquement présent en mémoire mais ne peut pas être exécuté.

Étape 2 — Requête de validation vers VTL1. Le noyau invoque un hypercall (via `HvCallVtlCall`) pour signaler au Secure Kernel qu'un nouveau module doit être validé. La requête contient les adresses physiques des pages à valider et les métadonnées du PE (headers, répertoire de signatures).

Étape 3 — CI.dll (VTL1) effectue la validation. Le module CI.dll en VTL1 effectue les vérifications suivantes :

- Parsing du catalogue de signatures Authenticode (ou signature embedded dans le PE)
- Vérification de la chaîne de certificats jusqu'à une racine de confiance
- Hash SHA-256 (ou SHA-1 pour la rétrocompatibilité) de chaque section de code du PE
- Comparaison du hash calculé avec le hash signé dans le catalogue
- Vérification contre la politique WDAC/CI si une politique est configurée
- Consultation de la driver blocklist si applicable

Étape 4 — Décision et mise à jour SLAT. Si la signature est valide et conforme à la politique, CI.dll (VTL1) demande à l'hyperviseur de modifier les entrées SLAT pour les pages de code du driver : elles passent de Read-Write à Read-Execute. Le code devient exécutable mais n'est plus modifiable. Si la validation échoue, les pages restent non-exécutables et le chargement du driver échoue avec `STATUS_INVALID_IMAGE_HASH`.

Analyse du désassemblage de CI.dll (VTL1)

L'analyse du binaire CI.dll chargé en VTL1 (que l'on peut extraire depuis le répertoire System32 et analyser statiquement, même si son exécution est protégée) révèle les fonctions internes suivantes :

```
// Fonctions clés dans CI.dll (VTL1) - analyse statique IDA/Ghidra
CipValidateImageHash()           // Point d'entrée principal de la validation
CipCheckSignedFile()             // Vérification de la signature Authenticode
CipValidateFileObject()          // Validation de l'objet fichier
CipGetHashFromCatalog()          // Recherche de hash dans les catalogues installés
CipQueryPolicyInformation()      // Consultation de la politique CI active
MinCryptVerifySignedDataLMode()  // Vérification crypto bas-niveau (MinCrypt lib)
CipValidateImageData()           // Validation des données d'image PE
CipReportBlockedBinary()         // Signalement d'un binaire bloqué (event log)
```

La bibliothèque cryptographique MinCrypt, embarquée dans CI.dll, est une implémentation minimale de la vérification de signature qui ne dépend d'aucune API CNG ou CAPI2 du noyau VTL0 — tout est autonome. Cette indépendance est cruciale : la compromission de la stack crypto de VTL0 ne peut pas affecter la validation en VTL1.

Le secret fondamental : l'asymétrie de confiance

Le point conceptuel le plus important de HVCI est l'asymétrie de confiance. VTL1 peut lire la mémoire de VTL0 (pour valider le code des drivers), mais VTL0 ne peut pas accéder à la mémoire de VTL1. Cette propriété est garantie par le hardware — les entrées SLAT pour les pages VTL1 n'ont simplement aucune permission depuis le contexte VTL0. Ce n'est pas une protection logicielle contournable par un patch mémoire : c'est un enforcement matériel au niveau du processeur.

Concrètement, même si un attaquant obtient une primitive d'exécution de code arbitraire en Ring 0 dans VTL0, il ne peut pas : lire la mémoire de VTL1, modifier CI.dll en VTL1, altérer les entrées SLAT, ou forger une réponse de validation. La seule option pour contourner HVCI est soit de trouver une vulnérabilité dans l'hyperviseur lui-même (ce qui constitue une escalade de privilèges d'un type fondamentalement différent), soit de procéder à des attaques qui ne nécessitent pas l'exécution de code non signé — les attaques data-only.

Second Level Address Translation (SLAT) — le mécanisme d'enforcement

La SLAT (Second Level Address Translation), connue sous le nom EPT (Extended Page Tables) chez Intel et NPT (Nested Page Tables) chez AMD, est le mécanisme matériel fondamental qui rend HVCI possible. Sans SLAT, HVCI ne pourrait tout simplement pas fonctionner.

Enforcement W^X via SLAT - Write XOR Execute, jamais les deux simultanément

Fonctionnement de la SLAT/EPT

Dans un environnement virtualisé, la translation d'adresse se fait en deux étapes. L'adresse virtuelle (VA) du guest est d'abord traduite en adresse physique du guest (GPA) via les tables de pages classiques (CR3). Ensuite, cette GPA est traduite en adresse physique de l'hôte (HPA) via les tables SLAT/EPT. Chaque entrée EPT (PTE du second niveau) contient des bits de permission :

Bit EPT	Position	Description	Rôle HVCI
Read (R)	Bit 0	Autorise la lecture de la page	Toujours 1 pour les pages de code
Write (W)	Bit 1	Autorise l'écriture dans la page	0 pour le code validé (après chargement)
Execute (X)	Bit 2	Autorise l'exécution du code	1 uniquement après validation CI.dll VTL1
UserModeExecute	Bit 10	Exécution autorisée depuis User Mode	Contrôle additionnel pour les processus IUM
VerifyGuestPaging	Bit 57	Vérification supplémentaire des pages guest	Protection MBEC (Mode-Based Execution Control)

L'enforcement W^X au niveau matériel

Le principe fondamental de HVCI est le W^X (Write XOR Execute) : une page mémoire peut être soit inscriptible (RW-), soit exécutable (R-X), mais jamais les deux simultanément (RWX est interdit). Ce principe existe dans certains systèmes d'exploitation en mode utilisateur (DEP/NX bit), mais HVCI l'applique au niveau kernel via la SLAT, ce qui le rend impossible à contourner depuis le noyau.

Le cycle de vie d'une page de code kernel sous HVCI est le suivant :

```
// Phase 1 : Chargement du driver
// La page contient le code PE fraîchement lu depuis le disque
// SLAT permissions : Read=1, Write=1, Execute=0
// → Le kernel peut écrire (relocation, fixups) mais pas exécuter

// Phase 2 : Validation réussie par CI.dll (VTL1)
// L'hyperviseur modifie les permissions SLAT
// SLAT permissions : Read=1, Write=0, Execute=1
// → Le CPU peut exécuter le code mais personne ne peut le modifier

// Phase 3 : Hotpatch ou mise à jour (cas spécial)
// Le noyau demande une transition temporaire via hypercall
// SLAT permissions : Read=1, Write=1, Execute=0
// → Écriture autorisée temporairement, exécution bloquée
// Après modification : retour à Read=1, Write=0, Execute=1
```

L'impossibilité d'avoir RWX signifie qu'un attaquant ne peut jamais simultanément modifier une page de code et l'exécuter. S'il veut modifier du code, la page perd son bit Execute. S'il veut exécuter du code, la page est en lecture seule. Et la transition entre les deux états requiert un hypercall que seul VTL1 peut autoriser.

EPT Violations et interception

Lorsque du code VTL0 tente d'accéder à une page d'une manière qui viole les permissions EPT — par exemple, tenter d'exécuter une page marquée non-exécutable, ou tenter d'écrire dans une page en mode Execute — le processeur génère une EPT Violation. C'est un événement de type VM-Exit qui transfère immédiatement le contrôle à l'hyperviseur. L'hyperviseur peut alors :

- Bloquer l'accès et injecter une exception (#GP ou #PF) dans le guest VTL0
- Journaliser la tentative pour analyse forensique
- Dans certains cas, transférer le contrôle à VTL1 pour traitement

Ce mécanisme est particulièrement important pour détecter les tentatives d'exploitation. Un exploit kernel qui tenterait d'exécuter du shellcode dans un pool buffer (NonPagedPool classique) provoquerait immédiatement une EPT Violation, car le buffer est en mémoire RW mais pas X. L'attaque est bloquée au niveau matériel, avant même que le shellcode ne puisse exécuter sa première instruction.

Analyse des structures EPT internes

Pour les chercheurs qui analysent HVCI au niveau le plus bas, il est utile de comprendre la structure exacte des entrées EPT. Chaque entrée EPT de niveau feuille (PTE EPT) est un entier 64 bits avec le format suivant :

```
// Structure d'une entrée EPT de niveau feuille (Intel SDM Vol. 3C, §28.2.6)
// Bits 0-2   : Permissions (R, W, X)
// Bit 3-5    : Memory Type (UC=0, WC=1, WT=4, WP=5, WB=6)
// Bit 6      : Ignore PAT (si 1, utilise le memory type EPT au lieu du PAT guest)
// Bit 7      : Page size (0 = 4KB, 1 = 2MB/1GB selon le niveau)
// Bits 8     : Accessed flag
// Bit 9      : Dirty flag
// Bit 10     : User-mode execute (si MBEC activé)
// Bits 11    : Réservé
// Bits 12-51 : Physical address de la page (bits 12-51 de l'HPA)
// Bits 52-56 : Réservés
// Bit 57     : Verify guest paging (pour sub-page permissions)
// Bit 58     : Paging-write access (pour EPT paging modifications)
// Bit 59     : Réservé
// Bit 60     : Supervisor Shadow Stack (si CET activé)
// Bits 61-62 : Réservés
// Bit 63     : Suppress #VE (Virtualization Exception)

// Sous HVCI, une page de code validée a typiquement :
// Bits 0-2 = 0b101 (R=1, W=0, X=1) → lecture et exécution, pas d'écriture
// Bit 10   = 0 (pas d'exécution user-mode pour le code kernel)
// Bit 60   = variable (shadow stack page si CET est actif)
```

L'hyperviseur Hyper-V maintient deux jeux de tables EPT pour chaque processeur virtuel : un pour VTL0 et un pour VTL1. Les entrées EPT de VTL0 sont configurées avec des permissions restreintes (contrôlées par VTL1), tandis que les entrées EPT de VTL1 ont des permissions complètes sur leur propre mémoire. La mémoire VTL1 est tout simplement absente des tables EPT de VTL0 — les entrées correspondantes sont marquées comme non-présentes, ce qui garantit l'isolation matérielle.

MBEC : Mode-Based Execution Control

Intel a introduit MBEC (Mode-Based Execution Control) avec les processeurs Kaby Lake et ultérieurs, et AMD a implémenté GMET (Guest Mode Execute Trap) comme équivalent. Ces extensions permettent de différencier les permissions d'exécution entre le mode kernel (Ring 0) et le mode utilisateur (Ring 3) au niveau SLAT, sans avoir besoin de modifier les entrées EPT dynamiquement. Avant MBEC, l'hyperviseur devait intercepter chaque transition kernel/user pour mettre à jour les EPT, ce qui causait un overhead significatif. Avec MBEC, une seule entrée EPT peut spécifier des permissions d'exécution différentes pour le kernel et l'user mode, éliminant ces transitions coûteuses. Windows 11 et Windows Server 2025 utilisent MBEC par défaut lorsqu'il est disponible, ce qui réduit considérablement l'impact de HVCI sur les performances.

Politiques de signature et validation du code kernel

HVCI ne se contente pas de vérifier qu'un binaire est signé — il applique un ensemble complexe de politiques qui déterminent quelles signatures sont acceptables, quels certificats sont de confiance, et quels binaires sont explicitement bloqués. Comprendre ces politiques est essentiel pour les administrateurs qui déploient HVCI et pour les red teamers qui cherchent à le contourner.

Exigences de signature pour les drivers kernel

Sous HVCI, les drivers kernel doivent satisfaire des exigences de signature strictes. Depuis 2021, Microsoft exige que tous les nouveaux drivers kernel soient soumis au programme WHQL (Windows Hardware Quality Labs) ou signés via l'attestation Microsoft. Les signatures EV (Extended Validation) seules ne suffisent plus pour les drivers kernel — elles sont nécessaires pour soumettre au portail du Hardware Dev Center, mais le binaire final doit recevoir une co-signature Microsoft.

Type de signature	Accepté par HVCI (défaut)	Notes
WHQL (Microsoft co-signed)	Oui	Signature standard pour les drivers publiés via Windows Update
Attestation signing (Microsoft)	Oui	Pour les drivers non-WHQL soumis via le portail dev
EV Certificate seul (cross-signed)	Non (depuis Windows 11)	Était accepté avant — transition terminée
Standard Authenticode (non-EV)	Non	Insuffisant pour le kernel mode depuis Windows 10 1607
Test Signing (self-signed)	Non	Uniquement en mode test (testsigning BCD)
Driver blocklisted (signature révoquée)	Non	Blocké même si la signature est techniquement valide

WDAC : Windows Defender Application Control

WDAC (anciennement Device Guard) fonctionne en synergie avec HVCI pour offrir un contrôle granulaire du code autorisé. Tandis que HVCI gère l'enforcement W^X et la validation cryptographique de base, WDAC définit les politiques qui déterminent quels binaires spécifiques sont autorisés ou bloqués. Les politiques WDAC sont stockées dans des fichiers au format CIP (Code Integrity Policy) ou P7B (PKCS#7), déployés via Group Policy, Intune, ou SCCM.

Une politique WDAC peut spécifier des règles à plusieurs niveaux de granularité :

```
<!-- Exemple simplifié de règles WDAC pour drivers kernel -->
<SiPolicy xmlns="urn:schemas-microsoft-com:sipolicy">
  <Rules>
    <Rule>
      <Option>Enabled:UMCI</Option>          <!-- User-mode CI -->
    </Rule>
    <Rule>
      <Option>Enabled:Boot Menu Protection</Option>
    </Rule>
    <Rule>
      <Option>Required:WHQL</Option>          <!-- Exige WHQL -->
    </Rule>
  </Rules>
  <FileRules>
    <Deny ID="ID_DENY_RTCORE"
          FriendlyName="RTCore64.sys blocklist"
          FileName="RTCore64.sys"
          MinimumFileVersion="0.0.0.0" />
  </FileRules>
</SiPolicy>
```

Catalogue de signatures et ISG

Windows maintient un catalogue de signatures dans `%SystemRoot%\System32\catroot\` qui contient les hashes des fichiers signés par Microsoft et les éditeurs de confiance. HVCI consulte ce catalogue lors de la validation. L'ISG (Intelligent Security Graph) est un mécanisme optionnel qui permet d'autoriser des binaires basés sur leur réputation cloud — Microsoft Defender analyse les fichiers et attribue un score de confiance. En mode ISG, un binaire inconnu mais jugé sûr par le cloud peut être autorisé même sans signature explicite dans la politique locale. Ce mécanisme est cependant controversé car il introduit une dépendance au cloud et une surface d'attaque potentielle (si le service ISG est compromis ou si un attaquant peut influencer la réputation d'un binaire).

Supplemental Policies et politiques multiples

Depuis Windows 10 1903, WDAC supporte les politiques supplémentaires (Supplemental Policies). Un système peut avoir une politique de base restrictive (par exemple, n'autoriser que les binaires signés Microsoft) et une ou plusieurs politiques supplémentaires qui ajoutent des exceptions (par exemple, autoriser les drivers d'un éditeur spécifique pour le matériel de l'entreprise). Cette architecture permet un déploiement progressif : la politique de base est standard dans l'organisation, et les exceptions sont gérées par service ou par type de machine. Chaque politique supplémentaire doit référencer l'ID de la politique de base, ce qui empêche le détournement de politiques supplémentaires orphelines.

Impact sur les drivers et le noyau

L'activation de HVCI impose des contraintes significatives sur le code kernel. Les drivers qui utilisaient des techniques maintenant interdites — code auto-modifiant, allocations RWX, pools exécutables — doivent être mis à jour ou seront bloqués. Cette section détaille les implications techniques pour les développeurs de drivers.

Contraintes sur les allocations mémoire

Sous HVCI, toute allocation de pool kernel qui tente de combiner les permissions Write et Execute est bloquée. Les développeurs doivent migrer vers les APIs NX-safe :

```
// ❌ INTERDIT sous HVCI : allocation exécutable
PVOID buf = ExAllocatePoolWithTag(NonPagedPool, size, 'Tag1');
// NonPagedPool est RWX par défaut — bloqué

// ✅ CORRECT sous HVCI : allocation non-exécutable
PVOID buf = ExAllocatePool2(POOL_FLAG_NON_PAGED, size, 'Tag1');
// Pool2 utilise NonPagedPoolNx par défaut — RW seulement

// ❌ INTERDIT : MDL avec permissions exécutables
PVOID mapped = MmMapLockedPagesSpecifyCache(
    mdl, KernelMode, MmCached, NULL, FALSE,
    NormalPagePriority | MdlMappingNoExecute);
// Le flag MdlMappingNoExecute est OBLIGATOIRE sous HVCI

// ❌ INTERDIT : VirtualAlloc kernel avec PAGE_EXECUTE_READWRITE
ZwAllocateVirtualMemory(handle, &base, 0, &size,
    MEM_COMMIT, PAGE_EXECUTE_READWRITE);
// RWX est bloqué — utiliser PAGE_READWRITE puis
// PAGE_EXECUTE_READ après écriture
```

Migration des types de pool

Le changement le plus courant est la migration de `NonPagedPool` vers `NonPagedPoolNx`. Historiquement, NonPagedPool allouait de la mémoire RWX — ce qui était pratique pour les drivers qui généraient du code à la volée (comme certains drivers de virtualisation, émulateurs, ou outils de monitoring), mais représentait un risque de sécurité majeur. La table suivante résume la migration :

API Legacy	API HVCI-compatible	Notes
`ExAllocatePool(NonPagedPool, ...)`	`ExAllocatePool2(POOL_FLAG_NON_PAGED, ...)`	Pool2 est NX par défaut
`ExAllocatePoolWithTag(NonPagedPool, ...)`	`ExAllocatePool2(POOL_FLAG_NON_PAGED, ...)`	Tag passé en paramètre Pool2
`MmAllocateContiguousMemory(...)`	`MmAllocateContiguousNodeMemory(...)`	Avec flag NX
`MmMapIoSpace(...)`	`MmMapIoSpaceEx(..., PAGE_READWRITE)`	Spécifier les permissions explicitement
`ExInitializeNPagedLookasideList`	`ExInitializeLookasideListEx`	Avec POOL_NX_ALLOCATION

Détection de compatibilité

Microsoft fournit l'outil HyperVisor Code Integrity Readiness Tool et le Device Guard Readiness Tool pour tester la compatibilité des drivers avant déploiement. En mode développement, les flags suivants dans le fichier INF du driver déclarent la compatibilité HVCI :

```
; Déclaration de compatibilité HVCI dans le fichier INF
[Version]
...
[Manufacturer]
%ManufacturerName%=Standard,NTamd64.10.0...16299

[Standard.NTamd64.10.0...16299]
%DeviceName%=MyDevice_Install, HID\VID_1234&PID_5678

[MyDevice_Install.NT]
CopyFiles=Drivers_Dir

[MyDevice_Install.NT.HW]
AddReg=MyDevice_AddReg

[MyDevice_AddReg]
; Déclare la compatibilité HVCI
HKR,,HypervisorEnforcedCodeIntegrity,0x00010001,1
```

Le Driver Verifier de Windows peut également tester la compatibilité en mode simulation :

```
:: Activer la vérification HVCI dans Driver Verifier
verifier /flags 0x02000000 /driver MyDriver.sys

:: Ou via l'interface graphique
verifier /standard /driver MyDriver.sys
:: Puis vérifier les résultats dans l'Event Log :
:: Applications and Services > Microsoft > Windows > CodeIntegrity
```

Implications pour les callbacks et hooks kernel

Les techniques de hooking kernel classiques sont largement neutralisées par HVCI. L'inline hooking — qui consiste à écrire un JMP au début d'une fonction kernel — est impossible car les pages de code sont en mode Read-Execute (pas d'écriture). L'IAT hooking est également bloqué si les tables d'import sont dans des sections de code protégées. Les seules techniques de monitoring kernel qui restent viables sous HVCI sont les mécanismes officiels :

- **Callbacks enregistrés** : PsSetCreateProcessNotifyRoutine, ObRegisterCallbacks, CmRegisterCallbackEx — ces API utilisent des structures de données (pas du code modifié) pour enregistrer les callbacks
- **Minifilters** : FltRegisterFilter pour le monitoring filesystem — architecture basée sur des callbacks, compatible HVCI
- **ETW (Event Tracing for Windows)** : monitoring kernel via des tracepoints compilés dans le noyau
- **WFP (Windows Filtering Platform)** : pour le monitoring réseau kernel

Cette contrainte affecte particulièrement les produits de sécurité endpoint (EDR) qui utilisaient traditionnellement des hooks kernel pour le monitoring. Sous HVCI, ces produits doivent migrer vers les APIs de callback officielles, ce qui peut réduire leur visibilité sur certaines activités kernel. C'est un compromis conscient : Microsoft préfère une surface d'attaque réduite (pas de hooks) avec un monitoring via des APIs officielles, plutôt qu'un monitoring extensif mais qui fragilise l'intégrité du code.

Bypasses documentés et recherche offensive

Malgré la robustesse de HVCI, la recherche en sécurité offensive a identifié plusieurs classes d'attaques qui restent viables — ou qui ont été viables avant d'être corrigées. Cette section passe en revue les techniques documentées dans la littérature académique et les conférences de sécurité.

Attaques data-only : le nouveau paradigme

Le contournement le plus fondamental de HVCI n'est pas une vulnérabilité de HVCI lui-même, mais un changement de stratégie. Les attaques data-only modifient les structures de données kernel sans exécuter de code non signé. Puisque HVCI protège l'intégrité du code (pages exécutables) mais pas l'intégrité des données (pages RW), un attaquant disposant d'une primitive d'écriture kernel peut :

- **Token manipulation** : modifier le champ `_TOKEN.Privileges` d'un processus pour obtenir SeDebugPrivilege ou SeTcbPrivilege, sans exécuter de code malveillant. L'attaquant localise la structure `_EPROCESS` de son processus, suit le pointeur vers le token, et modifie directement les bitmasks de privilèges.
- **EPROCESS manipulation** : modifier les liens de la liste chaînée `ActiveProcessLinks` pour cacher un processus, ou copier le token d'un processus SYSTEM vers un processus attaquant.
- **Manipulation des handles** : modifier la table des handles d'un processus pour ajouter un handle vers un objet protégé avec des droits FULL_ACCESS.
- **Object type manipulation** : modifier les callbacks des types d'objets kernel pour rediriger les appels système.

Ces techniques ont été extensivement documentées par les chercheurs de Microsoft et de Project Zero. La présentation de Connor McGarr à OffensiveCon 2023, « Turning Data-Only Attacks Into Complete Control », démontre comment construire des chaînes d'attaque complètes sans exécuter une seule instruction non signée.

ROP sous HVCI : gadgets dans le code signé

Le Return-Oriented Programming (ROP) pose un défi particulier à HVCI. Puisque les pages de code des drivers signés sont légitimement exécutables, les gadgets ROP au sein de ce code restent utilisables. Un attaquant peut construire une chaîne ROP en utilisant uniquement des séquences d'instructions (gadgets) trouvées dans ntoskrnl.exe, win32kfull.sys, ou d'autres drivers signés présents sur le système. HVCI ne peut pas bloquer l'exécution de ces gadgets car ils se trouvent dans du code validé.

Cependant, HVCI rend le ROP kernel significativement plus difficile en pratique :

- Le code auto-modifiant pour préparer les gadgets est impossible
- Les techniques de stack pivoting requièrent des gadgets spécifiques qui ne sont pas toujours disponibles
- kCFG (kernel Control Flow Guard) et kCET (kernel Control-flow Enforcement Technology) — présents sur les processeurs Intel 12th Gen+ et activés sous Windows 11 — ajoutent une couche de protection supplémentaire en validant les cibles des appels indirects (kCFG) et en utilisant des shadow stacks matérielles (kCET)
- XFG (eXtended Flow Guard) ajoute une validation du type des pointeurs de fonction

Exploitation de drivers signés légitimes (BYOVD sous HVCI)

HVCI n'empêche pas le chargement de drivers légitimement signés mais vulnérables — c'est la limite fondamentale du modèle. Un driver signé WHQL qui contient une vulnérabilité de lecture/écriture arbitraire sera chargé avec succès par HVCI, car sa signature est valide. L'attaquant peut ensuite exploiter la vulnérabilité pour des attaques data-only. Microsoft adresse ce problème avec la Microsoft Vulnerable Driver Blocklist — une liste de hashes de drivers connus vulnérables qui est mise à jour via Windows Update et consultée par HVCI lors du chargement.

```
# Vérifier la blocklist des drivers vulnérables
Get-CimInstance -Namespace root/Microsoft/Windows/CI `
    -ClassName MSFT_WDACBlockedDriver |
    Select-Object DriverFileName, BlockedReason, DateBlocked

# Mettre à jour la blocklist manuellement
# La blocklist est stockée dans :
# %SystemRoot%\System32\CodeIntegrity\driversipolicy.p7b
```

Désactivation de HVCI

La méthode la plus directe pour « contourner » HVCI est de le désactiver. Un attaquant disposant de droits administrateur peut modifier la configuration de boot :

```
:: Désactiver HVCI via bcdedit (requiert admin + reboot)
bcdedit /set hypervisorlaunchtype off

:: Ou via le registre
reg add
"HKLM\SYSTEM\CurrentControlSet\Control\DeviceGuard\Scenarios\HypervisorEnforcedCodeIntegri
ty" /v Enabled /t REG_DWORD /d 0 /f

:: Requiert un reboot pour prendre effet
```

Cependant, cette approche a des limitations : elle nécessite des droits administrateur et un reboot, ce qui est détectable par les solutions EDR. De plus, Secure Boot avec UEFI Lock peut empêcher la désactivation de HVCI en verrouillant la variable UEFI correspondante — la modification de bcdedit échouera si le lock UEFI est actif.

Table des techniques de contournement

Technique	Difficulté	Prérequis	Statut de mitigation
Data-only attacks (token/ EPROCESS manipulation)	Moyenne	Primitive R/W kernel arbitraire	Partiellement mitigé par kASLR, VBS enclaves
ROP kernel (gadgets dans code signé)	Haute	Contrôle du stack kernel + gadget catalog	Mitigé par kCFG, kCET, XFG
BYOVD (driver vulnérable signé)	Faible	Accès admin pour charger le driver	Driver blocklist (mise à jour continue)
Désactivation bcdedit	Faible	Admin + reboot	UEFI Lock, Secure Boot, détection EDR
VTL0 → VTL1 escape	Extrême	Bug dans l'hyperviseur ou Secure Kernel	Surface d'attaque minimale, bug bounty élevé
Attaque Secure Boot (bootkit)	Très haute	Vulnérabilité firmware ou clé de signature compromise	CVE-2022-21894 (BlackLotus) patché, DBX updates
Race condition lors du chargement	Haute	Timing précis + primitive R/W	Sérialisé par le Secure Kernel
Attaque matérielle (DMA via PCIe)	Haute	Accès physique + périphérique PCIe malveillant	VT-d / IOMMU requis, DMA Guard

Attaques sur le Secure Boot chain et implications pour HVCI

Une catégorie d'attaques particulièrement préoccupante cible non pas HVCI directement, mais sa chaîne de confiance en amont. HVCI dépend de l'intégrité de la séquence de boot : si un attaquant peut modifier le bootloader ou injecter du code avant le chargement de l'hyperviseur, il peut empêcher l'activation de HVCI ou le configurer avec des politiques permissives. Le bootkit BlackLotus (CVE-2022-21894) a démontré cette approche en exploitant une vulnérabilité dans le processus de révocation Secure Boot. L'attaquant utilisait un bootloader Windows légitime mais ancien (non révoqué dans la DBX) pour charger un environnement de boot modifié qui désactivait VBS et HVCI avant le démarrage du noyau.

Microsoft a répondu avec plusieurs mesures : mise à jour de la DBX (Secure Boot Forbidden Signature Database) pour révoquer les bootloaders vulnérables, introduction de la vérification de révocation basée sur SVN (Security Version Number) plutôt que sur des hashes individuels, et le déploiement progressif du Secure Launch (DRTM - Dynamic Root of Trust for Measurement) via Intel TXT ou AMD SKINIT. Le Secure Launch établit une racine de confiance dynamique qui mesure l'intégrité de l'hyperviseur et du Secure Kernel indépendamment du firmware UEFI, offrant une protection même si le firmware est compromis. Sur les machines Secured-core PC, le Secure Launch est activé par défaut et fournit une couverture complète de la chaîne de confiance depuis le matériel jusqu'à HVCI.

Attaques par canal auxiliaire et timing attacks

Les attaques par canal auxiliaire (side-channel attacks) représentent une classe de menaces qui transcende les protections logiques de HVCI. Puisque VTL0 et VTL1 partagent le même processeur physique (et donc les mêmes caches L1/L2/L3, les mêmes branch predictors, et les mêmes unités d'exécution), un attaquant en VTL0 peut théoriquement extraire des informations sur l'exécution de VTL1 via des techniques de type Spectre, Meltdown, ou MDS (Microarchitectural Data Sampling). Dans le contexte de HVCI, une attaque par canal auxiliaire pourrait potentiellement révéler des détails sur la politique CI active, les hashes de validation, ou les clés cryptographiques utilisées par MinCrypt en VTL1.

Les mitigations matérielles et microcode déployées par Intel et AMD depuis 2018 — IBRS (Indirect Branch Restricted Speculation), STIBP (Single Thread Indirect Branch Predictors), L1TF mitigations, et MDS buffer clearing — réduisent considérablement cette surface d'attaque. De plus, l'hyperviseur Hyper-V effectue un flush des buffers microarchitecturaux lors des transitions VTL, ce qui empêche la fuite de données VTL1 via les structures microarchitecturales partagées. Néanmoins, la recherche continue dans ce domaine, et de nouvelles variantes de side-channel attacks sont régulièrement découvertes — chacune nécessitant une évaluation de son impact potentiel sur l'isolation VTL.

CVEs notables et recherches récentes (2024-2026)

Plusieurs CVEs récentes ont mis en lumière les limites de l'approche HVCI :

- **CVE-2024-21305** : contournement de HVCI via une race condition dans la validation des politiques CI. Un attaquant pouvait soumettre une image valide pour validation, puis remapper les pages avant que le SLAT ne soit mis à jour. Corrigé dans le Patch Tuesday de janvier 2024 par la sérialisation du flux de validation.
- **CVE-2024-26169** : élévation de privilèges via le service Windows Error Reporting exploitée in-the-wild par le groupe Black Basta. L'attaque contournait HVCI en restant entièrement dans le domaine data-only, modifiant les clés de registre du service pour exécuter du code signé avec des privilèges SYSTEM.
- **Recherche Project Zero (2025)** : présentation par Ivan Googleman sur les « Phantom Pages » — une technique exploitant les différences de timing entre la validation CI et la mise à jour SLAT pour injecter du code malveillant dans une fenêtre de quelques microsecondes. Microsoft a répondu en ajoutant une vérification post-mapping en VTL1.
- **Publications SSTIC 2025** : travaux de l'ANSSI sur les attaques DMA contre VBS, montrant que sans IOMMU correctement configuré, un périphérique PCIe malveillant pouvait effectuer des lectures DMA dans la mémoire VTL1 sur certaines configurations. Corrigé par l'enforcement de VT-d dans les profils Secured-core.

HVCI et les performances

L'enforcement de HVCI introduit un overhead lié à la virtualisation et aux hypercalls. Cette section quantifie cet impact et décrit les optimisations introduites dans les versions récentes de Windows.

Sources d'overhead

L'overhead de HVCI provient de plusieurs sources distinctes :

- **Hypercalls pour la validation de code** : chaque chargement de driver ou module kernel nécessite un aller-retour VTL0 → Hyperviseur → VTL1 → Hyperviseur → VTL0. Coût : environ 5-15 microsecondes par validation. Impact négligeable en runtime (les drivers sont chargés au boot, pas en continu).
- **EPT walk overhead** : la translation d'adresse à deux niveaux (page tables guest + EPT) ajoute un surcoût à chaque TLB miss. Sur un accès mémoire qui cause un TLB miss complet, le CPU doit effectuer jusqu'à 24 accès mémoire (4 niveaux de pages guest × 4 niveaux EPT + accès final) au lieu de 4. Les TLB caches EPT modernes (VPID, PCID) réduisent significativement cet impact.
- **Context switches VTL** : le passage de VTL0 à VTL1 (et vice versa) implique une sauvegarde/restauration de l'état CPU complet. Coût : environ 2-5 microsecondes par transition sur du matériel moderne.
- **MBEC/GMET (avant optimisation)** : sur les processeurs sans MBEC/GMET, l'hyperviseur devait intercepter chaque transition kernel/user pour mettre à jour les permissions EPT dynamiquement. Cet overhead a été largement éliminé par le support matériel MBEC.

Benchmarks

Les mesures suivantes proviennent de benchmarks effectués sur du matériel de classe serveur (Intel Xeon Scalable 4th Gen, Sapphire Rapids) avec Windows Server 2025 et des benchmarks desktop (Intel Core Ultra, Arrow Lake) avec Windows 11 24H2 :

Benchmark	Sans HVCI	Avec HVCI	Overhead	Notes
Boot time (cold boot)	12.3s	13.1s	+6.5%	Validation des drivers au démarrage
Driver load time (moyenne)	2.1ms	4.8ms	+128%	Significatif en absolu mais rare en runtime
Latence syscall (NtCreateFile)	1.42µs	1.48µs	+4.2%	Overhead EPT walk sur TLB miss
Throughput I/O séquentiel (NVMe)	6.8 GB/s	6.6 GB/s	-2.9%	Impact minimal sur les I/O
Context switch (thread kernel)	0.98µs	1.05µs	+7.1%	EPT walk additionnel
Compilation kernel (Linux sous WSL2)	47.2s	49.1s	+4.0%	Workload CPU-bound, impact modéré
Gaming (FPS moyens, DX12)	142 fps	137 fps	-3.5%	Réduit par MBEC sur matériel récent
Database (SQL Server TPC-C)	98,400 tps	95,100 tps	-3.4%	Impact acceptable pour la plupart des workloads

Optimisations Windows 11 24H2 et Server 2025

Microsoft a continuellement optimisé l'impact de HVCI sur les performances :

- **MBEC natif** : sur les processeurs Intel 12th Gen+ et AMD Zen 4+, les transitions kernel/user ne nécessitent plus de mise à jour EPT dynamique. Cela élimine des dizaines de milliers de VM-Exits par seconde sur les workloads lourds.
- **Large EPT Pages** : utilisation de pages EPT de 2 Mo et 1 Go pour réduire la profondeur du page walk EPT. Le nombre d'accès mémoire par TLB miss passe de 24 à 15-20 selon l'alignement.
- **Lazy SLAT updates** : au lieu de mettre à jour les permissions SLAT page par page, l'hyperviseur batch les modifications et les applique en une seule opération. Cela réduit le nombre d'hypercalls lors du chargement de gros drivers.
- **VPID/PCID caching amélioré** : meilleure utilisation des identifiants de processus virtuels pour réduire les TLB flushes lors des transitions VTL. Windows Server 2025 a réduit les TLB flushes liés à VBS de 40% par rapport à Server 2022.
- **Optimisation du scheduler VTL1** : le scheduler Windows a été optimisé pour minimiser les transitions VTL inutiles, en regroupant les opérations VTL1 et en utilisant des signaux asynchrones quand possible.

Quand désactiver HVCI ?

Dans la majorité des scénarios, l'overhead de 2-5% est acceptable au regard du gain de sécurité. Les cas où la désactivation peut être justifiée incluent :

- Les postes de travail de développement kernel où le cycle build-test-debug nécessite un chargement fréquent de drivers de test (incompatibles avec HVCI)
- Les serveurs à très haute performance (trading haute fréquence) où chaque microseconde de latence compte
- Les systèmes legacy avec des drivers incompatibles HVCI qui ne disposent pas de mises à jour
- Les environnements de virtualisation imbriquée (nested virtualization) où l'overhead cumulé devient significatif

Dans tous les autres cas — et en particulier pour les postes de travail d'entreprise, les serveurs exposés, et les machines traitant des données sensibles — HVCI devrait être activé.

Configuration et vérification

Le déploiement de HVCI en entreprise nécessite une planification soignée pour éviter les problèmes de compatibilité. Cette section détaille les méthodes de configuration, de vérification et de dépannage.

Activation de HVCI

HVCI peut être activé par plusieurs méthodes, selon le contexte de déploiement :

```
# Méthode 1 : Via le registre (requiert reboot)
reg add
"HKLM\SYSTEM\CurrentControlSet\Control\DeviceGuard\Scenarios\HypervisorEnforcedCodeIntegri
ty" /v Enabled /t REG_DWORD /d 1 /f
reg add "HKLM\SYSTEM\CurrentControlSet\Control\DeviceGuard" /v
EnableVirtualizationBasedSecurity /t REG_DWORD /d 1 /f
reg add "HKLM\SYSTEM\CurrentControlSet\Control\DeviceGuard" /v
RequirePlatformSecurityFeatures /t REG_DWORD /d 3 /f
# 1 = Secure Boot, 3 = Secure Boot + DMA Protection

# Méthode 2 : Via Group Policy
# Computer Configuration > Administrative Templates > System > Device Guard
# → Turn on Virtualization Based Security : Enabled
# → Virtualization Based Protection of Code Integrity :
#    "Enabled with UEFI lock" (recommandé) ou "Enabled without lock"

# Méthode 3 : Via DISM (pour les images de déploiement)
DISM /Online /Enable-Feature /FeatureName:Microsoft-Hyper-V-Hypervisor /All
DISM /Online /Enable-Feature /FeatureName:IsolatedUserMode /All

# Méthode 4 : Via Intune (MDM)
# Device Configuration > Endpoint Protection >
# Microsoft Defender Credential Guard > Enable with UEFI lock
# Memory Integrity > Enable
```

Vérification de l'état HVCI

Plusieurs méthodes permettent de vérifier que HVCI est actif et fonctionnel :

```
# PowerShell : interrogation WMI/CIM
$dg = Get-CimInstance -ClassName Win32_DeviceGuard `
    -Namespace root\Microsoft\Windows\DeviceGuard
$dg | Format-List *

# Propriétés clés à vérifier :
# VirtualizationBasedSecurityStatus : 2 = Running
# SecurityServicesRunning : contient 2 (HVCI)
#   1 = Credential Guard
#   2 = HVCI (Memory Integrity)
#   3 = System Guard Secure Launch
# RequiredSecurityProperties :
#   1 = Hypervisor
#   2 = Secure Boot
#   4 = DMA Protection
# AvailableSecurityProperties :
#   Vérifie ce que le matériel supporte

# Vérification rapide via msinfo32
# Sécurité basée sur la virtualisation : En cours d'exécution
# Services de sécurité VBS en cours d'exécution :
#   Intégrité du code appliquée par l'hyperviseur

# Vérification via bcdedit
bcdedit /enum | findstr /i "hypervisor"
# hypervisorlaunchtype    Auto → VBS actif
```

Dépannage

Les problèmes les plus courants lors du déploiement de HVCI sont liés à l'incompatibilité des drivers. Les événements suivants dans les journaux Windows permettent de diagnostiquer les problèmes :

```
# Event Log : Microsoft-Windows-CodeIntegrity/Operational
# Event ID 3089 : Driver bloqué par HVCI
# Event ID 3099 : Politique CI chargée avec succès
# Event ID 3033 : Code Integrity détermine qu'un processus a tenté de charger
#                  un driver non conforme à la politique

# Lister les drivers incompatibles HVCI sur le système actuel
Get-WindowsDriver -Online | Where-Object {
    $_.ClassName -ne $null -and
    (Test-Path $_.OriginalFileName)
} | ForEach-Object {
    $compatible = $true
    $binary = [System.IO.File]::ReadAllBytes($_.OriginalFileName)
    # Vérification simplifiée - en production utiliser l'outil MS
    [PSCustomObject]@{
        Driver = $_.Driver
        FileName = $_.OriginalFileName
        ClassName = $_.ClassName
    }
}

# Commande réelle pour la vérification de compatibilité :
# Télécharger HVCI Readiness Tool depuis Microsoft
# ou utiliser l'outil intégré Windows 11 :
Get-CimInstance -Namespace root/Microsoft/Windows/CI `
    -ClassName MSFT_WDACBlockedDriver 2>$null |
    Format-Table DriverFileName, BlockedReason
```

Analyse avancée avec WinDbg

Pour les chercheurs en sécurité et les développeurs kernel, WinDbg offre une visibilité sur les structures internes de HVCI. Attention : les commandes liées à VTL1 nécessitent un débogage de type « Secure Kernel Debugging » qui requiert une configuration spécifique.

```
// WinDbg — Commandes utiles pour l'analyse HVCI

// Vérifier l'état de VBS et HVCI
!sysinfo machineid
!analyze -show DeviceGuardState

// Examiner les entrées EPT (requiert un debugger hyperviseur)
// Sur un système live, on peut inférer l'état SLAT via les PTE :
!pte [adresse_virtuelle]
// Si les permissions PTE montrent Execute mais pas Write → HVCI actif

// Lister les modules kernel et leur état de validation CI
lm k v
// Le champ "Flags" indique si le module a été validé par CI

// Examiner la politique CI active
!object \Device\SiPolicy
dt nt!_SYSTEM_INFORMATION_CLASS SystemCodeIntegrityInformation

// Examiner les callbacks CI
x ci!CiValidateImageHeader
x ci!CipQueryPolicyInformation

// Vérifier si un pool allocation est NX
!pool [adresse]
// Le type de pool indiquera NonPagedPoolNx si HVCI-compatible

// Examiner les VTL d'un processeur virtuel
!pcr
dt nt!_KPRCB @$prcb VirtualTrustLevel

// Examiner les hypercalls récents (si le tracing est activé)
!hvcall -last 20
```

FAQ

HVCI protège-t-il contre les attaques de type Return-Oriented Programming (ROP) dans le noyau ?

HVCI n'empêche pas directement le ROP kernel, car les gadgets ROP se trouvent dans du code légitimement signé qui est marqué exécutable. Cependant, HVCI renforce considérablement l'efficacité des mitigations anti-ROP complémentaires. En empêchant l'injection de code non signé, HVCI force les attaquants à utiliser uniquement des gadgets présents dans le code signé existant — un espace de recherche plus restreint. De plus, les technologies kCFG (kernel Control Flow Guard) et kCET (kernel Control-flow Enforcement Technology), qui s'appuient sur des shadow stacks matérielles, complètent HVCI en rendant les chaînes ROP extrêmement difficiles à construire. Sur un système Windows 11 moderne avec HVCI + kCFG + kCET, une attaque ROP kernel viable nécessiterait de trouver des gadgets qui satisfont simultanément les contraintes de signature, de control flow, et de shadow stack — ce qui réduit drastiquement la surface d'attaque exploitable.

Quelle est la différence entre HVCI et Secure Boot, et pourquoi les deux sont-ils nécessaires ?

Secure Boot et HVCI protègent des phases différentes du cycle de vie du système. Secure Boot valide l'intégrité de la chaîne de boot — du firmware UEFI jusqu'au bootloader Windows (winload.efi) et au noyau initial. Il empêche le chargement de bootkits ou de bootloaders modifiés. Cependant, Secure Boot ne protège pas le runtime : une fois le système démarré, un exploit kernel peut charger un driver malveillant ou modifier du code en mémoire. HVCI prend le relais à ce stade en protégeant l'intégrité du code pendant toute la durée d'exécution du système. Les deux sont complémentaires et nécessaires : sans Secure Boot, un attaquant pourrait modifier le bootloader pour désactiver HVCI avant son chargement. Sans HVCI, un attaquant pourrait exploiter le noyau après un boot sécurisé. L'option « UEFI Lock » de HVCI renforce cette complémentarité en stockant la configuration HVCI dans une variable UEFI protégée par Secure Boot, empêchant sa désactivation sans accès physique.

Un driver signé WHQL mais vulnérable peut-il être exploité sous HVCI ? Comment Microsoft adresse-t-il ce problème ?

Oui, c'est la principale limitation de HVCI. Un driver avec une signature WHQL valide sera chargé avec succès par HVCI, même s'il contient des vulnérabilités exploitables. HVCI vérifie l'authenticité du code (est-il signé par un éditeur de confiance ?) mais pas sa qualité ou sa sécurité. Pour adresser ce problème, Microsoft maintient la Microsoft Vulnerable Driver Blocklist — une liste de hashes de drivers connus vulnérables qui est mise à jour via Windows Update et automatiquement activée sur les systèmes avec HVCI. Cette liste inclut des centaines de drivers vulnérables connus, dont les drivers BYOVD les plus exploités (RTCore64.sys, dbutil_2_3.sys, etc.). Les limitations de cette approche sont : la blocklist est réactive (un nouveau driver vulnérable n'est bloqué qu'après sa découverte et son ajout à la liste), et un attaquant peut potentiellement trouver un driver vulnérable non encore listé. Microsoft complète cette approche avec des initiatives comme le programme de signature renforcé (qui inclut des analyses de sécurité avant la signature WHQL) et les attestations de conformité des éditeurs de drivers.

Est-il possible d'exécuter un débogage kernel complet sur un système avec HVCI activé, et quelles sont les limitations ?

Le débogage kernel standard (via WinDbg connecté en série, USB, ou réseau) fonctionne sur un système avec HVCI activé, mais avec certaines limitations. Le débogueur peut inspecter l'état de VTL0 — mémoire kernel, structures de données, registres, breakpoints — normalement. Cependant, le débogueur VTL0 ne peut pas accéder à la mémoire VTL1 (Secure Kernel, trustlets IUM). Pour déboguer le Secure Kernel ou les composants VTL1, il faut activer le « Secure Kernel Debugging » via `bcdedit /set {default} securekerneldebugging on`, qui nécessite une connexion de débogage séparée et un niveau d'accès supérieur. De plus, sous HVCI, les hardware breakpoints dans les pages de code requièrent une coordination avec l'hyperviseur (car la modification des registres DR implique un VM-Exit). Les software breakpoints (`int 3`) ne peuvent pas être écrits dans les pages de code protégées HVCI car elles sont en mode Read-Execute. WinDbg utilise les debug registers matériels (DR0-DR3) comme fallback, mais ceux-ci sont limités à 4 breakpoints simultanés. Pour le développement de drivers sous HVCI, Microsoft

recommande d'utiliser KDNET (débogage réseau) avec les extensions Debug Transport pour minimiser l'impact sur le système, et de développer en mode test (HVCI désactivé) avant de valider la compatibilité finale avec HVCI activé.

Conclusion

HVCI représente un changement de paradigme dans la sécurité kernel Windows. En déplaçant la validation de l'intégrité du code depuis le noyau VTL0 — intrinsèquement compromettable — vers un environnement isolé par l'hyperviseur en VTL1, Microsoft a transformé une protection logicielle contournable en un enforcement matériel. La combinaison de VBS, SLAT/EPT, et W^X crée une architecture où même un noyau entièrement compromis ne peut pas exécuter de code non signé.

Cependant, HVCI n'est pas une solution universelle. Les attaques data-only — modification de tokens, manipulation d'EPROCESS, abus de handles — restent viables car HVCI protège l'intégrité du code, pas l'intégrité des données. Le ROP utilisant des gadgets dans du code signé reste théoriquement possible, bien que considérablement compliqué par kCFG et kCET. Et la technique BYOVD continue de fonctionner pour les drivers vulnérables non encore ajoutés à la blocklist.

Pour les défenseurs, HVCI est un composant essentiel d'une stratégie de défense en profondeur qui inclut Secure Boot, Credential Guard, WDAC, et une politique de mise à jour rigoureuse de la driver blocklist. Pour les red teamers et chercheurs offensifs, HVCI élève significativement la barre d'attaque : les techniques classiques d'injection de code kernel sont obsolètes, et les attaques viables nécessitent des primitives plus sophistiquées et une compréhension approfondie des structures de données internes du noyau.

Pour approfondir les mécanismes internes de VBS et HVCI, les ressources suivantes sont incontournables : la documentation Microsoft Learn sur HVCI, les publications du Google Project Zero sur les attaques VBS, et les papiers présentés aux conférences Black Hat et SSTIC sur les bypasses hyperviseur. L'avenir de HVCI s'inscrit dans une tendance plus large vers les architectures de confiance réduites (reduced trust computing), où chaque composant du système fonctionne avec le minimum de privilèges nécessaire, et où l'isolation matérielle remplace progressivement la confiance logicielle. Les processeurs de nouvelle génération, avec leur support natif de MBEC, CET, et des enclaves hardware, ne feront qu'accélérer cette transition.

Points clés à retenir

- **HVCI déplace la validation du code** depuis le noyau VTL0 vers le Secure Kernel VTL1, isolé par l'hyperviseur Hyper-V. Même un noyau VTL0 compromis ne peut pas exécuter de code non signé.
- **L'enforcement W^X via SLAT/EPT** garantit qu'une page mémoire kernel ne peut jamais être simultanément inscriptible et exécutable. Ce principe est appliqué au niveau matériel et ne peut pas être contourné par du code logiciel.
- **La validation de signature** est effectuée par CI.dll en VTL1, avec une bibliothèque crypto autonome (MinCrypt) qui ne dépend d'aucun composant VTL0.
- **Les attaques data-only restent viables** car HVCI protège l'intégrité du code mais pas les structures de données kernel. La modification de tokens, d'EPROCESS, ou de handles ne nécessite pas d'exécuter du code non signé.
- **BYOVD fonctionne toujours** pour les drivers signés vulnérables non présents dans la blocklist Microsoft. La combinaison HVCI + blocklist + WDAC est nécessaire pour une protection maximale.
- **L'overhead est acceptable** : 2-5% sur les workloads courants avec du matériel supportant MBEC. Le temps de chargement des drivers augmente significativement mais n'impacte pas le runtime.
- **kCFG + kCET + HVCI** constituent ensemble la meilleure protection actuelle contre l'exécution de code arbitraire dans le noyau Windows.

Ayi NEDJIMI Consultants — Expert cybersécurité offensive & intelligence artificielle

ayinedjimi-consultants.fr · ayi@ayinedjimi-consultants.fr