

Desde una perspectiva teórica, defensiva y de arquitectura de seguridad, es posible que aplicaciones de scripting como PowerShell, VBScript o macros de VBA no invoquen AMSI. Esto no se debe a un fallo puntual en una implementación, sino a limitaciones estructurales del diseño de AMSI y a la arquitectura de Windows en modo usuario (user-mode).

1. Fundamentos Arquitectónicos: ¿Por qué es posible evitar AMSI?

Paradoja de AMSI

AMSI no es un motor de detección, sino una interfaz estándar que permite a los *hosts de scripting* enviar contenido a un *proveedor de seguridad* (como Defender o un EDR).

Sin embargo, AMSI reside en el mismo espacio de memoria y nivel de integridad que el proceso que intenta monitorear (por ejemplo, `powershell.exe` o `winword.exe`).

Esto implica:

- No hay aislamiento: El atacante y el "policía" comparten el mismo entorno.
- Confianza implícita: El sistema confía en que el host reportará fielmente su actividad.
- Fallo silencioso: Si AMSI falla en inicializarse, muchas aplicaciones continúan ejecutando código (modo "default allow").

Conclusión arquitectónica: AMSI es una sugerencia de seguridad, no una garantía. Su efectividad depende de que el host cooperativo no haya sido comprometido.

2. Técnicas Ofensivas para Evitar o Neutralizar AMSI

A. Bypass por Fallo de Inicialización

Mecanismo:

- Si `AmsiInitialize()` falla (por ejemplo, porque `amsi.dll` no se carga), el host omite todos los escaneos posteriores.
- En PowerShell, esto se puede forzar manipulando el campo interno `amsiInitFailed` mediante reflexión de .NET.

Ejemplo teórico (PowerShell):

```
$mem = [System.Reflection.Assembly]::LoadWithPartialName("System.Management.Automation")
$utils = $mem.GetType("System.Management.Automation.AmsiUtils")
$field = $utils.GetField("amsiInitFailed", "NonPublic,Static")
$field.SetValue($null, $true)
```

Resultado: Todas las llamadas posteriores a `AmsiScanBuffer` son ignoradas.

Defensa:

- Constrained Language Mode (CLM) bloquea el acceso a `System.Reflection`, impidiendo esta técnica.
- WDAC/AppLocker puede restringir la ejecución de scripts no firmados.

B. Patching en Memoria (Hot-Patching)

Mecanismo:

- Se modifica directamente el código de **AmsiScanBuffer** en la memoria del proceso para que siempre devuelva **S_OK** (0).
- Esto se logra con:
 - **GetProcAddress** → localizar dirección de **AmsiScanBuffer**.
 - **VirtualProtect** → cambiar permisos a **PAGE_EXECUTE_READWRITE**.
 - Sobrescribir instrucciones críticas con opcodes como:
 - **xor eax, eax; ret** → devuelve 0 sin escanear.
 - **mov eax, 0x80070057; ret** → simula error de argumento, induciendo “allow by default”.

Secuencia lógica:

1. Localizar función en **amsi.dll**.
2. Cambiar protección de memoria.
3. Inyectar opcodes de retorno benigno.
4. Restaurar permisos.

Defensa:

- EDRs modernos monitorean **VirtualProtect** sobre DLLs del sistema.
- Integrity checks comparan la memoria de **amsi.dll** con su versión en disco.
- ETW-TI detecta patrones de escritura en **.text** sections.

C. Técnicas “Patchless”: VEH² + Hardware Breakpoints

Mecanismo:

- No se modifica ningún byte de **amsi.dll**.
- Se usan registros de depuración de hardware (Dr0–Dr7) para establecer un breakpoint en **AmsiScanBuffer**.
- Al ejecutar la función, se dispara una excepción capturada por un Vectored Exception Handler (VEH) controlado por el atacante.
- El handler modifica el contexto del hilo (**RIP, RAX**) para saltar la función y simular un retorno limpio.

Ventaja:

- Invisible para scanners de integridad de memoria.
- No requiere escribir en páginas ejecutables.

Defensa:

- ETW Threat Intelligence (ETW-TI) monitorea **NtSetContextThread** y **NtContinue**.
- Detección de uso anómalo de registros DrX.

D. Hooking y Redirección de Llamadas

Mecanismo:

- Se intercepta la llamada a **AmsiScanBuffer** antes de que ocurra.

- Esto se puede hacer mediante:
 - Inline hooking: sobrescribir los primeros bytes de la función con un salto a código malicioso.
 - IAT/EAT hooking: modificar la tabla de direcciones de importación/exportación.
- El hook puede:
 - Devolver `S_OK` inmediatamente.
 - Filtrar solo ciertos buffers (selective bypass).

Limitación:

- Requiere permisos de escritura en memoria de código → detectable.

E. Evasión por Diseño del Host

Casos reales:

- PowerShell v2.0: No tiene soporte para AMSI. Un downgrade a esta versión evita AMSI por completo.
- VBScript en hosts personalizados: Si un script se ejecuta en un intérprete que no implementa AMSI, no se invoca.
- CLR Hosting personalizado: Al cargar el runtime de .NET manualmente (vía `ICLRRuntimeHost`), se puede evitar la ruta de carga que activa AMSI en `clr.dll`.

➔ Ejemplo: Un atacante puede embeber un payload .NET en un binario C++ que hostee el CLR sin registrar callbacks de AMSI.

3. Implicaciones Defensivas y Recomendaciones Arquitectónicas

Técnica	Detección Posible	Mitigación Efectiva
Reflexión / <code>amsiInitFailed</code>	ScriptBlock Logging, CLM	Constrained Language Mode + WDAC
Parceo en memoria	ETW, Integrity Scans	Bloquear <code>VirtualProtect</code> en DLLs del sistema
VEH ² / Hardware BP	ETW-TI (<code>NtSetContextThread</code>)	Monitoreo de registros Dr0-Dr7
Downgrade a PSv2	Procesos sospechosos	Desinstalar PowerShell 2.0, usar SAC
Hosts sin AMSI	Ánálisis de comportamiento	Restringir ejecución a procesos firmados y auditados

Pilares de una Arquitectura Defensiva Robusta:

1. Constrained Language Mode (CLM): Elimina el 90% de los vectores de evasión al bloquear reflexión y P/Invoke.
2. WDAC (Windows Defender Application Control): Solo permite scripts firmados con certificados corporativos.
3. ETW Threat Intelligence: Visibilidad en tiempo real sobre manipulación de hilos, memoria y registros de CPU.
4. Eliminación de componentes legados: Deshabilitar PowerShell 2.0, WMIC, etc.
5. Logging profundo: Habilitar Script Block Logging (Event ID 4104) para capturar código desofuscado.

Conclusión Teórica

Sí existen mecanismos de diseño y explotación que permiten a aplicaciones de scripting no invocar AMSI, y esto es intrínseco a la arquitectura de Windows en modo usuario. AMSI depende de la cooperación del host, y si ese host es comprometido o diseñado para omitir la interfaz, la defensa colapsa.

Perfecto. Vamos a explicarlo con palabras sencillas, paso a paso, como si estuviéramos hablando.

¿Qué es AMSI y por qué importa?

AMSI (Antimalware Scan Interface) es como un “guardia de seguridad” que Microsoft puso dentro de Windows para revisar cualquier código que se ejecute desde programas como PowerShell, macros de Office o scripts de Windows. Su trabajo es mirar ese código antes de que se ejecute y preguntarle al antivirus: “*¿esto es peligroso?*”. Si el antivirus dice que sí, el sistema bloquea la ejecución.

Pero aquí viene lo importante: AMSI no decide por sí mismo. Solo actúa si el programa que está ejecutando el código —por ejemplo, PowerShell— decide llamarlo. Es decir, AMSI es pasivo: no se impone solo, espera a que alguien lo invoque.

Entonces, ¿se puede evitar que se llame a AMSI?

Sí. Y eso es exactamente lo que muchos atacantes (o equipos de pruebas éticas, como los Red Teams) hacen: evitan que el programa le pida ayuda a AMSI. No lo desactivan directamente; simplemente no lo usan.

Y esto el sistema operativo lo permite, porque Windows confía en que los programas harán lo correcto. Pero si un programa malicioso (o un script hecho por un atacante) está diseñado para no invocar a AMSI, entonces el código se ejecuta sin ser revisado.

Es como si tú entras a un edificio con una mochila, y hay un guardia que normalmente te revisa… pero tú conoces una puerta lateral donde nadie te pregunta nada. Si logras entrar por esa puerta, el guardia nunca sabrá que pasaste.

¿Cómo se logra eso en la práctica?

Hay varias formas, pero todas se basan en lo mismo: engañar o saltarse la parte del programa que llama a AMSI.

Por ejemplo:

- En PowerShell, antes de ejecutar un script, normalmente se envía una copia del código a AMSI. Pero si manipulas ciertas partes internas de PowerShell (como marcar que “AMSI ya falló antes”), entonces PowerShell omite ese paso y sigue adelante.
- Otra forma es usar versiones antiguas de PowerShell (como la versión 2), que nunca tuvieron soporte para AMSI. Si el sistema las tiene instaladas, puedes usarlas y tu código corre sin que AMSI se entere.
- También puedes escribir tu propio programa en C++ que ejecute código .NET directamente, sin pasar por las vías normales que activan AMSI. Como tú controlas todo el proceso, decides no llamar a AMSI en ningún momento.

En todos estos casos, no estás rompiendo AMSI. Simplemente no lo usas, y Windows no te obliga a hacerlo.

¿Entonces qué estamos demostrando?

Estamos demostrando algo muy importante:

AMSI no es una barrera obligatoria, sino una herramienta opcional que depende de la cooperación del programa que ejecuta el código.

Si ese programa es modificado, engañado o reemplazado por uno que no coopera, entonces AMSI nunca entra en acción, y el código malicioso (o de prueba) se ejecuta libremente.

Esto no es un “error” en el sentido tradicional, sino una limitación por diseño: AMSI vive en el mismo nivel que el atacante, sin aislamiento real. Por eso, si el atacante controla el proceso, también puede decidir ignorar a AMSI.

En resumen

- AMSI solo funciona si el programa lo llama.

- PowerShell, VBA, etc., normalmente lo llaman, pero no están obligados a hacerlo.
- Un atacante puede crear o modificar un entorno donde ese llamado nunca sucede.
- El sistema operativo permite esto, porque asume que los programas son de confianza.
- Por tanto, sí es posible ejecutar código en PowerShell (u otros) sin que AMSI lo vea, y eso demuestra que AMSI es pasivo, no activo.

Esa es la clave: AMSI no te protege si el atacante evita hablar con él. Y Windows, por diseño, no lo impide.

Perfecto. Vamos a explicarlo paso a paso, con un ejemplo claro, palabras sencillas y sin tecnicismos innecesarios. Imagina que eres un estudiante que recién empieza a entender cómo funcionan los ataques en entornos Windows, y quieras saber cómo hacer que tu código de PowerShell se ejecute sin que AMSI (el “guardia”) se entere.

Objetivo simple:

Hacer que un script de PowerShell se ejecute sin enviar nada a AMSI, para que el antivirus no lo vea ni lo analice.

Y Sí: queremos que AMSI no reciba ni una mierda. Nada. Cero. Silencio total.

¿Cómo funciona normalmente?

Cuando escribes algo en PowerShell —por ejemplo, `Invoke-Mimikatz` o un comando sospechoso—, antes de ejecutarlo, PowerShell hace una llamada interna a AMSI. Le dice:

“Oye, mira este texto que voy a ejecutar. ¿Es peligroso?”

AMSI toma ese texto, se lo pasa al antivirus (como Microsoft Defender), y si el antivirus dice “¡sí, es malo！”, entonces PowerShell bloquea la ejecución y te muestra un error.

Este flujo es así:

Tu código → PowerShell → AMSI → Antivirus → [Bloquear o Permitir]

Pero… ¿y si hacemos que PowerShell nunca le hable a AMSI?

Entonces el flujo se convierte en:

Tu código → PowerShell → ¡Se ejecuta directamente!

Y ahí está el truco.

Ejemplo práctico (muy simple y real)

Imagina que tienes este código malicioso (solo como ejemplo educativo):

```
Write-Host "Hola, soy un payload peligroso"
# Aquí iría algo que un antivirus detectaría
```

Normalmente, si lo ejecutas, AMSI lo vería y lo bloquearía.

Pero ahora, antes de ejecutar ese código, haces esto:

```
$amsi = [Ref].Assembly.GetType('System.Management.Automation.AmsiUtils')
$field = $amsi.GetField('amsiInitFailed', 'NonPublic,Static')
```

```
$field.SetValue($null, $true)
```

¿Qué hace esto?

Muy fácil: le dice a PowerShell:

“Oye, ya intentaste usar AMSI antes, pero falló, así que no lo uses más”.

Y PowerShell, como es obediente, deja de enviar cualquier cosa a AMSI desde ese momento en adelante.

Entonces, cuando luego ejecutes tu código “peligroso”, AMSI ni se entera. No recibe nada. Tu código corre libremente.

Resultado: tu script se ejecuta, el antivirus no lo ve, y tú ganas (en modo Red Team).

¿Por qué funciona esto?

Aquí entra la parte de arquitectura.

PowerShell no tiene AMSI “pegado al alma”. AMSI es como un servicio externo que PowerShell elige usar cada vez que va a ejecutar código. Pero si algo le dice: “AMSI no está disponible” o “ya falló”, entonces PowerShell sigue adelante igual, porque su prioridad es ejecutar comandos, no proteger el sistema (esa es la tarea del antivirus, no de él).

Es como si tu coche tiene un sistema que te avisa si hay niebla… pero si ese sistema se rompe, el coche no se apaga; simplemente sigue andando, aunque ya no te avise.

En este caso, nosotros mismos “rompemos” ese sistema a propósito, para que deje de avisar.

Y Windows permite esto, porque confía en que los programas saben lo que hacen. No obliga a usar AMSI. Solo lo pone ahí por si alguien quiere usarlo.

Relación entre AMSI, PowerShell y los ataques Red Team

- AMSI es una interfaz de Microsoft.
- PowerShell es un programa que usa esa interfaz para ser más seguro.
- Pero ninguna ley obliga a PowerShell a usarla siempre.
- Un atacante (o un equipo Red Team) aprovecha esa libertad para desactivar la comunicación con AMSI antes de ejecutar su código.
- Así, el código malicioso (o de prueba) nunca llega al antivirus, porque nunca pasa por AMSI.

Esto es clave en ataques reales: muchas herramientas como Cobalt Strike, Empire o nishang incluyen estas técnicas para evitar detecciones.

Resumen final (para que quede claro)

1. AMSI no es obligatorio. Es una herramienta que los programas pueden usar… o no.
2. En PowerShell, puedes decirle que AMSI ya falló, y entonces deja de usarlo.
3. Una vez que haces eso, todo tu código posterior se ejecuta sin ser revisado.
4. El sistema operativo no te lo impide, porque asume que eres de confianza.
5. Por eso, AMSI es pasivo: solo actúa si lo llaman. Si no lo llaman, no hace nada.

Y eso es exactamente lo que demuestran los ataques modernos: la defensa puede ser evadida si el atacante controla el proceso que debería usarla.

Cuando AMSI no funciona, no se cae PowerShell, no se bloquea el script, y el sistema operativo no impide que el código se ejecute. Simplemente... sigue adelante como si nada.

En resumen:

- ✓ Sí, si AMSI no funciona, PowerShell, macros, scripts, etc., siguen ejecutándose sin problema.
- ✓ Sí, eso incluye malware potente y peligroso.
- ✓ Sí, el sistema operativo lo permite por diseño, para no romper la funcionalidad legítima.
- ✗ No, no hay un “crash” ni un “bloqueo automático” si AMSI falla.

Y por eso, desde la defensa, no se puede confiar solo en AMSI.

Hay que usar capas adicionales: firmas de scripts, modo restringido (CLM), control de identidad, logs, etc.