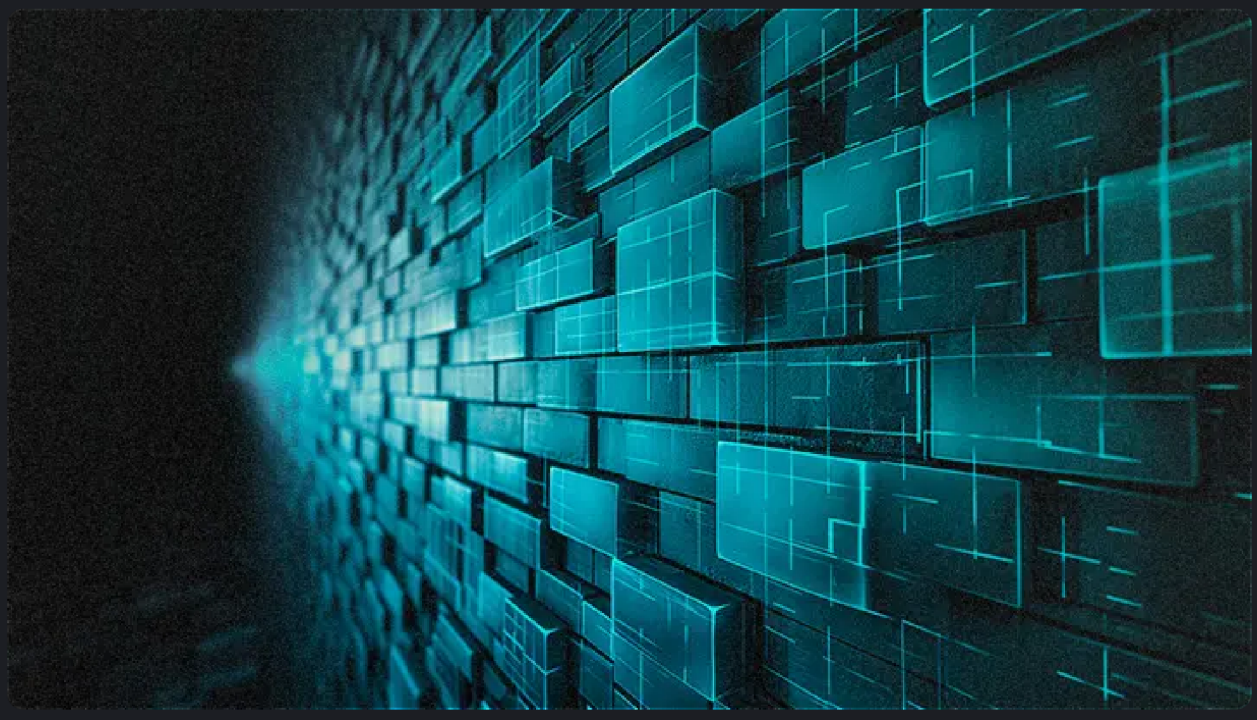
13 SEPTEMBER 2024

JOHN UHLMANN

Kernel ETW is the best ETW

This research focuses on the importance of native audit logs in secure-by-design software, emphasizing the need for kernel-level ETW logging over user-mode hooks to enhance anti-tamper protections.

14 min read Perspectives



Preamble

A critical feature of secure-by-design software is the generation of audit logs when

privileged operations are performed. These native audit logs can include details of the internal software state, which are impractical for third-party security vendors to bolt on after the fact.

Most Windows components generate logs using **Event Tracing for Windows** (ETW). These events expose some of Windows's inner workings, and there are scenarios when endpoint security products benefit from subscribing to them. For security purposes, though, not all ETW providers are created equal.

The first consideration is typically the reliability of the event provider itself - in particular, where the logging happens. Is it within the client process and **trivially vulnerable to ETW tampering**? Or is it perhaps slightly safer over in an RPC server process? Ideally, though, the telemetry will come from the **kernel**. Given the user-to-kernel security boundary, this provides stronger anti-tamper guarantees over in-process telemetry. This is Microsoft's recommended approach. Like Elastic Endpoint, Microsoft Defender for Endpoint also uses kernel ETW in preference to fragile user-mode `ntdll` hooks.

For example, an adversary might be able to easily avoid an in-process user-mode hook on `ntdll!NtProtectVirtualMemory`, but bypassing a kernel **PROTECTVM** ETW event is significantly harder. Or, at least, **it should be**.

The Security Event Log is effectively just persistent storage for the events from the Microsoft-Windows-Security-Auditing ETW provider. Surprisingly, **Security Event 4688** for process creation is not a kernel event. The kernel dispatches the data to the Local Security Authority (`lsass.exe`) service, emitting an ETW event for the Event Log to consume. So, the data could be tampered with from within that server process. Contrast this with the `ProcessStart` event from the Microsoft-Windows-Kernel-Process provider, which is logged directly by the kernel and requires kernel-level privileges to interfere with.

The second consideration is then the reliability of the information being logged. You might trust the event source, but what if it is just **blindly logging client-supplied data** that is extrinsic to the event being logged?

In this article, we'll focus on kernel ETW events. These are typically the most security-relevant because they are difficult to bypass and often pertain to privileged actions being performed on behalf of a client thread.

When Microsoft introduced Kernel Patch Protection, security vendors were significantly

constrained in their ability to monitor the kernel. Given the limited number of kernel extension points provided by Microsoft, they were increasingly compelled to rely on asynchronous ETW events for after-the-fact visibility of kernel actions performed on behalf of malware.

Given this dependency, the public documentation of Windows kernel telemetry sources is unfortunately somewhat sparse.

Kernel ETW Events

There are currently **four types of ETW providers** that we need to consider.

Firstly, there are legacy and modern variants of "event provider":

- legacy (**mof**-based) event providers
- modern (**manifest**-based) event providers

And then there are legacy and modern variants of "trace provider":

- legacy Windows software trace preprocessor (**WPP**) trace providers
- modern **TraceLogging** trace providers

The "event" versus "trace" distinction is mostly semantic. Event providers are typically registered with the operating system ahead of time, and you can inspect the available telemetry metadata. These are typically used by system administrators for troubleshooting purposes and are often semi-documented. But when something goes really, *really* wrong there are (hidden) trace providers. These are typically used only by the original software authors for advanced troubleshooting and are undocumented.

In practice, each uses a slightly different format file to describe and register its events and this introduces minor differences in how the events are logged - and, more importantly, how the potential events can be enumerated.

Modern Kernel Event Providers

The modern kernel ETW providers aren't strictly documented. However, registered event details can be queried from the operating system via the **Trace Data Helper API**. Microsoft's

PerfView tool uses these APIs to reconstruct the provider's registration manifest, and Pavel Yosifovich's EtwExplorer then wraps these manifests in a simple GUI. You can use these tab-separated value files of registered manifests from successive Windows versions. A single line per event is very useful for grepping, though others have since published the raw XML manifests.

These aren't all of the possible Windows ETW events, however. They are only the ones registered with the operating system by default. For example, the ETW events for many server roles aren't registered until that feature is enabled.

Legacy Kernel Event Providers

The legacy kernel events are documented by Microsoft. Mostly.

Legacy providers also exist within the operating system as WMI EventTrace classes. Providers are the root classes, groups are the children, and events are the grandchildren.

To search the legacy events in the same way as modern eventTo search legacy events in the same way as modern events, these classes were parsed, and the original MOF (mostly) reconstructed. This MOF support was added to EtwExplorer, and tab-separated value summaries of the legacy events were these classes were parsed and the original MOF (mostly) reconstructed. This MOF support was added to EtwExplorer and tab-separated value summaries of the legacy events published.

The fully reconstructed Windows Kernel Trace MOF is here (or in a tabular format here).

Of the 340 registered legacy events, only 116 were documented. Typically, each legacy event needs to be enabled via a specific flag, but these weren't documented either. There was a clue in the documentation for the kernel Object Manager Trace events. It mentioned `PERF_OB_HANDLE`, a constant that is not defined in the headers in the latest SDK. Luckily, Geoff Chappell and the Windows 10 1511 WDK came to the rescue. This information was used to add support for `PERFINFO_GROUPMASK` kernel trace flags to Microsoft's KrabsETW library. It also turned out that the Object Trace documentation was wrong. That non-public constant can only be used with an undocumented API extension. Fortunately, public Microsoft projects such as `PerfView` often provide examples of how to use undocumented APIs.

With both manifests and MOFs published on GitHub, most kernel events can now be found

with **this query**.

Interestingly, Microsoft often **obfuscates** the names of security-relevant events, so searching for events with a generic name prefix such as `task_` yields some **interesting results**.

Sometimes the keyword hints to the event's purpose. For example, `task_014` in `Microsoft-Windows-Kernel-General` is enabled with the keyword `KERNEL_GENERAL_SECURITY_ACCESSCHECK.`

And thankfully, the parameters are almost always well-named. We might guess that `task_05` in `Microsoft-Windows-Kernel-Audit-API-Calls` is related to **OpenProcess** since it logs fields named `TargetProcessId` and `DesiredAccess`.

Another useful query is to search for events with an explicit `ProcessStartKey` field. ETW events can be **configured** to include this field for the logging process, and any event that includes this information for another process is often security relevant.

If you had a specific API in mind, you might query for its name or its parameters. For example, if you want Named Pipe events, you might use **this query**.

In this instance, though, `Microsoft-Windows-SEC` belongs to the built-in Microsoft Security drivers that Microsoft Defender for Endpoint (MDE) utilizes. This provider is only officially available to MDE, though **Sebastian Feldmann and Philipp Schmied** have demonstrated how to start a session using an **AutoLogger** and subscribe to that session's events. This is only currently useful for MDE users as otherwise, the driver is not configured to emit events.

But what about trace providers?

Modern Kernel Trace Providers

TraceLogging metadata is stored as an opaque blob within the logging binary. Thankfully this format has been reversed by **Matt Graeber**. We can use Matt's script to dump all TraceLogging metadata for `ntoskrnl.exe`. A sample dump of Windows 11 TraceLogging metadata is **here**.

Unfortunately, the metadata structure alone doesn't retain the correlation between providers and events. There are interesting provider names, such as `Microsoft.Windows.Kernel.Security` and `AttackSurfaceMonitor`, but it's not yet clear from our metadata dump which events belong to these providers.

Legacy Kernel Trace Providers

WPP metadata is stored within symbols files (PDBs). Microsoft includes this information in the **public symbols for some, but not all, drivers**. The kernel itself, however, does not produce any WPP events. Instead, the legacy Windows Kernel Trace event provider can be passed undocumented flags to enable the legacy "trace" events usually only available to Microsoft kernel developers.

Provider	Documentation	Event Metadata
Modern Event Providers	None	Registered XML manifests
Legacy Event Providers	Partial	EventTrace WMI objects
Modern Trace Providers	None	Undocumented blob in binary
Legacy Trace Providers	None	Undocumented blob in Symbols

Next Steps

We now have kernel event metadata for each of the four flavours of ETW provider, but a list of ETW events is just our starting point. Knowing the provider and event keyword may not be enough to generate the events we expect. Sometimes, an additional configuration registry key or API call is required. More often, though, we just need to understand the exact conditions under which the event is logged.

Knowing exactly where and what is being logged is critical to truly understanding your telemetry and its limitations. And, thanks to decompilers becoming readily available, we have the option of some just-enough-reversing available to us. In IDA we call this "press F5". Ghidra is the open-source alternative and it supports scripting … with Java.

For kernel ETW, we are particularly interested in `EtwWrite` calls that are reachable from system calls. We want as much of the call site parameter information as possible, including any associated public symbol information. This meant that we needed to walk the call graph but also attempt to resolve the possible values for particular parameters.

The necessary parameters were the `RegHandle` and the `EventDescriptor`. The former is an opaque handle for the provider, and the latter provides event-specific information, such as the event id and its associated keywords. An ETW keyword is an identifier used to enable a set of events.

Even better, these event descriptors were typically stored in a global constant with a public symbol.

We had sufficient event metadata but still needed to resolve the opaque provider handle assigned at runtime back to the metadata about the provider. For this, we also needed the `EtwRegister` calls.

The typical pattern for kernel modern event providers was to store the constant provider GUID and the runtime handle in globals with public symbols.

Another pattern encountered was calls to `EtwRegister`, `EtwEwrite`, and `EtwUnregister`, all in the same function. In this case, we took advantage of the locality to find the provider GUID for the event.

Modern TraceLogging providers, however, did not have associated per-provider public symbols to provide a hint of each provider's purpose. However, Matt Graeber had **reversed the TraceLogging metadata** format and documented that the provider name is stored at a **fixed offset** from the provider GUID. Having the exact provider name is even better than just the public symbol we recovered for modern events.

This just left the legacy providers. They didn't seem to have either public symbols or metadata blobs. Some constants are passed to an undocumented function named `EtwTraceKernelEvent` which wraps the eventual ETW write call.

Those constants are present in the Windows 10 1511 WDK headers (and the **System Informer** headers), so we could label these events with the constant names.

This script has been recently updated for Ghidra 11, along with improved support for TraceLogging and Legacy events. You can now find it on GitHub here - **https://github.com/jdu2600/API-To-ETW**

Sample output for the Windows 11 kernel is **here**.

Our previously anonymous `Microsoft-Windows-Kernel-Audit-API-Calls` events are

quickly unmasked by this script.

Id	EVENT_DESCRIPTOR Symbol	Function
1	KERNEL_AUDIT_API_PSSETLOADIMAGENOTIFYROUTINE	PsSetLoadImageNotifyRoutineEx
2	KERNEL_AUDIT_API_TERMINATEPROCESS	NtTerminateProcess
3	KERNEL_AUDIT_API_CREATESYMBOLICLINKOBJECT	ObCreateSymbolicLink
4	KERNEL_AUDIT_API_SETCONTEXTTHREAD	NtSetContextThread
5	KERNEL_AUDIT_API_OPENPROCESS	PsOpenProcess
6	KERNEL_AUDIT_API_OPENTHREAD	PsOpenThread
7	KERNEL_AUDIT_API_IOREGISTERLASTCHANCESHUTDOWNNOTIFICATION	IoRegisterLastChanceShutdownNotification
8	KERNEL_AUDIT_API_IOREGISTERSHUTDOWNNOTIFICATION	IoRegisterShutdownNotification

Symbol and containing function for Microsoft-Windows-Kernel-Audit-API-Calls events

With the call path and parameter information recovered by the script, we can also see that the `SECURITY_ACCESSCHECK` event from earlier is associated with the **SeAccessCheck** kernel API, but only logged within a function named `SeLogAccessFailure`. Only logging failure conditions is a very common occurrence with ETW events. For troubleshooting purposes, the original ETW use case, these are typically the most useful and the implementation in most components reflects this. Unfortunately, for security purposes, the inverse is often true. The successful operation logs are usually more useful for finding malicious activity. So, the value of some of these legacy events is often low.

Modern **Secure by Design** practice is to audit log both success and failure for security

relevant activities and Microsoft continues to add new security-relevant ETW events that do this. For example, the preview build of Windows 11 24H2 includes some **interesting new ETW events** in the `Microsoft-Windows-Threat-Intelligence` provider. Hopefully, these will be documented for security vendors ahead of its release.

Running this decompiler script across interesting Windows drivers and service DLLs is left as an exercise to the reader.