

ETW internals for security research and forensics

 blog.trailofbits.com/2023/11/22/etw-internals-for-security-research-and-forensics/

By Yarden Shafir

Why has Event Tracing for Windows (ETW) become so pivotal for endpoint detection and response (EDR) solutions in Windows 10 and 11? The answer lies in the value of the intelligence it provides to security tools through secure ETW channels, which are now also a target for offensive researchers looking to bypass detections.

In this deep dive, we're not just discussing ETW's functionalities; we're exploring how ETW works internally so you can conduct novel research or forensic analysis on a system. Security researchers and malware authors already target ETW. They have developed several techniques to tamper with or bypass ETW-based EDRs, hook system calls, or gain access to ETW providers normally reserved for anti-malware solutions. Most recently, the Lazarus Group bypassed EDR detection by disabling ETW providers. Here, we'll explain how ETW works and what makes it such a tempting target, and we'll embark on an exciting journey deep into Windows.

Overview of ETW internals

Two main components of ETW are providers and consumers. Providers send events to an ETW globally unique identifier (GUID), and the events are written to a file, a buffer in memory, or both. Every Windows system has hundreds or thousands of providers registered. We can view available providers by running the command `logman query providers`:

```
Administrator: Command Prompt
C:\Windows\System32>logman query providers

Provider                                         GUID
-----
ACPI Driver Trace Provider                     {DAB01D4D-2D48-477D-B1C3-DAAD0CE6F06B}
Active Directory Domain Services: SAM          {8E598056-8993-11D2-819E-0000F875A064}
Active Directory: Kerberos Client              {BBA3ADD2-C229-4CDB-AE2B-57EB6966B0C4}
Active Directory: NetLogon                     {F33959B4-DBEC-11D2-895B-00C04F79AB69}
ADODB.1                                        {04C8A86F-3369-12F8-4769-24E484A9E725}
ADOMD.1                                        {7EA56435-3F2F-3F63-A829-F0B35B5CAD41}
Application Error                             {A0E9B465-B939-57D7-B27D-95D8E925FF57}
Application Hang                             {C631C3DC-C676-59E4-2DB3-5C0AF00F9675}
Application Popup                             {47BFA2B7-BD54-4FAC-B70B-29021084CA8F}
Application-Addon-Event-Provider              {A83FA99F-C356-4DED-9FD6-5A5EB8546D68}
ATA Port Driver Tracing Provider              {D08BD885-501E-489A-BAC6-B7D24BFE68BF}
AuthFw NetShell Plugin                        {935F4AE6-845D-41C6-97FA-380DAD429B72}
BCP.1                                         {24722B88-DF97-4FF6-E395-DB533AC42A1E}
BFE Trace Provider                           {106B464A-8043-46B1-8CB8-E92A0CD7A560}
BITS Service Trace                           {4A8AAA94-CFC4-46A7-8E4E-17BC45608F0A}
Certificate Services Client CredentialRoaming Trace {EF4109DC-68FC-45AF-B329-CA2825437209}
Certificate Services Client Trace             {F01B7774-7ED7-401E-8088-B576793D7841}
Circular Kernel Session Provider              {54DEA73A-ED1F-42A4-AF71-3E63D056F174}
Classpnp Driver Tracing Provider              {FA8DE7C4-ACDE-4443-9994-C4E2359A9EDB}
Critical Section Trace Provider               {3AC66736-CC59-4CFF-8115-8DF50E39816B}
CrowdStrike-Falcon Sensor-CSFalconService    {07A88C90-6EDA-4F36-0A2F-70D7006E5482}
DBNETLIB.1                                   {BD568F20-FCCD-B948-054E-DB3421115D61}
Deduplication Tracing Provider                {5EBB59D1-4739-4E45-872D-B8703956D84B}
Disk Class Driver Tracing Provider            {945186BF-3DD6-4F3F-9C8E-9EDD3FC9D558}
Downlevel IPsec API                           {94335EB3-79EA-44D5-8EA9-306F49B3A041}
Downlevel IPsec NetShell Plugin              {E4FF10D8-8A88-4FC6-82C8-8C23E9462FE5}
```

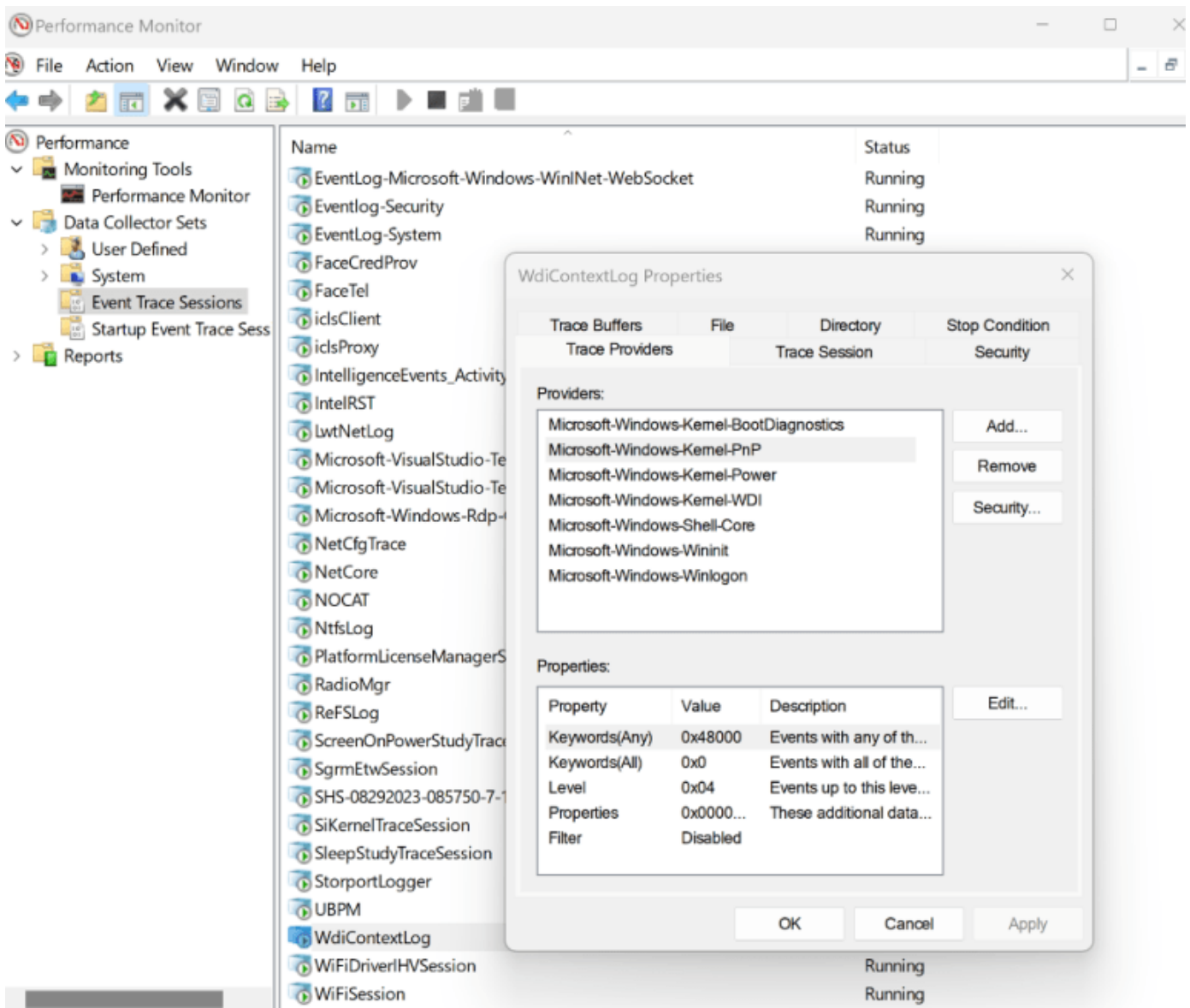
By checking my system, we can see there are nearly 1,200 registered providers:

```
C:\Windows\System32>logman query providers | find /c /v ""
1197
```

Each of these ETW providers defines its own events in a manifest file, which is used by consumers to parse provider-generated data. ETW providers may define hundreds of different event types, so the amount of information we can get from ETW is enormous. Most of these events can be seen in Event Viewer, a built-in Windows tool that consumes ETW events. But you'll only see some of the data. Not all logs are enabled by default in Event Viewer, and not all event IDs are shown for each log.

On the other side we have consumers: trace logging sessions that receive events from one or several providers. For example, EDRs that rely on ETW data for their detection will consume events from security-related ETW channels such as the Threat Intelligence channel.

We can look at all running ETW consumers via Performance Monitor; clicking one of the sessions will show the providers it subscribes to. (You may need to run as SYSTEM to see all ETW logging sessions.)



The list of processes that receive events from this log session is useful information but not easy to obtain. As far as I could see there is no way to get that information from user mode at all, and even from kernel mode it's not an easy task unless you are very familiar with ETW internals. So we will see what we can learn from a kernel debugging session using WinDbg.

Finding ETW consumer processes

There are ways to find consumers of ETW log sessions from user mode. However, they only supply very partial information that isn't enough in all cases. So instead, we'll head to our kernel debugger session. One way to get information about ETW sessions from the debugger is using the built-in extension `!wmitrace`. This extremely useful extension allows users to investigate all of the running loggers and their attributes, consumers, and buffers. It even allows users to start and stop log sessions (on a live debugger connection). Still, like all legacy extensions, it has its limitations: it can't be easily automated, and since it's a precompiled binary it can't be extended with new functionality.

So instead we'll write a JavaScript script—scripts are easier to extend and modify, and we can use them to get as much data as we need without being limited to the preexisting functionality of a legacy extension.

Every handle contains a pointer to an object. For example, a file handle will point to a kernel structure of type `FILE_OBJECT`. A handle to an object of type `EtwConsumer` will point to an undocumented data structure called `ETW_REALTIME_CONSUMER`. This structure contains a pointer to the process that opened it, events that get notified for different actions, flags, and also one piece of information that will (eventually) lead us back to the log session—`LoggerId`. Using a custom script, we can scan the handle tables of all processes for handles to `EtwConsumer` objects. For each one, we can get the linked `ETW_REALTIME_CONSUMER` structure and print the `LoggerId`:

```
"use strict";

function initializeScript()
{
    return [new host.apiVersionSupport(1, 7)];
}

function EtwConsumersForProcess(process)
{
    let dbgOutput = host.diagnostics.debugLog;
    let handles = process.Io.Handles;

    try
    {
        for (let handle of handles)
        {
            try
            {
                let objType = handle.Object.ObjectType;
                if (objType === "EtwConsumer")
                {
                    let consumer = host.createTypedObject(handle.Object.Body.address, "nt",
"_ETW_REALTIME_CONSUMER");
                    let loggerId = consumer.LoggerId;

                    dbgOutput("Process ", process.Name, " with ID ", process.Id, " has handle ",
handle.Handle, " to Logger ID ", loggerId, "\n");
                }
            } catch (e) {
                dbgOutput("\tException parsing handle ", handle.Handle, "in process ", process.Name,
"! \n");
            }
        }
    } catch (e) {
    }
}
```

Next, we load the script into the debugger with `.scriptload` and call our function to identify which process consumes ETW events:

```
dx @$cursession.Processes.Select(p => @$scriptContents.EtwConsumersForProcess(p))
@$cursession.Processes.Select(p => @$scriptContents.EtwConsumersForProcess(p))
Process svchost.exe with ID 0x558 has handle 0x7cc to Logger ID 31
Process svchost.exe with ID 0x114c has handle 0x40c to Logger ID 36
Process svchost.exe with ID 0x11f8 has handle 0x2d8 to Logger ID 17
Process svchost.exe with ID 0x11f8 has handle 0x2e8 to Logger ID 3
Process svchost.exe with ID 0x11f8 has handle 0x2f4 to Logger ID 9
Process NVDisplay.Container.exe with ID 0x1478 has handle 0x890 to Logger ID 38
Process svchost.exe with ID 0x1cec has handle 0x1dc to Logger ID 7
Process svchost.exe with ID 0x1d2c has handle 0x780 to Logger ID 8
Process CSFalconService.exe with ID 0x1e54 has handle 0x760 to Logger ID 3
Process CSFalconService.exe with ID 0x1e54 has handle 0x79c to Logger ID 45
Process CSFalconService.exe with ID 0x1e54 has handle 0xbb0 to Logger ID 10
Process Dell.TechHub.Instrumentation.SubAgent.exe with ID 0x25c4 has handle 0xcd8 to Logger ID 41
Process Dell.TechHub.Instrumentation.SubAgent.exe with ID 0x25c4 has handle 0xdb8 to Logger ID 35
Process Dell.TechHub.Instrumentation.SubAgent.exe with ID 0x25c4 has handle 0xf54 to Logger ID 44
Process SgrmBroker.exe with ID 0x17b8 has handle 0x178 to Logger ID 15
Process SystemInformer.exe with ID 0x4304 has handle 0x30c to Logger ID 16
Process PerfWatson2.exe with ID 0xa60 has handle 0xa3c to Logger ID 46
Process PerfWatson2.exe with ID 0x81a4 has handle 0x9c4 to Logger ID 40
Process PerfWatson2.exe with ID 0x76f0 has handle 0x9a8 to Logger ID 47
Process operfmon.exe with ID 0x3388 has handle 0x88c to Logger ID 48
Process operfmon.exe with ID 0x3388 has handle 0x8f4 to Logger ID 49
```

While we still don't get the name of the log sessions, we already have more data than we did in user mode. We can see, for example, that some processes have multiple consumer handles since they are subscribed to multiple log sessions. Unfortunately, the `ETW_REALTIME_CONSUMER` structure doesn't have any information about the log session besides its identifier, so we must find a way to match identifiers to human-readable names.

The registered loggers and their IDs are stored in a global list of loggers (or at least they were until the introduction of server silos; now, every isolated process will have its own separate ETW loggers while non-isolated processes will use the global list, which I will also use in this post). The global list is stored inside an `ETW_SILODRIVERSTATE` structure within the host silo globals, `nt!PspHostSiloGlobals`:

```

dx ((nt!_ESERVERSILO_GLOBALS*)&nt!PspHostSiloGlobals)->EtwSiloState
((nt!_ESERVERSILO_GLOBALS*)&nt!PspHostSiloGlobals)->EtwSiloState :
0xfffffe38f3deeb000 [Type: _ETW_SILODRIVERSTATE *]
[+0x000] Silo : 0x0 [Type: _EJOB *]
[+0x008] SiloGlobals : 0xfffff8052bd489c0 [Type: _ESERVERSILO_GLOBALS *]
[+0x010] MaxLoggers : 0x50 [Type: unsigned long]
[+0x018] EtwpSecurityProviderGuidEntry [Type: _ETW_GUID_ENTRY]
[+0x1c0] EtwpLoggerRundown : 0xfffffe38f3deca040 [Type: _EX_RUNDOWN_REF_CACHE_AWARE * *]
[+0x1c8] EtwpLoggerContext : 0xfffffe38f3deca2c0 [Type: _WMI_LOGGER_CONTEXT * *]
[+0x1d0] EtwpGuidHashTable [Type: _ETW_HASH_BUCKET [64]]
[+0xfd0] EtwpSecurityLoggers [Type: unsigned short [8]]
[+0xfe0] EtwpSecurityProviderEnableMask : 0x3 [Type: unsigned char]
[+0xfe4] EtwpShutdownInProgress : 0 [Type: long]
[+0xfe8] EtwpSecurityProviderPID : 0x798 [Type: unsigned long]
[+0xff0] PrivHandleDemuxTable [Type: _ETW_PRIV_HANDLE_DEMUX_TABLE]
[+0x1010] RTBacklogFileRoot : 0x0 [Type: wchar_t *]
[+0x1018] EtwpCounters [Type: _ETW_COUNTERS]
[+0x1028] LogfileBytesWritten : {4391651513} [Type: _LARGE_INTEGER]
[+0x1030] ProcessorBlocks : 0x0 [Type: _ETW_SILO_TRACING_BLOCK *]
[+0x1038] ContainerStateWnfSubscription : 0xfffffaf8de0386130 [Type: _EX_WNF_SUBSCRIPTION *]
[+0x1040] ContainerStateWnfCallbackCalled : 0x0 [Type: unsigned long]
[+0x1048] UnsubscribeworkItem : 0xfffffaf8de0202170 [Type: _WORK_QUEUE_ITEM *]
[+0x1050] PartitionId : {00000000-0000-0000-0000-000000000000} [Type: _GUID]
[+0x1060] ParentId : {00000000-0000-0000-0000-000000000000} [Type: _GUID]
[+0x1070] QpcOffsetFromRoot : {0} [Type: _LARGE_INTEGER]
[+0x1078] PartitionName : 0x0 [Type: char *]
[+0x1080] PartitionNameSize : 0x0 [Type: unsigned short]
[+0x1082] UnusedPadding : 0x0 [Type: unsigned short]
[+0x1084] PartitionType : 0x0 [Type: unsigned long]
[+0x1088] SystemLoggerSettings [Type: _ETW_SYSTEM_LOGGER_SETTINGS]
[+0x1200] EtwpStartTraceMutex [Type: _KMUTANT]

```

The **EtwpLoggerContext** field points to an array of pointers to **WMI_LOGGER_CONTEXT** structures, each describing one logger session. The size of the array is saved in the **MaxLoggers** field of the **ETW_SILODRIVERSTATE**. Not all entries of the array are necessarily used; unused entries will be set to 1. Knowing this, we can dump all of the initialized entries of the array. (I've hard coded the array size for convenience):


```

dx ((nt!_WMI_LOGGER_CONTEXT*)(*)[0x50])(((nt!_ESERVERSILO_GLOBALS*)&nt!PspHostSiloGlobals)-
>EtwSiloState->EtwpLoggerContext))->Where(1 => 1 != 1)
((nt!_WMI_LOGGER_CONTEXT*)(*)[0x50])(((nt!_ESERVERSILO_GLOBALS*)&nt!PspHostSiloGlobals)-
>EtwSiloState->EtwpLoggerContext))->Where(1 => 1 != 1)
[2]          : 0xfffffe38f3f0c9040 [Type: _WMI_LOGGER_CONTEXT *]
[3]          : 0xfffffe38f3fe07640 [Type: _WMI_LOGGER_CONTEXT *]
[4]          : 0xfffffe38f3f0c75c0 [Type: _WMI_LOGGER_CONTEXT *]
[5]          : 0xfffffe38f3f0c9780 [Type: _WMI_LOGGER_CONTEXT *]
[6]          : 0xfffffe38f3f0cb040 [Type: _WMI_LOGGER_CONTEXT *]
[7]          : 0xfffffe38f3f0cb600 [Type: _WMI_LOGGER_CONTEXT *]
[8]          : 0xfffffe38f3f0ce040 [Type: _WMI_LOGGER_CONTEXT *]
[9]          : 0xfffffe38f3f0ce600 [Type: _WMI_LOGGER_CONTEXT *]
[10]         : 0xfffffe38f79832a40 [Type: _WMI_LOGGER_CONTEXT *]
[11]         : 0xfffffe38f3f0d1640 [Type: _WMI_LOGGER_CONTEXT *]
[12]         : 0xfffffe38f89535a00 [Type: _WMI_LOGGER_CONTEXT *]
[13]         : 0xfffffe38f3dacc940 [Type: _WMI_LOGGER_CONTEXT *]
[14]         : 0xfffffe38f3fe04040 [Type: _WMI_LOGGER_CONTEXT *]
...

```

Each logger context contains information about the logger session such as its name, the file that stores the events, the security descriptor, and more. Each structure also contains a logger ID, which matches the index of the logger in the array we just dumped. So given a logger ID, we can find its details like this:

```

dx (((nt!_ESERVERSILO_GLOBALS*)&nt!PspHostSiloGlobals)->EtwSiloState->EtwpLoggerContext)[@$loggerId]
(((nt!_ESERVERSILO_GLOBALS*)&nt!PspHostSiloGlobals)->EtwSiloState->EtwpLoggerContext)[@$loggerId]
: 0xfffffe38f3f0ce600 [Type: _WMI_LOGGER_CONTEXT *]
[+0x000] LoggerId      : 0x9 [Type: unsigned long]
[+0x004] BufferSize     : 0x10000 [Type: unsigned long]
[+0x008] MaximumEventSize : 0xffb8 [Type: unsigned long]
[+0x00c] LoggerMode     : 0x19800180 [Type: unsigned long]
[+0x010] AcceptNewEvents : 0 [Type: long]
[+0x018] GetCpuClock    : 0x0 [Type: unsigned __int64]
[+0x020] LoggerThread   : 0xfffffe38f3f0d0040 [Type: _ETHREAD *]
[+0x028] LoggerStatus   : 0 [Type: long]
...

```

Now we can implement this as a function (in DX or JavaScript) and print the logger name for each open consumer handle we find:

```

dx @$cursession.Processes.Select(p => @$scriptContents.EtwConsumersForProcess(p))
@$cursession.Processes.Select(p => @$scriptContents.EtwConsumersForProcess(p))
Process svchost.exe with ID 0x558 has handle 0x7cc to Logger ID 31
    Logger Name: "UBPM"

Process svchost.exe with ID 0x114c has handle 0x40c to Logger ID 36
    Logger Name: "WFP-IPsec Diagnostics"

Process svchost.exe with ID 0x11f8 has handle 0x2d8 to Logger ID 17
    Logger Name: "EventLog-System"

Process svchost.exe with ID 0x11f8 has handle 0x2e8 to Logger ID 3
    Logger Name: "Eventlog-Security"

Process svchost.exe with ID 0x11f8 has handle 0x2f4 to Logger ID 9
    Logger Name: "EventLog-Application"

Process NVDisplay.Container.exe with ID 0x1478 has handle 0x890 to Logger ID 38
    Logger Name: "NOCAT"

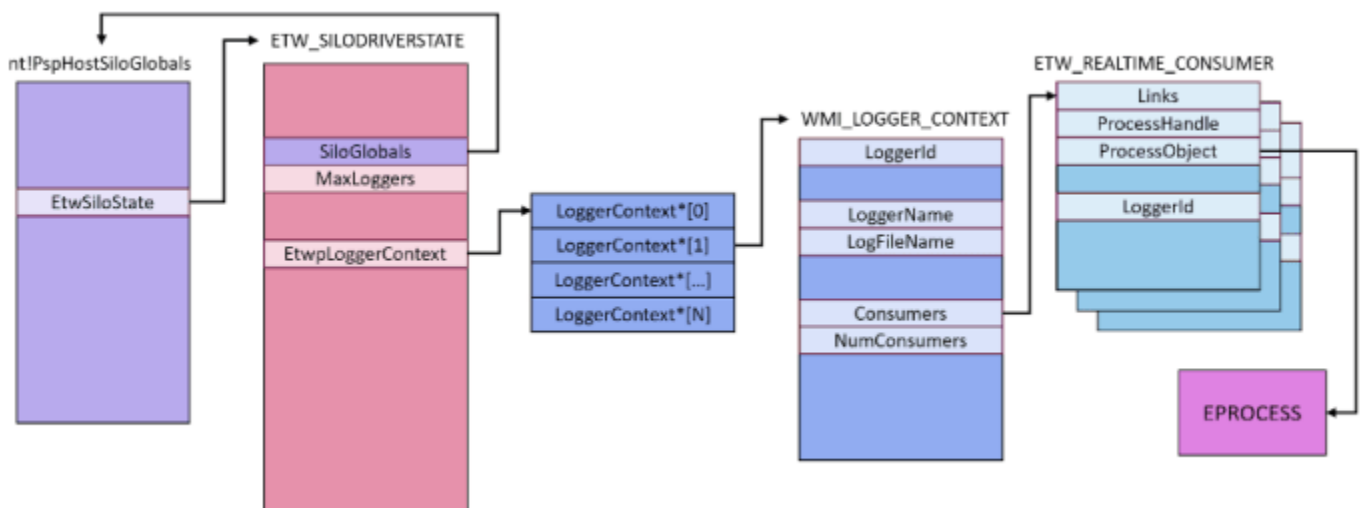
Process svchost.exe with ID 0x1cec has handle 0x1dc to Logger ID 7
    Logger Name: "DiagLog"

Process svchost.exe with ID 0x1d2c has handle 0x780 to Logger ID 8
    Logger Name: "Diagtrack-Listener"

Process CSFalconService.exe with ID 0x1e54 has handle 0x760 to Logger ID 3
    Logger Name: "Eventlog-Security"
...

```

In fact, by using the logger array, we can build a better way to enumerate ETW log session consumers. Each logger context has a **Consumers** field, which is a linked list connecting all of the **ETW_REALTIME_CONSUMER** structures that are subscribed to this log session:



So instead of scanning the handle table of each and every process in the system, we can go directly to the loggers array and find the registered processes for each one:

```
function EtwLoggersWithConsumerProcesses()
{
    let dbgOutput = host.diagnostics.debugLog;
    let hostSiloGlobals = host.getModuleSymbolAddress("nt", "PspHostSiloGlobals");
    let typedhostSiloGlobals = host.createTypedObject(hostSiloGlobals, "nt",
        "_ESERVERSILO_GLOBALS");

    let maxLoggers = typedhostSiloGlobals.EtwSiloState.MaxLoggers;
    for (let i = 0; i < maxLoggers; i++)
    {
        let logger = typedhostSiloGlobals.EtwSiloState.EtwpLoggerContext[i];
        if (host.parseInt64(logger.address, 16).compareTo(host.parseInt64("0x1")) != 0)
        {
            dbgOutput("Logger Name: ", logger.LoggerName, "\n");

            let consumers =
host.namespace.Debugger.Utility.Collections.FromListEntry(logger.Consumers,
"nt!_ETW_REALTIME_CONSUMER", "Links");
            if (consumers.Count() != 0)
            {
                for (let consumer of consumers)
                {
                    dbgOutput("\tProcess Name: ",
consumer.ProcessObject.SeAuditProcessCreationInfo.ImageFileName.Name, "\n");
                    dbgOutput("\tProcess Id: ",
host.parseInt64(consumer.ProcessObject.UniqueProcessId.address, 16).toString(10), "\n");
                    dbgOutput("\n");
                }
            }
            else
            {
                dbgOutput("\tThis logger has no consumers\n\n");
            }
        }
    }
}
```

Calling this function should get us the exact same results as earlier, only much faster!

After getting this part, we can continue to search for another piece of information that could be useful—the list of GUIDs that provide events to a log session.

Finding provider GUIDs

Finding the consumers of an ETW log session is only half the battle—we also want to know which providers notify each log session. We saw earlier that we can get that information from Performance Monitor, but let's see how we can also get it from a debugger session, as it might be useful when the live machine isn't available or when looking for details that aren't supplied by user-mode tools like Performance Monitor.

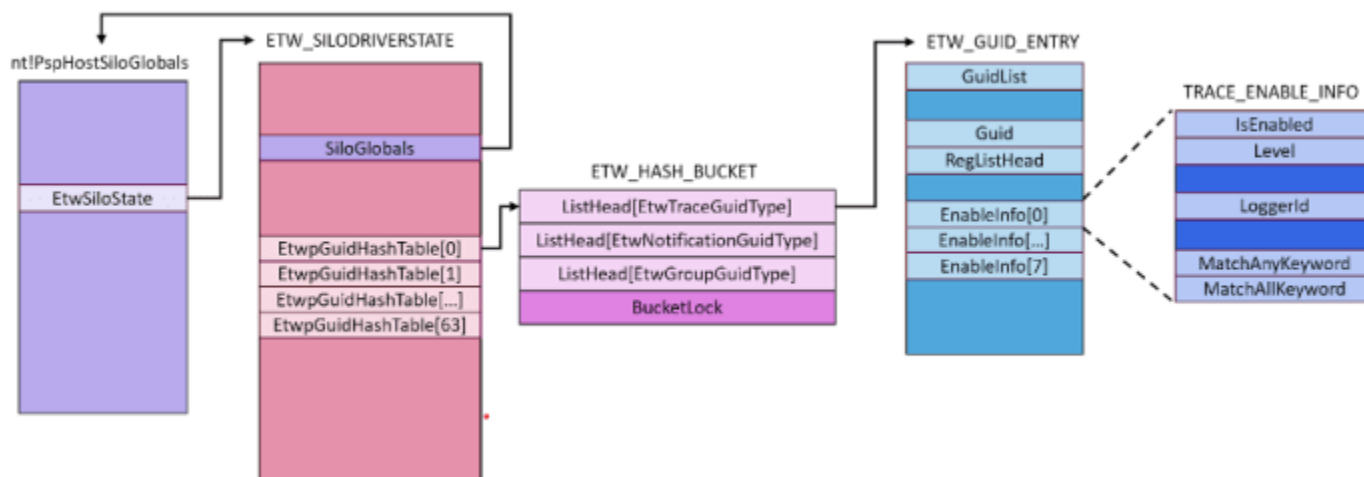
If we look at the `WMI_LOGGER_CONTEXT` structure, we won't see any details about the providers that notify the log session. To find this information, we need to go back to the `ETW_SILODRIVERSTATE` structure from earlier and look at the `EtwpGuidHashTable` field. This is an array of buckets storing all of the registered provider GUIDs. For performance reasons, the GUIDs are hashed and stored in 64 buckets. Each bucket contains three lists linking `ETW_GUID_ENTRY` structures. There is one list for each `ETW_GUID_TYPE`:

- `EtwpTraceGuidType`
- `EtwpNotificationGuidType`
- `EtwpGroupGuidType`

Each `ETW_GUID_ENTRY` structure contains an `EnableInfo` array with eight entries, and each contains information about one log session that the GUID is providing events for (which means that an event GUID entry can supply events for up to eight different log sessions):

```
dt nt!_ETW_GUID_ENTRY EnableInfo.
+0x080 EnableInfo : [8]
+0x000 IsEnabled   : Uint4B
+0x004 Level       : UChar
+0x005 Reserved1   : UChar
+0x006 LoggerId    : Uint2B
+0x008 EnableProperty : Uint4B
+0x00c Reserved2   : Uint4B
+0x010 MatchAnyKeyword : Uint8B
+0x018 MatchAllKeyword : Uint8B
```

Visually, this is what this whole thing looks like:



As we can see, the `ETW_GUID_ENTRY` structure contains a `LoggerId` field, which we can use as the index into the `EtwpLoggerContext` array to find the log session.

With this new information in mind, we can write a simple JavaScript function to print the GUIDs that match a logger ID. (In this case, I chose to go over only one `ETW_GUID_TYPE` at a time to make this code a bit cleaner.) Then we can go one step further and parse the `ETW_REG_ENTRY` list in each GUID entry to find out which processes notify it, or if it's a kernel-mode provider:

```

function GetGuidsForLoggerId(loggerId, guidType)
{
    let dbgOutput = host.diagnostics.debugLog;

    let hostSiloGlobals = host.getModuleSymbolAddress("nt", "PspHostSiloGlobals");
    let typedhostSiloGlobals = host.createTypedObject(hostSiloGlobals, "nt",
"_ESERVERSILO_GLOBALS");
    let guidHashTable = typedhostSiloGlobals.EtwSiloState.EtwpGuidHashTable;
    for (let bucket of guidHashTable)
    {
        let guidEntries =
host.namespace.Debugger.Utility.Collections.FromListEntry(bucket.ListHead[guidType],
"nt!_ETW_GUID_ENTRY", "GuidList");
        if (guidEntries.Count() != 0)
        {
            for (let guid of guidEntries)
            {
                for (let enableInfo of guid.EnableInfo)
                {
                    if (enableInfo.LoggerId === loggerId)
                    {
                        dbgOutput("\tGuid: ", guid.Guid, "\n");
                        let regEntryLinkField = "RegList";
                        if (guidType == 2)
                        {
                            // group GUIDs registration entries are linked through the GroupRegList
field
                            regEntryLinkField = "GroupRegList";
                        }
                        let regEntries =
host.namespace.Debugger.Utility.Collections.FromListEntry(guid.RegListHead, "nt!_ETW_REG_ENTRY",
regEntryLinkField);
                        if (regEntries.Count() != 0)
                        {
                            dbgOutput("\tProvider Processes:\n");
                            for (let regEntry of regEntries)
                            {
                                if (regEntry.DbgUserRegistration != 0)
                                {
                                    dbgOutput("\t\tProcess: ",
regEntry.Process.SeAuditProcessCreationInfo.ImageFileName.Name, " ID: ",
host.parseInt64(regEntry.Process.UniqueProcessId.address, 16).toString(10), "\n");
                                }
                                else
                                {
                                    dbgOutput("\t\tKernel Provider\n");
                                }
                            }
                        }
                        break;
                    }
                }
            }
        }
    }
}

```

```
}  
  }  
    }  
      }  
        }
```

As an example, here are all of the trace provider GUIDs and the processes that notify them for ETW session UBPM (**LoggerId** 31 in my case):

```

dx @$scriptContents.GetGuidsForLoggerId(31, 0)
  Guid: {9E03F75A-BCBE-428A-8F3C-D46F2A444935}
  Provider Processes:
    Process: "\\Device\\HarddiskVolume3\\Windows\\System32\\svchost.exe" ID: 2816
  Guid: {2D7904D8-5C90-4209-BA6A-4C08F409934C}
  Guid: {E46EEAD8-0C54-4489-9898-8FA79D059E0E}
  Provider Processes:
    Process: "\\Device\\HarddiskVolume3\\Windows\\System32\\dwm.exe" ID: 2268
  Guid: {D02A9C27-79B8-40D6-9B97-CF3F8B7B5D60}
  Guid: {92AAB24D-D9A9-4A60-9F94-201FED3E3E88}
  Provider Processes:
    Process: "\\Device\\HarddiskVolume3\\Windows\\System32\\svchost.exe" ID: 2100
    Kernel Provider
  Guid: {FBCFAC3F-8460-419F-8E48-1F0B49CDB85E}
  Guid: {199FE037-2B82-40A9-82AC-E1D46C792B99}
  Provider Processes:
    Process: "\\Device\\HarddiskVolume3\\Windows\\System32\\lsass.exe" ID: 1944
  Guid: {BD2F4252-5E1E-49FC-9A30-F3978AD89EE2}
  Provider Processes:
    Process: "\\Device\\HarddiskVolume3\\Windows\\System32\\svchost.exe" ID: 16292
  Guid: {22B6D684-FA63-4578-87C9-EFFCBE6643C7}
  Guid: {3635D4B6-77E3-4375-8124-D545B7149337}
  Guid: {0621B9DF-3249-4559-9889-21F76B5C80F3}
  Guid: {BD8FEA17-5549-4B49-AA03-1981D16396A9}
  Guid: {F5528ADA-BE5F-4F14-8AEF-A95DE7281161}
  Guid: {54732EE5-61CA-4727-9DA1-10BE5A4F773D}
  Provider Processes:
    Process: "\\Device\\HarddiskVolume3\\Windows\\System32\\svchost.exe" ID: 4428
  Guid: {18F4A5FD-FD3B-40A5-8FC2-E5D261C5D02E}
  Guid: {8E6A5303-A4CE-498F-AFDB-E03A8A82B077}
  Provider Processes:
    Kernel Provider
  Guid: {CE20D1C3-A247-4C41-BCB8-3C7F52C8B805}
  Provider Processes:
    Kernel Provider
  Guid: {5EF81E80-CA64-475B-B469-485DBC993FE2}
  Guid: {9B307223-4E4D-4BF5-9BE8-995CD8E7420B}
  Provider Processes:
    Kernel Provider
  Guid: {AA1F73E8-15FD-45D2-ABFD-E7F64F78EB11}
  Provider Processes:
    Kernel Provider
  Guid: {E1BDC95E-0F07-5469-8E64-061EA5BE6A0D}
  Guid: {5B004607-1087-4F16-B10E-979685A8D131}
  Guid: {AEDD909F-41C6-401A-9E41-DFC33006AF5D}
  Guid: {277C9237-51D8-5C1C-B089-F02C683E5BA7}
  Provider Processes:
    Kernel Provider
  Guid: {F230D19A-5D93-47D9-A83F-53829EDFB8DF}
  Provider Processes:
    Process: "\\Device\\HarddiskVolume3\\Windows\\System32\\svchost.exe" ID: 2816

```

Putting all of those steps together, we finally have a way to know which log sessions are running on the machine, which processes notify each of the GUIDs in the session, and which processes are subscribed to them. This can help us understand the purpose of different ETW log sessions running on the machine, such as identifying the log sessions used by EDR software or interesting hardware components. These scripts can also be modified as needed to identify ETW irregularities, such as a log session that has been disabled in order to blind security products. From an attacker perspective, gathering this information can tell us which ETW providers are used on a machine and which ones are ignored and, therefore, don't present us with any risk of detection.

Overall, ETW is a very powerful mechanism, so getting more visibility into its internal workings is useful for attackers and defenders alike. This post only scratches the surface, and there's so much more work that can be done in this area.

All of the JavaScript functions shown in this post can be found in this [GitHub repo](#).