Hooking Context Swaps with ETW

Archie ⋮⋮ 4/9/2025

Apr 9, 2025

Event Tracing for Windows (ETW) is a kernel mechanism designed to log certain activity happening in the system. Despite its seemingly innocuous description, ETW can be a valuable source of information and a very interesting hook point for both anti-cheats and other drivers.

Part 1: Finding the hook point

All ETW logging functions eventually end up inside `nt!EtwpLogKernelEvent` which, in summary, reserves a buffer for the log using `nt!EtwpReserveTraceBuffer` and then writes the log to that buffer.

Deep inside `nt!EtwpReserveTraceBuffer` is where the real fun begins. The function accesses a `_WMI_LOGGER_CONTEXT` structure - the kernel's representation of a logger - and looks at the `GetCpuClock` member before deciding on how to get the current time.

Anyone who's ever looked at how *InfinityHook* works will immediately recognize this member variable, as it was the hook point used by its creators. In the past, the variable was a function pointer that could be directly swapped to easily gain execution at each captured event. In an effort to patch *InfinityHook*, Microsoft turned the variable into an index, with each index representing a different way of getting time.

Looking at the relevant code inside `EtwpReserveTraceBuffer`, we can deduce what indices are valid, together with their meaning:

```
const auto get_cpu_clock = LoggerContext->GetCpuClock;
LARGE_INTEGER current_time = { .QuadPart = 0 };

// Crash the computer if the index is invalid.
if (get_cpu_clock > 3)
    KeBugCheck(KERNEL_SECURITY_CHECK_FAILURE);

switch (get_cpu_clock)
{
case 3:
    current_time.QuadPart = __rdtsc();
    break;
```

```
case 2:

HalPrivateDispatchTable.HalTimerQueryHostPerformanceCounter(&current_time);
    break;
case 1:
    current_time = KeQueryPerformanceCounter(nullptr);
    break;
case 0:
    current_time = RtlGetSystemTimePrecise();
    break;
default:
    KeBugCheck(KERNEL_SECURITY_CHECK_FAILURE);
}
```

For the purposes of this article, we'll focus on what happens when the value is set to `1`. At the beginning of the `nt!KeQueryPerformanceCounter` function, we can see the following snippet.

Edit: Thanks to @sixtyvividtails on X, it has come to my attention that `HalpPerformanceCounter` is actually a `hal!_REGISTERED_TIMER` structure.

```
LARGE_INTEGER result = { .QuadPart = 0 };

// This seems to always be true - the TimerProcessor constant (= 5) comes
from hal!_KNOWN_TIMER_TYPE.
if (HalpPerformanceCounter.KnownType == TimerProcessor)
{
    PVOID internal_data = HalpTimerGetInternalData(HalpPerformanceCounter);
    if (HalpTimerReferencePage)
    {
        result =
HalpPerformanceCounter.FunctionTable.QueryCounter(internal_data);
    }
    else
    {
        // ...
        result =
HalpPerformanceCounter.FunctionTable.QueryCounter(internal_data);
        // ...
    }
}
```

Simply swapping the pointer to `QueryCounter` is enough to get us a hook. There is just one problem - `nt!KeQueryPerformanceCounter` is a function that is called **very** often. It's also impossible to set a breakpoint inside, as any connected kernel debugger will hang upon the breakpoint being hit.

To prevent false positives (and get our debugger to work again), we need to figure out if calls made to our hook come from ETW. In the tested version of Windows 11 24H2, there is a pointer to the logger context in the `r15` register if the call comes from ETW. In other versions of Windows (mainly Windows 10), one may have to resort to scanning the stack for pointers to the logger context.

Part 2: Configuring the logger

Making ETW call our hook is not that simple - we will first need to access the `GetCpuClock` variable of the `_WMI_LOGGER_CONTEXT` structure to make the kernel call our hook. While it is possible to create a new logger and get a pointer to the structure that way, I chose to instead hijack the **Circular Kernel Context Logger** (CKCL), as it is usually not used for anything important. A pointer to its context can be retrieved quite easily, as there is a pointer chain that leads us right to it.

This pointer chain is stable for all tested versions of Windows, and is unlikely to change in the future. It begins at the undocumented `nt!EtwpDebuggerData` global, whose RVA can be found via parsing the PDB of `ntoskrnl.exe`.

```
PWMI_LOGGER_CONTEXT GetCKCLContext(
    IN UINT_PTR EtwpDebuggerData
)
{
    PVOID* debugger_data_silo = *reinterpret_cast<PVOID**>(EtwpDebuggerData +
0x10);
    return static_cast<PWMI_LOGGER_CONTEXT>(debugger_data_silo[2]);
}
```

We will also need to configure the logger's target events (internally called `EnableFlags`). This is done via the `nt!ZwTraceControl` function, which is thankfully exported for all drivers to use.

The function takes a `_WMI_LOGGER_INFORMATION` structure as the input buffer. While undocumented by Microsoft, its definition can be found inside PHNT headers. Inside this structure, we will need to specify what logger to target. This is done by setting the `GUID` and `LoggerName`.

Already having the `_WMI_LOGGER_CONTEXT` structure, extracting the information is simple:

```
kd> dt _WMI_LOGGER_CONTEXT poi(poi(EtwpDebuggerData+0x10)+0x10)
nt!_WMI_LOGGER_CONTEXT
    ...
```

```
    +0x088 LoggerName       : _UNICODE_STRING "Circular Kernel Context
Logger"
    ...
    +0x114 InstanceGuid     : _GUID {54dea73a-ed1f-42a4-af71-3e63d056f174}
```

Upon configuring the logger and starting it, we're ready to roll.

Part 3: Hooking context switches

We now have a function that gets called on each context switch - awesome! Finding the new thread is simple - we're executing in its context, meaning `KeGetCurrentThread` will get us a pointer to it's object.

Looking at the functions called prior to our hook, we notice that the last function that has access to the `OldThread` and `NewThread` parameters is `EtwpLogContextSwapEvent`, where they are passed in `rdx` and `r8`. Breakpointing there shows that `rbx` and `rdi` contain copies of the two arguments.

```
1: kd> r rbx, rdx, rdi, r8
rbx=ffffd8878177d080 rdx=ffffd8878177d080
rdi=ffffd8878627c080 r8=ffffd8878627c080
nt!EtwpLogContextSwapEvent:
fffff8028bbd79d0 48895c2410      mov     qword ptr [rsp+10h],rbx
ss:0018:fffff500a54bbef8=fffff8028bbd7885
```

These registers are both pushed onto the stack in the function prologue, with the current thread (stored in `rdi` and `r8`) coming first:

```
kd> uu EtwpLogContextSwapEvent
nt!EtwpLogContextSwapEvent:
fffff805`81bd79d0 48895c2410      mov     qword ptr [rsp+10h],rbx
fffff805`81bd79d5 55              push    rbp
fffff805`81bd79d6 56              push    rsi
fffff805`81bd79d7 57              push    rdi
```

Looking at the code, we can figure out that `rbx` will be at a constant offset of `0x28` from `rdi` on the stack. Given we know the value of `rdi` (it's a pointer to the current thread), we can scan the stack up from our hook, and look at each possible thread:

```
// We loop until stack_limit - 0x28 to prevent OOB access when checking the
previous thread.
for (ULONG_PTR iterator = rsp; iterator < (stack_limit - 0x28); iterator +=
sizeof(PKTHREAD))
{
```

```
    PKTHREAD thread_at_iterator = *reinterpret_cast<PKTHREAD*>(iterator);

    // If we found our own thread's pointer on the stack
    if (thread_at_iterator == current_thread)
    {
        // Look at the thread at the target offset
        PKTHREAD possible_prev_thread = *reinterpret_cast<PKTHREAD*>(iterator
+ 0x28);
        PDISPATCHER_HEADER possible_dispatcher_header =
reinterpret_cast<PDISPATCHER_HEADER>(possible_prev_thread) - 1;

        const ULONG_PTR possible_prev_thread_raw =
*reinterpret_cast<ULONG_PTR*>(iterator + 0x28);
        // Threads are not stack-allocated.
        if (possible_prev_thread_raw >= stack_base &&
possible_prev_thread_raw <= stack_limit)
            continue;

        // Threads are not in userspace.
        if (possible_prev_thread < MmSystemRangeStart)
            continue;

        // Threads have accessible memory.
        if (!MmIsAddressValid(possible_prev_thread) ||
!MmIsAddressValid(possible_dispatcher_header))
            continue;

        // Reference the thread to check the object type.
        NTSTATUS status = ObReferenceObjectByPointer(
            possible_prev_thread,
            0,
            *PsThreadType,
            KernelMode
        );

        // If the function fails, we can be sure that the address is not one
of a thread.
        if (!NT_SUCCESS(status))
            continue;

        // Dereference the thread, and store it.
```

```
            ObfDereferenceObject(possible_prev_thread);
            previous_thread = possible_prev_thread;
            break;
        }
    }
```

Part 4: Usage & Detection

Many anti-cheat solutions have started hooking context swaps in an effort to create hidden memory regions that are only visible to certain threads in the system. One notable example is Riot Vanguard which uses a different method that I'll definitely write about in the near future.

The hook can also be used to detect threads executing in unsigned memory, as there's little preventing you from walking the stack of the old thread, and seeing whether code is running in any region it shouldn't be.

As for detection, there's the obvious artifact of `HalpPerformanceCounter + 0x70` pointing outside of `ntoskrnl.exe`, and `GetCpuClock` being set to 1 in the CKCL. Although the latter may happen under normal system operation (and could therefore trigger false positives), it's never been set by default over the course of my testing.

Part 5: Epilogue

This is my very first written article, inspired by reading countless posts from people far smarter than I am. One person I should definitely mention is Denis Skvortcov who wrote about this method more than two years ago when reverse-engineering Avast Antivirus.

I should also thank you, the reader, for sticking with me this far - I hope we meet again next time!