Detecting Remote Thread Creation with Windows Driver

medium.com/@s12deff/detecting-remote-thread-creation-with-windows-driver-9901fdbaf7b1

S12 - 0x12Dark Development May 10, 2026

Welcome to this new Medium post, today I will show you an interesting way used by security software to detect the remote thread creation in Windows

Remote thread creation is one of the most common techniques used by malware to inject code into another process. Tools like Cobalt Strike, Metasploit, or custom malware use Windows APIs like `CreateRemoteThread` to start a new thread inside a target process. This allows the attacker to execute arbitrary code in the context of a legitimate process, making detection harder for traditional antivirus solutions

But how do EDR products and security drivers actually detect this? The answer is **kernel callbacks**, specifically, `PsSetCreateThreadNotifyRoutine`

```c
VOID EvtDriverUnload(_In_ WDFDRIVER Driver){
    UNREFERENCED_PARAMETER(Driver);
    NTSTATUS Status;

    THREAD_MONITOR_DBG("%s", "Driver unloading...");

    // Remove the thread notification callback
    Status = PsRemoveCreateThreadNotifyRoutine(ThreadNotifyCallback);
    if (!NT_SUCCESS(Status)) {
        THREAD_MONITOR_ERR("PsRemoveCreateThreadNotifyRoutine failed: 0x%X", Status);
    }

    THREAD_MONITOR_DBG("%s", "Driver unloaded successfully!");
}

VOID ThreadNotifyCallback(IN HANDLE ProcessId, IN HANDLE ThreadId, IN BOOLEAN Create){
    // The notification occurs in the context of the CREATOR process,
    if (Create) {

        HANDLE CurrentProcessId = PsGetCurrentProcessId();

        // ===== THREAD CREATED =====
        if(CurrentProcessId != ProcessId){
            // If thread is created from a different process that means remote thread creation
            // We can log this as a potential injection attempt

            // Filter out System process (PID 4)
            if (CurrentProcessId == (HANDLE)4) {
                return;  // Normal kernel behavior, ignore
            }

            THREAD_MONITOR_DBG("Remote Thread Created: PID=%d, TID=%d (Created by PID=%d)",
                (ULONG)(ULONG_PTR)ProcessId,
                (ULONG)(ULONG_PTR)ThreadId,
                (ULONG)(ULONG_PTR)CurrentProcessId);

        }
        else {
            // Thread Created
        }
    }
}
```

No issues found

Courses: Learn how offensive development works on Windows OS from beginner to advanced taking our courses, all explained in C++.

All Courses

Learn how real Windows offensive development works

0x12darkdev.net

Technique Database: Access 70+ real offensive techniques with weekly updates, complete with code, PoCs, and AV scan results:

Malware Techniques Database

Explore an ever-growing collection of techniques

0x12darkdev.net

Modules: Dive deep into essential offensive topics with our modular **text-training** program! Get a new module every 14 days. Start at just **$1.99 per module**, or unlock **lifetime access to all modules for $100**.

0x12 Dark Development

Learn the best offensive techniques for Windows OS, with content ranging from beginner to advanced levels. All…

0x12darkdev.net

What is PsSetCreateThreadNotifyRoutine?

Windows provides a kernel API called `PsSetCreateThreadNotifyRoutine` that allows a driver to register a callback function. Every time a thread is created or destroyed anywhere in the system, the OS will call your function and give you three pieces of information:

- the PID of the process where the thread will run
- : the TID of the new thread
- a boolean telling you if the thread is being created or destroyed

The key detail here is that this callback runs **in the context of the creator process**, not the target process. This is what makes remote thread detection possible

How Do We Detect Remote Threads?

The logic is simple. Inside the callback, we compare two values:

- `ProcessIdd:`the process that owns the new thread (the target)
- `PsGetCurrentProcessId():`the process that is currently executing (the creator)

If these two PIDs are **different**, it means one process is creating a thread inside another process. That is remote thread creation

And this looks like:

```
VOID {     (Create) {          PsGetCurrentProcessId();         (CurrentProcessId != I
```

There is one exception we need to handle: the System process (PID 4). The Windows kernel itself creates threads in other processes as part of normal system behavior, so we filter it out to avoid false positives

Complete code

```
EXTERN_C_START;DRIVER_INITIALIZE DriverEntry;;;EXTERN_C_END{    NTSTATUS Status;    WDI
```

Testing

To test the detection we can use any tool that performs remote thread injection. For this demo I used a simple program that calls `CreateRemoteThread` on a target process.

Once the driver is loaded, we can see the detection both in DebugView and in the log file:

```
Remote Thread Created: PID=, TID= (Created  PID=)
```

The driver correctly identifies the creator process and the target process, which is exactly the information an EDR would use to decide if the activity is malicious

Conclusion

This is the same fundamental mechanism that real EDR products use to monitor thread creation across the system. Of course, commercial solutions add many more layers on top: they correlate the creator process reputation, inspect the thread start address, check if it points to unbacked memory, and combine this with other telemetry sources like image load callbacks and object handle monitoring.

But the core detection starts here, a simple kernel callback that tells you when one process creates a thread inside another

📌 **Follow me:** YouTube | 🐦 X | 💬 Discord Server | 📸 Instagram | Newsletter

S12.