Remote Process Write Primitive via APC Routines

medium.com/@s12deff/remote-process-write-primitive-via-apc-routines-82c2598c6419

S12 - 0x12Dark Development May 14, 2026

S12 - 0x12Dark Development

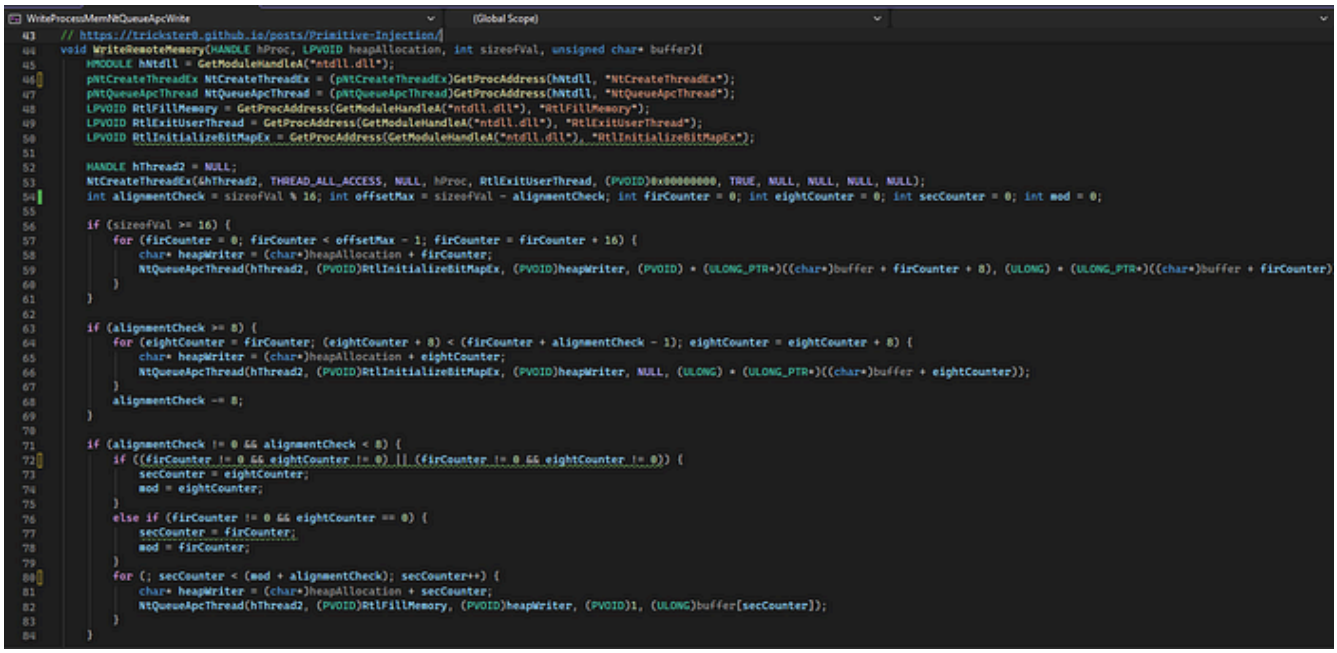
Welcome to this new Medium post, today we'll explore a clever process injection primitive that abuses Windows APC (Asynchronous Procedure Call) routines to write arbitrary data into a remote process, without ever calling `WriteProcessMemory`

This technique, originally documented by trickster0, takes advantage of how APC arguments are passed to queued routines. By carefully selecting Windows API functions whose argument layout matches a memory write operation (like `RtlFillMemory` and `RtlInitializeBitMapEx`), we can effectively turn the APC mechanism itself into a remote write primitive

Why This Matters

Standard process injection workflows typically rely on `WriteProcessMemory` to deliver payloads into a target process. This API is heavily monitored by security solutions, it's one of the first hooks any defensive product places, and calls to it from non-system processes are a strong indicator

By replacing `WriteProcessMemory` with a chain of `NtQueueApcThread` calls, we achieve the same outcome, arbitrary bytes written to a remote address, while completely skipping the most obvious write API. The data is delivered as **function arguments** to legitimate ntdll routines, which the kernel happily executes



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

The Core Idea

The trick lies in choosing the right victim functions:

- **`RtlFillMemory(Destination, Length, Fill)`** : writes a single byte (`Fill`) to a destination buffer. By queuing it with `Length = 1`, we get a 1-byte write primitive where the byte value is whatever we pass as the third argument
- **`RtlInitializeBitMapEx(BitMapHeader, BitMapBuffer, SizeOfBitMap):`**this one is the real gem. Its internal implementation ends up writing the second and third arguments directly to the structure pointed to by the first argument. That gives us a per APC call, where the two 8-byte halves of our data are simply passed as `ApcArgument2` and `ApcArgument3`

So instead of calling `WriteProcessMemory(hProc, dest, buffer, 0x100, NULL)`, we queue ~16 APC calls (each one writing 16 bytes of our buffer) and let the target thread execute them

Methodology

Before diving into the full source code, let's break down the logic step by step. To achieve a remote memory write without using `WriteProcessMemory`, we need to follow these logical steps:

1. First, we need to obtain a handle to the target process with the right permissions. We use the WinAPI `OpenProcess` requesting `PROCESS_VM_OPERATION` (to allocate memory) and `PROCESS_VM_WRITE` (required by the kernel even when we never call `WriteProcessMemory` ourselves). The target process must be accessible at our integrity level
2. Once we have the process handle, we allocate a buffer inside the target's address space using `VirtualAllocEx`. This is the destination where our APCs will write the payload byte by byte. The allocation is unavoidable: APCs cannot create memory regions, they can only write into them
3. We call `NtCreateThreadEx` to spawn a new thread in the target process, pointing it to `RtlExitUserThread` as its start routine. The thread is created (`CREATE_SUSPENDED`). This thread exists only to drain our queued APCs and then cleanly terminate itself, it never executes the start routine until all APCs are processed.

4. Queue the APCs (The Write Loop): This is the heart of the technique. We walk through our payload buffer in chunks and queue one APC per chunk:

- queued against `RtlInitializeBitMapEx`. The function's argument layout means our two 8-byte payload halves get written into the destination address
- also queued against `RtlInitializeBitMapEx` but with one argument set to `NULL`
- queued against `RtlFillMemory`, which writes a single byte per call

Each APC, when executed, performs a tiny memory write without us ever calling a write API directly

Join Medium for free to get updates from this writer.

Remember me for faster sign in

5. Resume and Wait: We call `ResumeThread` on our suspended thread. The thread's APC queue is now drained, each queued routine fires in order, writing its piece of the payload. Oncce all APCs are processed, control returns to `RtlExitUserThread` and the thread terminates

```
┌───────────────────┐│  Local Process   │└─────────┬─────────┘          │ . (hProc)
```

Implementation

Now, let's look at how to translate that logic into C++ code. I have broken down the most important parts.

Resolving the ntdll Functions

We need pointers to the undocumented NT functions and the "victim" routines we'll abuse as write primitives. All of them live inside `ntdll.dll`, which is always loaded in every Windows process, so a single `GetModuleHandleA` + `GetProcAddress` is enough

```
GetModuleHandleA();   (pNtCreateThreadEx)GetProcAddress(hNtdll, );   (pNtQueueApcTh
```

Two things to note here:

- `RtlExitUserThread` is the start routine for our sacrificial thread, it just terminates the thread cleanly.
- `RtlFillMemory` and `RtlInitializeBitMapEx` are our . Their function signatures happen to align perfectly with how `NtQueueApcThread` passes arguments

Creating the Sacrificial Thread

Next, we create the thread that will execute our queued APCs. The key flag is the second-to-last `TRUE`, which translates to `CREATE_SUSPENDED`, the thread is paused immediately after creation so we can queue APCs before it runs

```
HANDLE hThread = ;NtCreateThreadEx(    &hThread,    THREAD_ALL_ACCESS,    ,    hProc,
```

At this point we have a suspended thread inside the remote process. Its APC queue is empty, and nothing has executed yet

The Alignment Math

Before we start queuing APCs, we need to know how to split the payload. Our primitives operate at three granularities (16 bytes, 8 bytes, and 1 byte) so we calculate how many of each we need:

```
alignmentCheck = sizeofVal % ;  offsetMax      = sizeofVal - alignmentCheck;   firCou
```

For a 44-byte payload, for example, we'd write 32 bytes as two 16 byte chunks, 8 bytes as one 8 byte chunk, and 4 bytes as four 1 byte chunks

The 16 Byte Write Loop

This is the most efficient part, each APC writes 16 bytes at once. The trick is in how `NtQueueApcThread` passes arguments to `RtlInitializeBitMapEx`:

```
(sizeofVal >= ) {     (firCounter = ; firCounter < offsetMax - ; firCounter += ) {
```

Here's what's happening: we're not really initializing a bitmap. We're using the fact that `RtlInitializeBitMapEx`'s internal logic writes its 2nd and 3rd arguments to the location pointed to by its 1st argument. By passing 16 bytes of our payload as those two arguments, we get a 16 byte write for free

The 8-Byte and 1-Byte Cleanup Loops

For payloads that don't divide evenly by 16, we mop up the remainder in two phases. First, any 8 byte chunk we can fit:

```
(alignmentCheck >= ) {     (eightCounter = firCounter; (eightCounter + ) < (firCounte
```

Then, any remaining 1–7 bytes are written one at a time using `RtlFillMemory`:

```
(; secCounter < ( + alignmentCheck); secCounter++) {    * heapWriter = (*)heapAllocat
```

`RtlFillMemory(dest, length, value)` is essentially `memset`, by passing `length = 1`, it writes exactly one byte. We sacrifice efficiency for the last few bytes, but it's only ever 1–7 calls at most

Triggering Execution

With all APCs queued, the suspended thread holds them in its kernel-side APC queue. A single call to `ResumeThread` makes the kernel drain that queue in FIFO order, each APC fires, performs its tiny write, and returns. Once the queue is empty, the thread proceeds to its actual start routine (`RtlExitUserThread`) and terminates

```
(hThread);(hThread, FALSE);
```

And that’s all, now let’s see the whole code.

Code

main.cpp

```
   { USHORT Length; USHORT MaximumLength; PWSTR  Buffer;} UNICODE_STRING, * PUNICODE_S
```

Proof of Concept

```
Writting Remote Process Memory!Target Memory : E0000
```

We build custom C2 agents and implants for red teams, giving full control and stealthy operation in real-world tests.

[Custom Agents — 0x12 Dark Development]

[Custom C2 Agents Built for the Real World. Command & Control agents compatible with Mythic, Havoc and leading…]

0x12darkdev.net

Detection

Now it’s time to see if the defenses are detecting this as a malicious threat

Kleenscan API

```
[]
```

YARA

Here a YARA rule to detect this technique:

```
rule {    meta:        description          author          date          reference
```

Here you have my collection of YARA rules:

GitHub — S12cybersecurity/YaraRules: Collection of interesting Yara Rules

Collection of interesting Yara Rules. Contribute to S12cybersecurity/YaraRules development by creating an account on…

github.com

Conclusions

This primitive shows how creative argument abuse can turn legitimate ntdll routines into a fully functional remote write capability, without ever touching `WriteProcessMemory`. By chaining `NtQueueApcThread` with carefully chosen victim functions like `RtlInitializeBitMapEx` and `RtlFillMemory`, we deliver arbitrary bytes into a remote process using nothing more than the kernel's own APC dispatcher. The result is a write primitive that blends into normal thread activity

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

S12.