



Your PC ran into a problem  
and needs to restart

# APC Injection for Offensive Tactics

## A Red Teaming Deep Dive into Windows Internals

A technical exploration of APC queue and thread manipulation techniques for  
advanced offensive operations

# #whoami

## Monish Kumar

@m0n1x90

Security Engineer @ ZOHO

Experienced professional in Red Teaming, Malware Development and Application Security.

Coding full stack applications for fun.

**[Building an open source EDR product](#)**

# Agenda

1

## Introduction

Setting the stage and establishing context

2

## Windows Internals

Fundamental architecture of threads and processes

3

## APC Mechanism

Understanding the Asynchronous Procedure Call framework

4

## APC Injection Techniques

Implementation strategies with code examples

5

## Detection & Evasion

Understanding blue team responses and counter-techniques

6

## Demo

Live demonstration of APC injection techniques

7

## Summary & Q&A

Key takeaways and discussion

# Windows Internals: Threads & Processes

At its core, Windows process execution relies on two fundamental components:

- A **process** is a container for resources with its own private virtual address space. Each process also has a **Process Environment Block (PEB)**, which stores critical process-specific information like image base addresses and loaded modules.
- A **thread** is the basic unit of execution scheduled by the kernel. Each thread possesses its own **Thread Environment Block (TEB)**, containing thread-specific data, including the address of its stack and a pointer to the current exception handler.

For a thread to execute, it utilizes key components:

- The **Stack**: A dedicated memory region used for storing local variables, function call parameters, and return addresses, defining the flow of execution.
- **Registers**: Small, high-speed storage locations within the CPU that hold temporary data, instructions, and memory addresses essential for the thread's active execution and context.

Process injection techniques, like APC injection, manipulate these structures to execute arbitrary code within legitimate processes - making them powerful tools in red team operations.

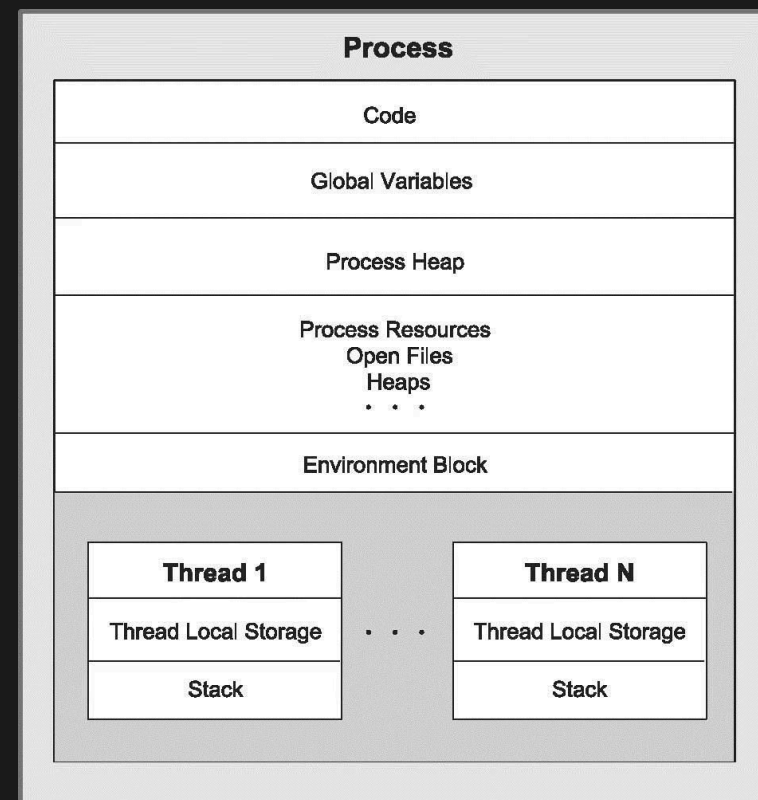


Image Credit @ CODE Magazine

Reference: [Demystifying Windows Internals: Part 1 of 2 – Windows Threads](#)

# Kernel vs User Mode Execution

Windows employs a fundamental dual-mode architecture to ensure system stability and security. This separation defines distinct privilege levels for executing code:



## User Mode (Userland / Ring 3)

This is where typical applications and most of your programs run. It's a restricted environment designed for safety and stability:

- **Limited Privileges:** Applications cannot directly access hardware, modify core operating system structures, or interfere with other applications' memory.
- **Protected Environment:** If a user-mode application crashes, it usually only affects that specific application, not the entire operating system (preventing a "Blue Screen of Death").
- **Indirect Interaction:** To perform privileged operations (like reading a file, allocating memory, or sending network data), user-mode applications must request the operating system's help.



## Kernel Mode (Kernelland / Ring 0)

This is the highly privileged mode where the core operating system components, device drivers, and critical services execute. It has complete control over the system:

- **Full System Privileges:** The kernel has direct and unrestricted access to all hardware, memory, and system resources.
- **Trusted Code:** Only highly trusted code (like the Windows kernel itself and certified device drivers) is allowed to run in kernel mode.
- **System Stability:** A crash in kernel mode is critical and will typically lead to a system-wide failure, often resulting in a Blue Screen of Death (BSOD), as the entire OS relies on this mode's integrity.

# Kernel vs User Mode Execution

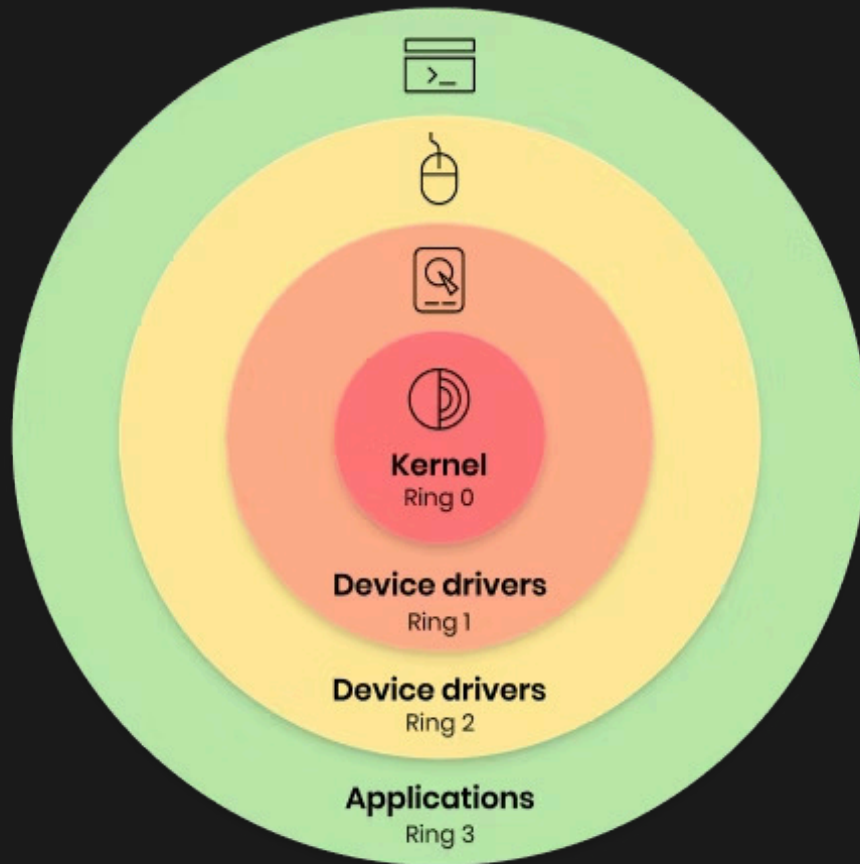


Image Credit @ outbyte.com

Windows uses a dual-mode architecture to ensure stability and security, separating code execution into distinct privilege levels:

- **Ring 0: Kernel Mode (The Core)**

The most privileged layer. It houses the OS kernel and device drivers, with direct access to hardware and memory. Errors here cause system crashes (BSOD).

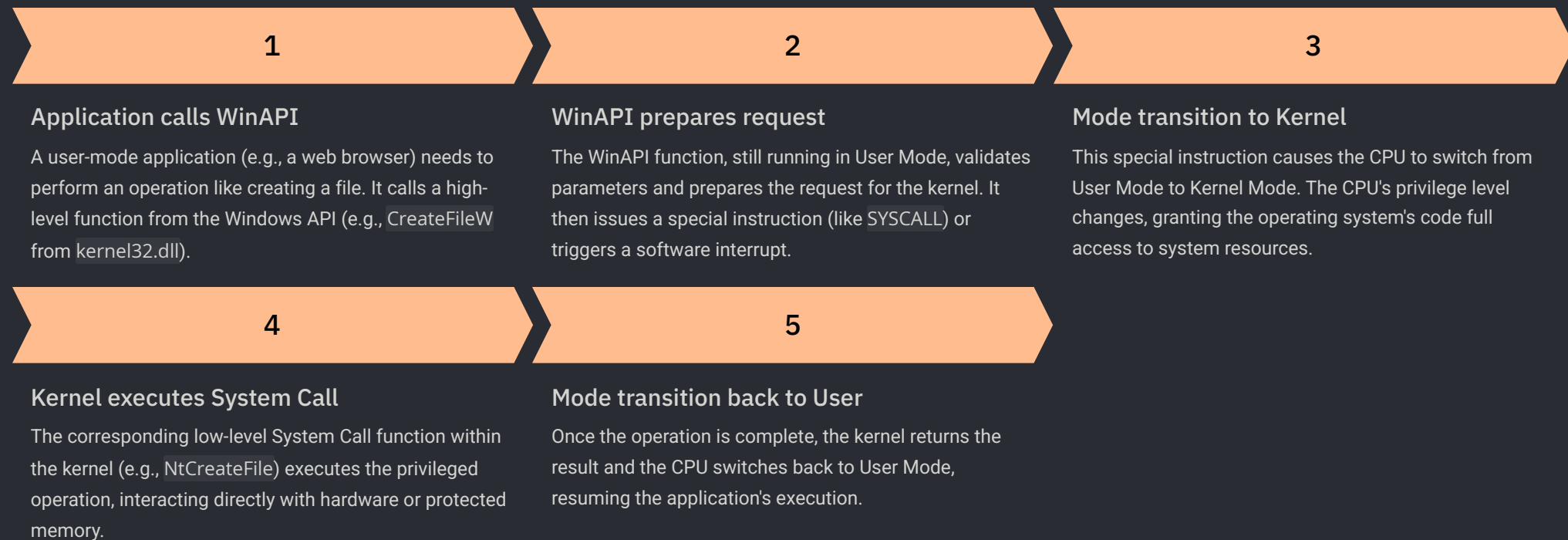
- **Ring 3: User Mode (Outer Layer)**

Where applications run. It's a restricted environment; apps cannot directly access hardware or critical system memory. Crashes usually only affect the application.

User-mode applications perform privileged operations (like file access or network communication) by initiating a **system call**, temporarily transitioning to Kernel Mode. This separation prevents malicious software from harming core system components.

# WinAPIs and System Calls: Bridging the Divide

Since User Mode applications can't directly access privileged resources, they rely on a controlled mechanism to interact with the Kernel. This is primarily done through **WinAPIs**, which often lead to **System Calls**.



Asynchronous Procedure Calls (APCs) can be queued from either mode, but their dispatch mechanisms and implications for offensive operations differ significantly between User Mode and Kernel Mode.

# Win APIs and System Calls: Bridging the Divide

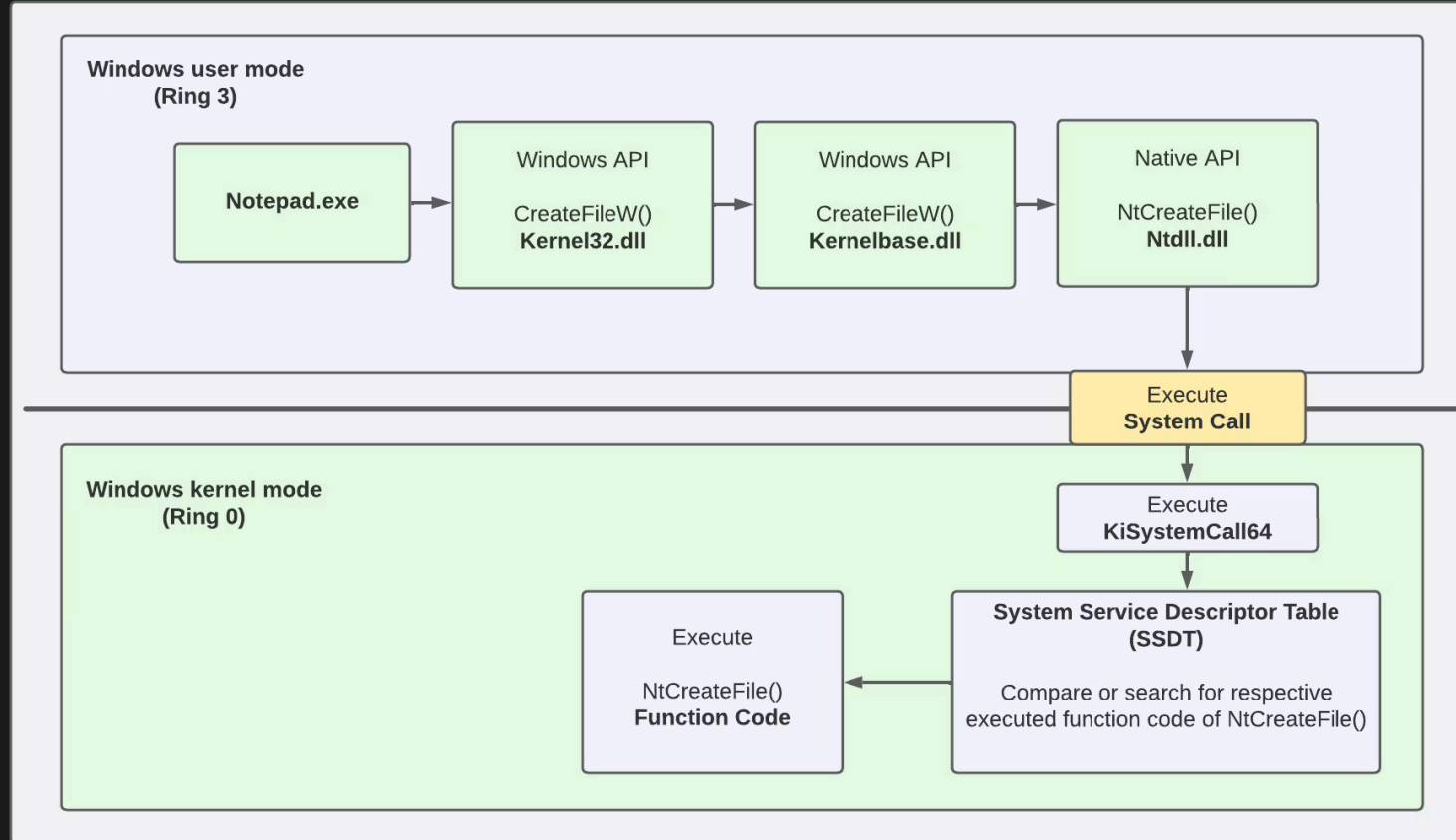


Image Credit: RedOps



# Thread States & Execution Context

## Thread States

A thread can be in one of several states:

- **Running**: Currently executing on a processor
- **Ready**: Ready to execute when processor time is available
- **Waiting**: Blocked for I/O, synchronization, or APCs
- **Terminated**: Execution has completed

## Execution Context

Each thread maintains:

- Thread ID (TID)
- Register values
- Stack (user & kernel mode)
- Thread Environment Block (TEB)
- **APC Queue** for asynchronous calls

```
typedef struct _KTHREAD {  
    // ...  
    union { ULONG Alertable; ... }  
    ULONG ApcQueueLock;  
    // ...  
} KTHREAD, *PKTHREAD;
```

Reference: [https://www.nirsoft.net/kernel\\_struct/vista/KTHREAD.html](https://www.nirsoft.net/kernel_struct/vista/KTHREAD.html)

# Thread Objects: ETHREAD, KTHREAD, TEB

In Windows, threads are complex entities managed by several interconnected data structures, operating across both user and kernel modes:

## 1 ETHREAD (Executive Thread)

This is the executive thread object, representing a thread from the perspective of the operating system's executive layer. It resides in kernel memory and is the primary object exposed to user-mode code. It contains pointers to both the KTHREAD and TEB structures.

## 2 KTHREAD (Kernel Thread)

The kernel thread object is the core representation of a thread within the Windows kernel. It's strictly a kernel-mode structure, containing critical thread state information, such as the thread's execution context, scheduling priority, and its APC queue.

## 3 TEB (Thread Environment Block)

This is a user-mode data structure that holds thread-specific information accessible to user-mode code. Each thread has its own TEB, which contains data like thread-local storage (TLS), exception handling information, and a pointer back to its ETHREAD object.

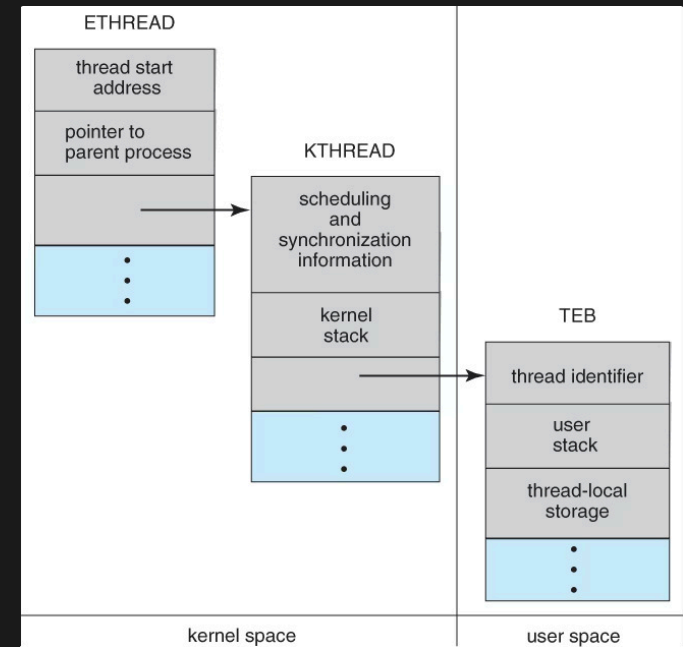


Image Credit @ cs.uic.edu

# What Are Asynchronous Procedure Calls?

**Asynchronous Procedure Calls (APCs)** are a fundamental Windows mechanism that allows:

- Functions to be executed asynchronously in the context of a specific thread
- System-level mechanisms to trigger user-mode callbacks
- Internal signaling between components

Every process in Windows can have multiple threads, and each thread maintains its own dedicated APC queue.

APCs are classified into two primary types based on their origin and execution context:

## User-mode APCs

Queued by user-mode applications or the operating system for execution within a target thread when that thread enters an **alertable** state. They execute in the context of the target thread's user-mode address space. User-mode APCs are a common target for code injection techniques.

## Kernel-mode APCs

Queued by kernel-mode components and execute in kernel mode. They can interrupt a thread regardless of its alertable state. Kernel-mode APCs are used for critical system operations and are more challenging to inject offensively, which needs higher privileges.

They're **built into the Windows thread scheduling system** and are not inherently malicious - they're used extensively by the OS itself.

APCs are like function pointers that get pushed onto a thread's execution queue, interrupting its normal flow to execute arbitrary code in its context.

// Signature for the QueueUserAPC function

```
DWORD QueueUserAPC(  
    PAPCFUNC pfnAPC,    // Pointer to the APC function (our payload)  
    HANDLE  hThread,    // Handle to the target thread  
    ULONG_PTR dwData    // Argument to be passed to the APC function  
);
```

// Signature for the QueueUserAPC2 function

```
DWORD QueueUserAPC2(  
    PAPCFUNC2 pfnAPC,    // Pointer to the APC function (our payload)  
    HANDLE  hThread,    // Handle to the target thread  
    ULONG_PTR dwData,    // Argument to be passed to the APC function  
    ULONG   Flags        // Flags for APC behavior  
);
```

# The Alertable Thread: A Prerequisite for User-Mode APCs

## WinAPIs Enabling Alertable States

For a user-mode APC to be dispatched, the target thread must enter an alertable state. This is achieved when the thread calls specific Windows API functions that support this mechanism. Common WinAPIs include:

- `SleepEx`: Suspends thread execution for a specified interval and allows queued APCs to be executed.
- `WaitForSingleObjectEx`: Waits for a single object to be signalled, entering an alertable state during the wait.
- `WaitForMultipleObjectsEx`: Waits for multiple objects, similarly allowing APC dispatch during the wait.
- `MsgWaitForMultipleObjectsEx`: Extends the above to also wait for messages in a message queue.
- `SignalObjectAndWait`: Atomically signals one object and waits on another, with an alertable option.

## Why the Alertable State?

User-mode APCs require the target thread to explicitly enter an alertable state. This acts as a crucial **synchronization point** and a cooperative **software interrupt** mechanism. APCs are processed from the queue in **First-In, First-Out (FIFO)** order.

- **Controlled Execution**: The thread voluntarily yields control, allowing the OS to process pending APCs. This prevents arbitrary, unscheduled interruptions.
- **System Stability**: It mitigates race conditions, deadlocks, and data corruption by ensuring APCs are dispatched when the thread is in a stable, predictable state.
- **Application Responsibility**: The application (or injected code) must explicitly call an alertable function, contrasting with kernel-mode APCs which can interrupt regardless of thread state.

This cooperative model ensures user-mode APC processing maintains system integrity.

# APC Dispatch Mechanisms

## Alertable Wait States

User APCs only execute when a thread enters an **alertable wait state** via functions such as:

```
SleepEx(DWORD dwMilliseconds, BOOL bAlertable);  
WaitForSingleObjectEx(..., BOOL bAlertable);  
WaitForMultipleObjectsEx(..., BOOL bAlertable);  
SignalObjectAndWait(..., BOOL bAlertable);  
MsgWaitForMultipleObjectsEx(..., ALERTABLE);
```

The **bAlertable** parameter must be **TRUE** for queued APCs to execute!

APC injection effectiveness depends on identifying threads in alertable states or forcing threads into them.

## Kernel Mode Dispatch

Kernel APCs can execute during:

- Thread context switches
- System calls
- Interrupt processing

Kernel APCs are more reliable for attackers as they don't require alertable states, but require elevated privileges to queue payloads into a thread context.

# User-Mode APCs: QueueUserAPC vs. QueueUserAPC2

Understanding the distinctions between these two primary user-mode APC functions is crucial for advanced injection techniques.

## QueueUserAPC (Standard User-Mode APC)

- Classic function for queuing user-mode APCs.
- Straightforward and universally available across Windows versions.
- Allows functions to be executed asynchronously within a target thread's user-mode address space.
- Target thread **must** enter an **alertable** wait state for the APC to be dispatched.
- Common usage patterns are well-documented and known to security tools, making detection relatively straightforward.

## QueueUserAPC2 (Special User-Mode APC)

- Newer function offering extended control via specific behavioral flags.
- Can queue an APC that will execute even if the target thread is not in an explicit alertable wait state (e.g., with **QUEUE\_USER\_APC\_FLAGS\_SPECIAL\_USER\_APC**).
- Enhances the reliability of code injection by reducing dependency on target thread activity.
- Less common use offers potential evasion benefits against simpler detection signatures.

## NtQueueApcThread & NtQueueApcThreadEx (Syscalls)

- These native APIs call the system to invoke a syscall, the underlying mechanisms for **QueueUserAPC** and **QueueUserAPC2**.
- These syscalls are used by higher-level Windows APIs.
- Crucial for advanced, stealthier injection techniques which may not be hooked.
- Can be chained in **Direct Syscalls** / **Indirect Syscalls** evasion


# Early Bird APC Injection: Leveraging Suspended Threads

Early Bird APC injection is a refined technique that capitalises on the Windows thread creation process to inject code into a new thread **before it begins executing** its intended routine. This is primarily achieved by targeting the thread while it is still in a **suspended state**.

## The Suspended State Advantage

When a new process or thread is initiated, it can be created in a suspended state using the `CREATE_SUSPENDED` flag in functions like `CreateRemoteThread` or `NtCreateThreadEx`. This provides a crucial, pre-execution window for the attacker:

- A malicious payload (e.g., shellcode) is written into the target process's memory.
- An APC is then queued to the newly created, suspended thread using `QueueUserAPC` (or similar). The APC's routine points to the injected payload.
- Crucially, the thread is then resumed using `ResumeThread`. As soon as it enters an alertable state (often immediately upon resumption or during its first alertable API call), the queued APC is dispatched and the payload executes.



This technique offers a significant advantage by allowing payload execution before the thread's legitimate code can initiate API hooking or crucial initializations.

It also eliminates race conditions, ensuring reliable execution of the injected code.

Only effective against newly created or suspended threads.

Incorrectly injected code or an incompatible payload can lead to application crashes.

The act of memory allocation, writing, and queuing an APC can still be detected by advanced EDR/AV solutions.

# KAPC: The Kernel's Asynchronous Procedure Call Representation

Deep within the Windows kernel, every Asynchronous Procedure Call (APC) is represented by a KAPC structure. This object serves as the core mechanism for delivering asynchronous execution requests to threads, defining how and when various routines are executed.



## NormalRoutine

This is the user-mode callback function, typically set by APIs like `QueueUserAPC`. It executes within the user-mode context of the target thread when that thread enters an alertable wait state. This routine is responsible for running the attacker's injected payload or the legitimate application's asynchronous task.



## KernelRoutine

A kernel-mode callback that executes at an elevated Interrupt Request Level (IRQL) within the kernel context. It performs pre-processing or post-processing for the APC, often dealing with system-level resources or setting up the environment for the `NormalRoutine`. Attackers with kernel access can leverage this for highly privileged execution.



## RundownRoutine

This is a cleanup routine, also executing in kernel mode. It's invoked if the APC cannot be delivered to its target thread, for instance, if the thread terminates before processing the APC. The `RundownRoutine` ensures proper resource deallocation and prevents memory leaks, maintaining system stability.

Understanding these routines is critical for both implementing and detecting advanced APC injection techniques, as they dictate the privilege level and context of execution.

Reference: [The Vergilius Project: \\_KAPC Structure](#)



# NtTestAlert: Forcing Alertable States

**NtTestAlert** is a powerful, undocumented (though often exposed through public symbols) native function that plays a critical role in controlling Asynchronous Procedure Call (APC) execution flow within the Windows operating system.

## Mechanism and Purpose

Unlike explicit alertable wait functions (e.g., `SleepEx`, `WaitForSingleObjectEx`), `NtTestAlert()` does not cause the calling thread to wait. Instead, it serves as a direct instruction to the kernel:

- It checks if any user-mode APCs are pending for the current thread.
- If pending APCs are found, it immediately dispatches them without requiring the thread to enter a wait state.
- This effectively forces the thread into an alertable state, allowing queued APCs to execute synchronously at the point of the call.
- This works on current thread only, not on remote thread.

## Strategic Use in Injection

### Guaranteed Dispatch

When used in an injected payload, `NtTestAlert()` ensures that any subsequent user-mode APCs queued to the same thread are immediately processed, regardless of the thread's current activity.

### Bypassing Waits

It eliminates the dependency on the target thread entering a traditional alertable wait state, which might be unpredictable or not occur at all in certain application contexts.

### Enhanced Control

Provides attackers with precise control over the timing of their injected code execution, making multi-stage injection techniques more reliable.

### Stealth Potential

Can sometimes bypass detection mechanisms that solely monitor for common alertable wait API calls, as the thread is effectively made alertable "on demand."

# Early Cryo Bird Injections: Evasion via Job Object Freezing

Early Cryo Bird Injection leverages Windows Job Objects to freeze processes, enabling stealthy DLL or shellcode injection via APC without relying on traditional `CREATE_SUSPENDED` flags.

## Key Features

- Leverages undocumented Job Object API (`NtSetInformationJobObject`) to freeze state (`JOBOBJECT_FREEZE_INFORMATION`).
- No `CREATE_SUSPENDED` or `DEBUG_PROCESS` flags, so **lower EDR visibility**.
- Supports injection via APC (`NtQueueApcThread`).
- Unfreeze process to execute payload.

## Injection Steps

1. Create Job Object & enable freeze.
2. Bind Job to process (`STARTUPINFOEXW` attributes) via `UpdateProcThreadAttribute`.
3. Create process inside frozen Job.
4. Allocate & write payload (DLL path / shellcode).
5. Queue APC for malicious payload.
6. Unfreeze process to execute the payload.

## Advantages

- Utilises **native syscalls**, bypassing high-level API monitoring (new technique in wild).
- **Minimal behavioural anomalies** compared to standard injection methods.
- Effective for both DLL and shellcode injection scenarios.

Reference: <https://github.com/zero2504/Early-Cryo-Bird-Injections>

# Detection Mechanisms

## WinAPI Monitoring

EDRs implement user-mode hooks (e.g., IAT, EAT, inline patching) and kernel-mode callbacks (e.g., `PsSetLoadImageNotifyRoutine`, `CmRegisterCallbackEx`) on critical functions like `NtQueueApcThread`, `QueueUserAPC`, `NtCreateThreadEx`, and memory allocation APIs (`NtAllocateVirtualMemory`, `VirtualAllocEx`). Detection occurs if unusual parameters or sequences of calls are observed.

## Memory Scanning

Continuous or periodic scanning of process memory for suspicious regions. This includes identifying pages with `PAGE_EXECUTE_READWRITE` permissions, non-backed executable memory, or code sections that do not align with loaded module characteristics. Focus is on regions injected via cross-process operations and newly allocated executable pages.

## Thread Analysis

Monitoring the execution context of threads for anomalies. This includes checking for `RIP/EIP` redirection to non-image base addresses, abrupt changes in thread state (e.g., from `Waiting` to `UserMode` execution via APC), and unusual call stack patterns that deviate from typical application behaviour or legitimate APC dispatch.

## Behavioral Analysis

Establishing baselines for legitimate APC usage (e.g., I/O completion, timer callbacks) and flagging deviations. Anomalies include APCs queued to processes not typically utilizing them, APCs targeting specific threads (e.g., the main thread of a suspended process), or a high frequency of APC queuing, especially when originating from unexpected parent processes or with unusual APC routine addresses.

# Evasion Techniques

1

## Direct/Indirect Syscalls

Bypass API hooking by implementing syscalls directly/indirectly instead of calling WinAPIs

2

## Obfuscated Memory Allocation

Use multiple small allocations with different permissions, then modify them later

3

## Thread Stack Spoofing

Manipulate thread call stacks to mask the true origin of APC calls

4

## Legitimate Program Abuse

Leverage legitimate programs that use APCs naturally to blend malicious activity

Effective evasion requires understanding both **offensive techniques** and **defensive capabilities** to identify and exploit gaps in detection.

**DEMO TIME?**



**'NUFF POWERPOINT!!**

## Demonstration Flow

This demonstration will walk through a practical example of APC injection, showcasing how it can be used for code execution in a target process.

Classic APC Injection

1

2

Early Bird APC Injection

NtTestAlert APC  
Injection

3

# Summary & Next Steps

## Key Takeaways

APCs provide a powerful mechanism for thread manipulation, allowing for sophisticated control over execution flows.

## Alertable States

These are crucial for user-mode APC execution; without them, APCs remain pending.

## Cross-Process Injection

APC injection works effectively across process boundaries, enabling access to high-trust environments.

## Evolving Landscape

Both offensive and defensive tools are rapidly evolving, necessitating continuous adaptation.

## Stealth & Persistence

APC injection can be leveraged for stealthy code execution and maintaining persistence on a system.

## Internal Structures

A deep understanding of thread states and internal Windows structures is vital for effective APC usage.

## Robust Detection

Modern EDRs monitor API calls, memory, and thread behavior to detect and mitigate APC injection attempts.

## Connect with Me!



Twitter

[@m0n1x90](https://twitter.com/m0n1x90)



GitHub

[m0n1x90](https://github.com/m0n1x90)



LinkedIn

[Monish Kumar](#)



Blog Site

[m0n1x90.dev](https://m0n1x90.dev)



Vettaiyan EDR

[vettaiyan.m0n1x90.dev](https://vettaiyan.m0n1x90.dev)