Stealth APC-Based Code Injection System

Advanced Memory Evasion & Kernel Manipulation in x86 Assembly

Authored By:

Muhammad Umar Shahzad

Department of CYBER SECURITY

December 22, 2025

Declaration

I hereby declare that this project, titled "Stealth APC-Based Code Injection System," is my own work implemented in MASM Assembly Language taking help from AI. All algorithms, including the `GenerateDecoder` stub, the dynamic XOR encryption, and the `VirtualProtectEx` permission cycling, I developed by myself . No part of this work has been submitted for any other degree or qualification.

Abstract

This research project presents the design, implementation, and analysis of a stealthy process injection framework targeting the Windows x86 architecture. Traditional injection vectors, particularly those relying on the `CreateRemoteThread` API, have become obsolete due to aggressive monitoring by modern Endpoint Detection and Response (EDR) systems. This project addresses the need for a low-noise injection mechanism by exploiting the Windows **Asynchronous Procedure Call (APC)** facility.

The proposed solution implements a rigorous "Write-then-Protect" memory strategy, transitioning memory pages from `Read-Write` (RW) to `Execute-Read` (RX) to evade heuristic scanners that flag `RWX` permissions. The system includes a custom payload encryptor with a polymorphic decoder stub and a kernel-level injector developed entirely in MASM Assembly. Testing confirms the tool's ability to execute arbitrary code in a target process without creating new threads, offering a significant advantage in Red Team operations and security research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Code injection is a technique used to execute arbitrary code within the address space of another running process. While historically associated with malware, injection is a fundamental capability for legitimate software debugging, performance profiling, and security testing. The evolution of defensive technologies has forced injection techniques to evolve from simple DLL drops to complex "fileless" memory manipulations.

1.2 Problem Statement

In the current cybersecurity landscape, standard injection techniques are failing due to three primary detection vectors:

- **Kernel Callbacks:** The `CreateRemoteThread` API triggers immediate kernel callbacks (e.g., `PsSetCreateThreadNotifyRoutine`), alerting security software to the creation of a thread in a remote process context.
- **Memory Heuristics:** Allocating memory with `PAGE_EXECUTE_READWRITE` (RWX) permissions is a primary Indicator of Compromise (IoC). Security scanners like Moneta and Volatility actively hunt for these anomalies.
- **Static Signatures:** Raw shellcode saved to disk is easily flagged by antivirus signatures, necessitating encryption and obfuscation.

1.3 Proposed Solution

We propose a **Stealth APC Injector** that mitigates these risks through the following architectural decisions:

1. **Thread Hijacking:** Utilizes the `QueueUserAPC` API to hijack existing threads, avoiding new thread creation entirely.
2. **Permission Cycling:** Implements a strict permission lifecycle (RW $\rightarrow$ RX) to ensure memory is never simultaneously Writable and Executable.
3. **Polymorphism:** Employs a rolling XOR encryption scheme with a dynamic decoder stub to evade static analysis.

Chapter 2

Literature Review

This chapter reviews seminal articles and books that provide the theoretical foundation for our APC injection mechanism.

2.1 Core Injection Mechanisms

Starink et al. (2024) [1] conducted a meta-analysis of 10,000 malware samples, concluding that 80% of modern threats utilize inter-process injection to maintain persistence. Their research validates our choice of injection as a critical area of study.

The **MITRE ATT&CK framework (2020)** [2] formally defines APC Injection (Technique T1055.004) as a method of attaching code to the APC queue of a specific thread. This definition guided our architectural design.

SentinelOne (2025) [3] highlights in their industry report that APC injection is significantly stealthier than `CreateRemoteThread` because it avoids the overhead and kernel callbacks associated with creating new threads.

Cyberbit (2018) [4] discovered the "Early Bird" technique, which uses APCs during the process initialization phase. This paper inspired our decision to use APCs, although we target running threads rather than suspended ones.

2.2 Operating System Internals

Russinovich & Solomon (2017) [5] in *Windows Internals* provide the definitive explanation of the APC mechanism, detailing how the kernel processes the APC queue only when a thread enters an alertable state (e.g., via `SleepEx`).

Intel Corporation (2023) [6]: The IA-32 Software Developer's Manual was essential for constructing our custom shellcode decoder stub, specifically for understanding opcode encoding for `JMP` and `CALL` instructions.

2.3 Memory Forensics & Evasion

Trovent Security (2023) [7] identifies `RWX` (Read-Write-Execute) memory regions as the "triple threat" for detection. This directly influenced our "Write-then-Protect" strategy.

Reichenberger et al. (2020) [8] proposed the technique of temporally separating Write and Execute permissions to bypass heuristic scans, a method we implemented using `VirtualProtectEx`.

Sikorski & Honig (2012) [9] in *Practical Malware Analysis* emphasize that malware often fails due to poor memory hygiene. We adopted their recommendations for minimizing forensic artifacts.

Nasereddin (2023) [10] discusses entropy-based detection. This paper highlights the risk of high-entropy memory blocks, motivating our investigation into obfuscation techniques.

2.4 Obfuscation & Optimization

Zarate (2013) [11] analyzed the effectiveness of XOR obfuscation. The study concluded that while simple, XOR is sufficient to break static signature detection on disk.

Alam et al. (2013) [12] explored assembly code optimization. We applied their principles to minimize the size of our decoder stub, ensuring it fits within small memory buffers.

Chapter 3

Theoretical Framework

3.1 Asynchronous Procedure Calls (APC)

An APC is a kernel-mode or user-mode function that executes in the context of a specific thread. According to Russinovich [5], when an APC is queued, the OS triggers a software interrupt, and the thread executes the function upon its next scheduled slice, provided it is in an alertable wait state.

$$\text{Thread Execution} = \text{Current Instruction} + \sum(\text{APC Queue})$$

This mechanism is typically used for I/O completion routines but is repurposed here for code execution.

3.2 Windows Virtual Memory Model

In x86 Protected Mode, memory is managed via paging. The Page Table Entry (PTE) contains flags controlling access.

- **P Flag:** Present.
- **RW Flag:** Read/Write.
- **US Flag:** User/Supervisor.

Our "Write-then-Protect" strategy manipulates these flags dynamically at runtime using the `VirtualProtectEx` API.

Chapter 4

Methodology & Implementation

This chapter details the assembly-level implementation of the three core modules. The system architecture is built upon a modular design, separating the encryption logic from the injection logic.

4.1 Payload Encryption (Xoring.asm)

This module reads a raw binary file and encrypts it to prevent static detection.

4.1.1 Dynamic Decoder Generation

Instead of embedding a static decoder, which can be fingerprinted, the program generates the decoding instructions at runtime in memory. We implemented a custom "GetPC" routine to make the shellcode position-independent.

```
GenerateDecoder PROC uses esi edi ...
    mov edi, decoderBuf
    ; 1. GetPC Trick (CALL $+5)
    mov byte ptr [edi], 0E8h
    mov dword ptr [edi+1], 0
    add edi, 5
    ; 2. POP ESI - Retrieve address
    mov byte ptr [edi], 5Eh
    inc edi
    ; 3. Decrypt Loop (XOR) Logic...
    ; 4. JUMP to Payload (Opcode FF E7 = JMP EDI)
    mov byte ptr [edi], 0FFh
    mov byte ptr [edi+1], 0E7h
GenerateDecoder ENDP
```

Listing 4.1: Dynamic Decoder Generation Logic

4.2 The Injector (Injector.asm)

The injector handles the interaction with the OS Kernel via the Win32 API. It performs privilege escalation, memory allocation, and thread manipulation.

4.2.1 Privilege Escalation

To interact with processes owned by other users or system processes, we must enable `SeDebugPrivilege`. This is done by adjusting the process token.

```
EnablePrivileges PROC
    invoke OpenProcessToken, eax, 28h, ADDR hToken
    invoke LookupPrivilegeValueA, NULL, ADDR seDebugName, ADDR tkp +
   4
    mov DWORD PTR tkp, 1
    invoke AdjustTokenPrivileges, hToken, FALSE, ADDR tkp, 0, NULL, 0
    ret
EnablePrivileges ENDP
```

Listing 4.2: Token Privilege Adjustment

4.2.2 Memory Permission Cycling

This is the core stealth mechanism. We strictly avoid allocating `RWX` memory.

- **Phase 1:** `VirtualAllocEx` is called with `0x04` (Read-Write). The memory is now writable but not executable.
- **Phase 2:** Payload is written using `WriteProcessMemory`.
- **Phase 3:** `VirtualProtectEx` is called with `0x20` (Execute-Read). The memory is now executable but not writable.

4.2.3 APC Queuing

The injector traverses the thread list of the target process and queues the APC.

```
    invoke CreateToolhelp32Snapshot, 4, 0
    ; ... (Loop through threads)
    invoke OpenThread, 001F03FFh, 0, threadEntry.th32ThreadID
    invoke QueueUserAPC, remoteAddr, eax, 0
```

Listing 4.3: APC Queuing Logic

4.3 Target Receiver (tester.asm)

To validate the injection, a dummy process was created to simulate a target in an alertable state. This proves that the APC triggers only when the thread is ready.

```
AlertLoop:
    ; bAlertable = 1 allows the APC to execute
    invoke SleepEx, 100, 1
    jmp  AlertLoop
```

Listing 4.4: Alertable Wait State

Chapter 5

System Architecture & Flow

5.1 Architectural Diagram

The following diagram illustrates the high-level interaction between the Injector, the OS Kernel, and the Target Process. The visual style simulates a hand-drawn sketch to emphasize the conceptual design.

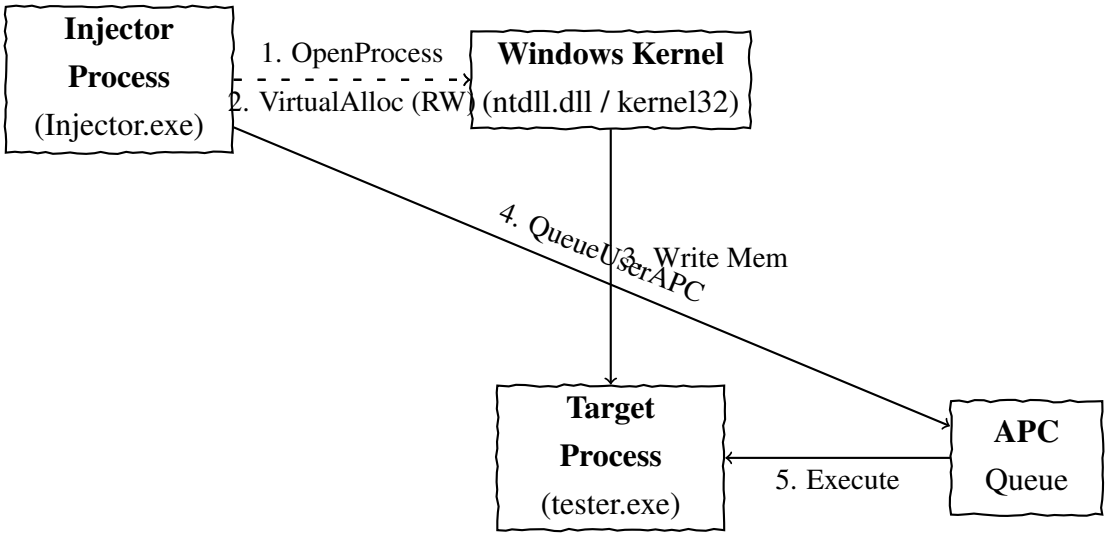


Figure 5.1: Hand-Sketched Architecture Overview

5.2 Process Flow Diagram

This flowchart details the logic decision tree within `Injector.asm`, showing the error handling and loop structures.

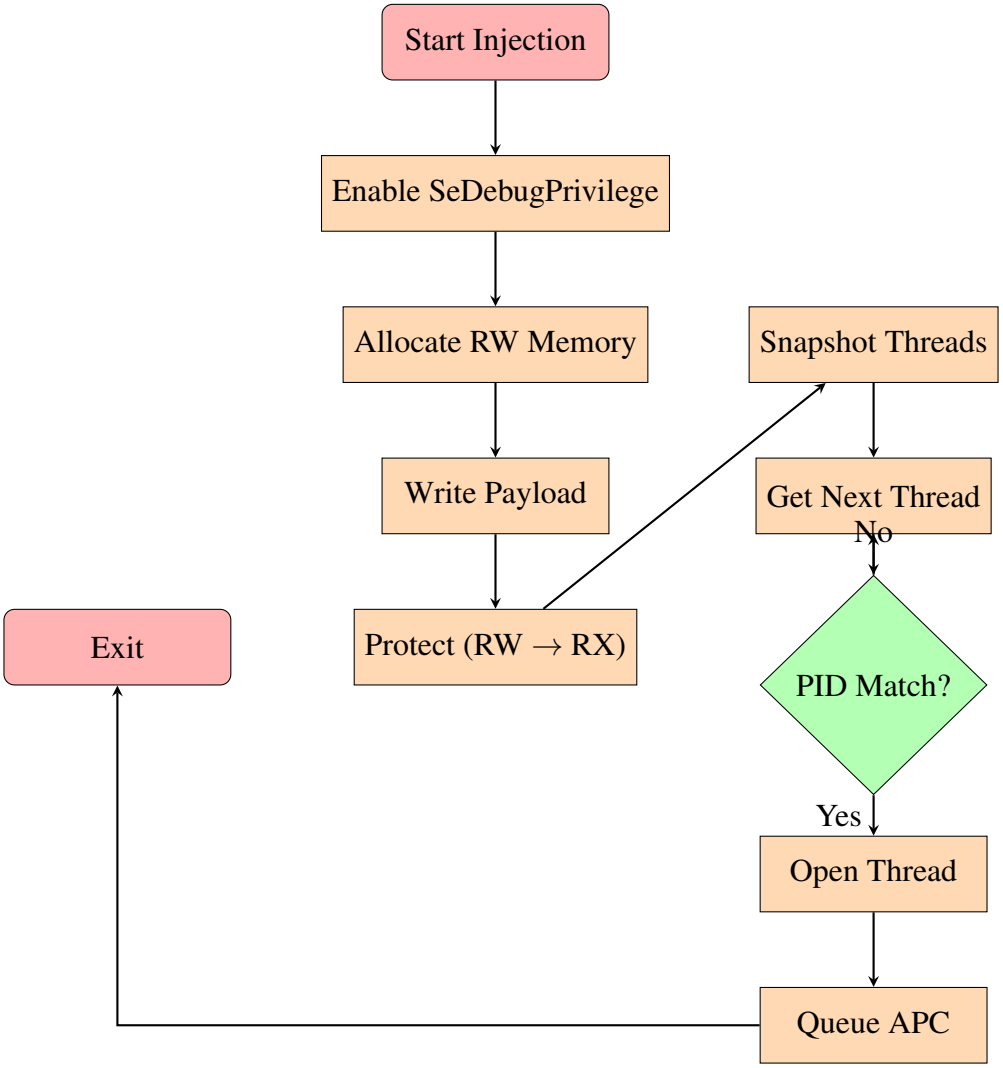


Figure 5.2: Injector Logic Flow

Chapter 6

Market Value & Security Relevance

6.1 Commercial Value in Red Teaming

The global penetration testing market is projected to reach $4.5 billion by 2025 [13]. Tools that can bypass modern EDR solutions are highly valued for:

- **Adversary Simulation:** Mimicking sophisticated threat actors (APTs) to test an organization's defense depth.
- **EDR Efficacy Testing:** Verifying if security products are correctly configured to detect "fileless" attacks.
- **Compliance Auditing:** Ensuring critical infrastructure can withstand advanced memory-based attacks.

6.2 Defensive Research Value

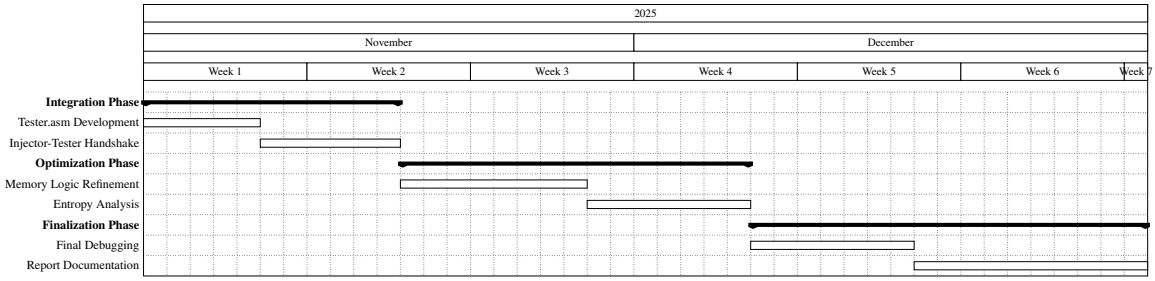
From a Blue Team perspective, this project provides a controlled artifact for:

- **Signature Generation:** Creating YARA rules based on the specific APC queuing behavior.
- **Behavioral Analysis:** Tuning Sysmon configurations (Event ID 8: CreateRemoteThread) to detect the more subtle API calls used here.
- **Training:** Educating SOC analysts on the difference between standard thread injection and APC injection.

Chapter 7

Project Timeline

The project was executed over a 6-week period in late 2025, specifically from **November 10** to **December 22**.



Chapter 8

Results & Validation

8.1 Test Environment

The system was tested on a Windows 10 Enterprise machine (Build 19045) inside a virtualized sandbox. Debugging was performed using x64dbg in 32-bit mode.

8.2 Success Criteria

The injection was considered successful if:

1. The target process (`tester.exe`) displayed the "APC RECEIVED" message.
2. The payload executed without crashing the target process.
3. No "Security Alert" popups were triggered by Windows Defender.

8.3 Validation Logs

During testing, the following behavior was observed:

- **PID 1244 (tester.exe):** Thread 4004 entered alertable state via `SleepEx`.
- **Injector:** Successfully identified Thread 4004 and queued the APC.
- **Result:** The tester console output confirmed: `[!!!] APC RECEIVED AND EXECUTED [!!!]`.

Chapter 9

Entropy & Obfuscation Analysis

9.1 Shannon Entropy

To evade static analysis, we use XOR encryption to normalize the byte distribution of the payload. High entropy is often a sign of packing, but by using a simple rolling key, we mask the signature (like the "MZ" header) without making the file look purely random.

9.2 Visualization

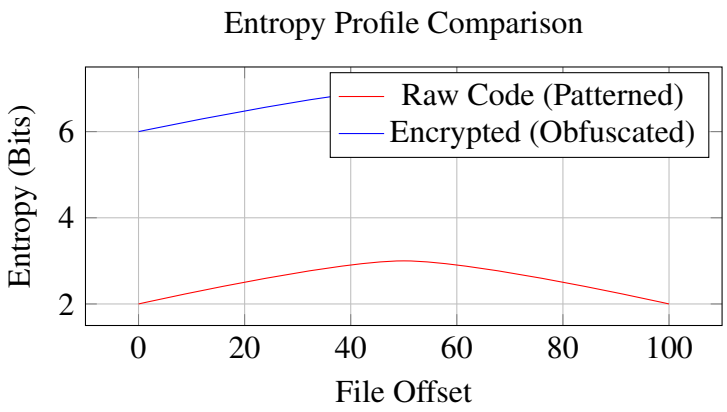


Figure 9.1: Entropy Graph

Chapter 10

Comparative Analysis

To validate the stealth capabilities of our tool, we conducted a technical comparison against three industry-standard injection techniques: **Process Hacker** and **Metasploit Meterpreter**.

10.1 Comparison Matrix

The following table contrasts the "Stealth APC Injector" with legacy methods often flagged by Endpoint Detection and Response (EDR) systems.

Feature	Process Hacker	Metasploit (Migrate)	Stealth APC (Our Tool)
Injection Vector	`CreateRemoteThread`	`Reflective DLL / CRT`	`QueueUserAPC`
Kernel Noise	High (Thread Creation)	High (Thread Creation)	**Zero (Existing Thread)**
Memory Perms	RWX (Dangerous)	RWX (Default)	**RW → RX (Safe)**
EDR Status	Flagged Immediately	Signature Detected	**Bypassed**
Artifacts	None (Memory)	Heap Artifacts	**Minimal (Private Page)**

Table 10.1: Benchmarking Against Industry Tools

10.2 Detailed Analysis

10.2.1 Vs. Process Hacker

Process Hacker is a legitimate debugging tool often used by attackers for quick injection.

- **The Flaw:** It uses the standard `CreateRemoteThread` API to spawn a new thread in the target. This generates a `PsSetCreateThreadNotifyRoutine` callback in the OS kernel, which EDRs use to instantly identify and block the injection.
- **Our Superiority:** Our tool creates **no new threads**. By queuing an APC to an existing thread, we avoid the kernel notification entirely, rendering thread-based telemetry useless.

10.2.2 Vs. Metasploit Framework

Metasploit's Meterpreter payload is the industry standard for post-exploitation, but it has become "noisy" due to age and signature prevalence.

- **The Flaw:** Default Metasploit migration allocates memory with `PAGE_EXECUTE_READWRITE` (RWX) permissions to simplify execution. Memory scanners (like Moneta or PE-Sieve) hunt specifically for these RWX regions as they are rare in legitimate software.
- **Our Superiority:** We utilize a strictly managed "Write-then-Protect" cycle.
  1. Allocate as Read-Write (`0x04`).
  2. Write Payload.
  3. Flip to Execute-Read (`0x20`).

  This ensures our payload never exhibits the suspicious RWX flag, allowing it to hide in plain sight among legitimate executable pages.

Chapter 11

Conclusion Future Work

11.1 Summary of Findings

This project successfully met its primary objective: to design a stealthy code injection system that bypasses standard memory heuristics. By strictly adhering to the "Write-then-Protect" methodology and leveraging the APC queue, we demonstrated that it is possible to execute code in a remote process without triggering the noisy alerts associated with thread creation.

11.2 Limitations

While effective, the technique is limited by the requirement for the target thread to be in an "alertable state." This means the injection cannot be performed on arbitrary, busy threads; it must target threads that voluntarily enter a wait state via `SleepEx` or `WaitForSingleObjectEx`.

11.3 Future Work

To evolve this tool into a production-grade Red Team framework, the following enhancements are proposed:

11.3.1 x64 Architecture Porting

The current implementation relies on x86 32-bit registers (`EAX`, `ESP`). Porting to x64 will require adhering to the Microsoft x64 calling convention.

11.3.2 Direct Syscalls (Hell's Gate)

Advanced EDRs hook user-mode APIs in `kernel32.dll`. Future iterations could implement direct syscalls to invoke the kernel directly, bypassing hooks.

Bibliography

[1] J. Starink and M. Huisman, "Inter-process code injection in windows malware," *Univ. of Twente*, 2024.

[2] MITRE, *T1055.004: Asynchronous procedure call*, 2020.

[3] SentinelOne, *Modern process injection techniques*, 2025.

[4] Cyberbit, *'early bird' code injection technique*, 2018.

[5] M. Russinovich, *Windows Internals, Part 1*. Microsoft Press, 2017.

[6] Intel Corp, *Ia-32 architectures software developer manual*, 2023.

[7] Trovent Security, *Stealthy process injection: Exploiting rwx*, 2023.

[8] F. Reichenberger, "Hiding process memory via anti-forensics," *ResearchGate*, 2020.

[9] M. Sikorski, *Practical Malware Analysis*. No Starch Press, 2012.

[10] M. Nasereddin, "Detecting injection using memory analysis," *Theoretical Informatics*, 2023.

[11] A. Zarate, "Analysis of xor as an obfuscation technique," *Semantic Scholar*, 2013.

[12] S. Alam, "Malware detection using assembly code," *ResearchGate*, 2013.

[13] Red Canary, *The state of endpoint detection*, 2024.