Advanced Evasion Techniques Against Microsoft EDR and Cloud-Based Protection

Cybersecurity Research Report

An-Najah National University
Faculty of Engineering and Information Technology
Department of Network and Information Security

Prepared by: Yazan Azmi Balawneh
Submitted to: Dyaa Tumezeh
Submission Date: July 22, 2025

Table of Contents

1. Introduction

Background on EDR and Cloud-based AI/ML Protections

In recent years, security solutions such as Endpoint Detection and Response (EDR) and cloud-based protection mechanisms have significantly evolved. Traditional signature-based antivirus tools are no longer sufficient against modern threats, which has led vendors like Microsoft to adopt behavior-based, machine learning (ML)-powered detection systems. EDR is designed to monitor, analyze, and respond to suspicious activity at the endpoint level, often in real-time. It inspects processes, memory allocations, API usage, and process injection behavior.

On the other hand, Cloud-based Protection (e.g., Microsoft Defender for Endpoint Cloud AI) offloads telemetry data from the endpoint to the cloud for deeper inspection using advanced ML models. This layer can detect anomalies that are missed locally by EDR, such as encrypted or obfuscated shellcode behavior, indirect syscalls, and unusual memory operations.

These technologies, while effective, pose a significant challenge for red teamers, malware analysts, and researchers testing post-exploitation techniques. This report explores practical evasion strategies against both layers, under realistic test conditions.

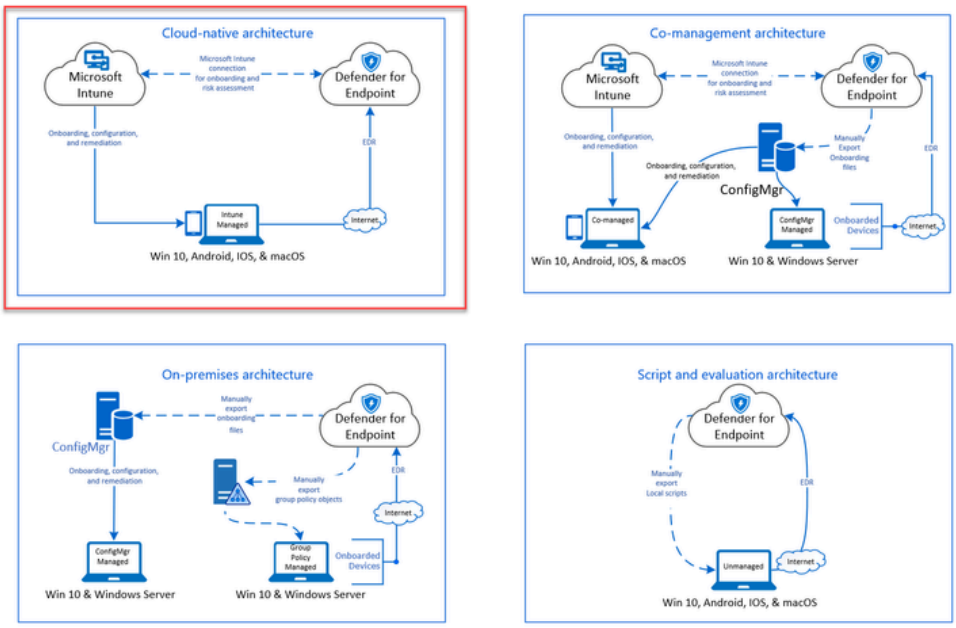

Figure 1.1 – Microsoft Defender for Endpoint Architecture

Objectives of the Evaluation

The objective of this report is to evaluate the effectiveness of four advanced evasion techniques in bypassing both Microsoft EDR and Cloud Protection layers. Specifically, this research aims to:

- Test and compare various injection and execution methods, including shellcode, DLL, and Native Messaging techniques.
- Measure detection response from local EDR and cloud-based ML engines.
- Highlight trade-offs between stealth, complexity, and reliability.
- Provide insights on which techniques are currently effective and which are flagged.

Summary of the Four Techniques Tested

Four advanced techniques were tested under controlled conditions, with Microsoft Defender (EDR and Cloud Protection) fully enabled:

- Early Cryo Bird Shellcode Injection:
  Uses Job Object freezing and APC (Asynchronous Procedure Call) to inject shellcode stealthily without using CREATE_SUSPENDED.
- Direct Syscalls with Encrypted Shellcode:
  Avoids IAT and API monitoring by using XOR-encrypted shellcode and executing it via direct system calls (Nt* functions) using SysWhispers2.
- ZigStrike Injection (XLL and DLL):
  Uses Zig-based loader with anti-sandbox checks and various injection methods (Manual Mapping, Thread Hijacking) via Excel XLL and C++ DLL+SFX dropper.
- Native Messaging API via Covert C2:
  Establishes a C2 channel using browser extension and native messaging API. Allows persistent, stealthy communication and command execution.

2. Methodology

To evaluate the effectiveness of modern evasion techniques against Microsoft Defender for Endpoint, including both local EDR components and cloud-based AI/ML protections, a consistent test environment was prepared. Each technique was executed in isolation and measured against the same criteria.

Test Environment

- Operating System: Windows 10 Pro (Build 19045.5371)
- Security Configuration:
  - Microsoft Defender for Endpoint (latest, fully updated)
  - Real-time protection: Enabled
  - Cloud-delivered protection: Enabled
  - Automatic sample submission: Enabled
- Connectivity: Internet ON during cloud detection testing
- Testing Cycles: Each method was tested twice:
  - First with internet disconnected (EDR only)
  - Then with internet connected (EDR + Cloud AI)

Payload Preparation Process

- Base Payload: Reverse shell EXE from previous project
- Shellcode Generation: Converted to shellcode using Donut
- Encryption & Obfuscation:
- XOR encryption , Base64 encoding using myEncoder3.py
- Junk code, dummy variables, and string hiding (where applicable)
- Some techniques used manual mapping, direct syscalls, APC injection, or browser-native execution to evade traditional detection mechanisms.

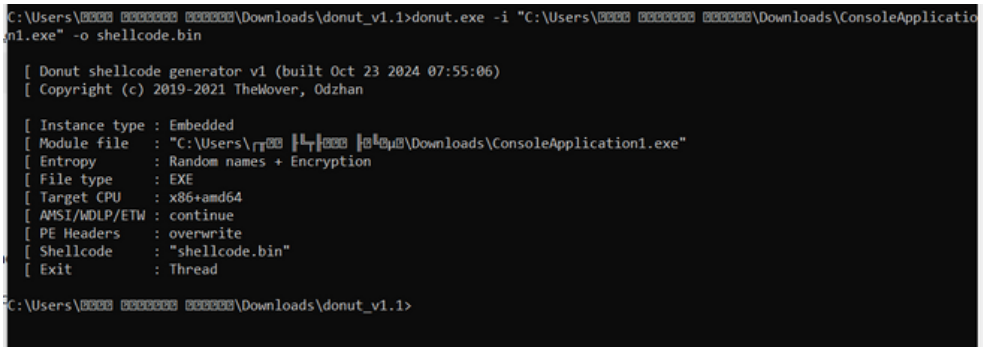

Figure 2.1 – genrate shell code

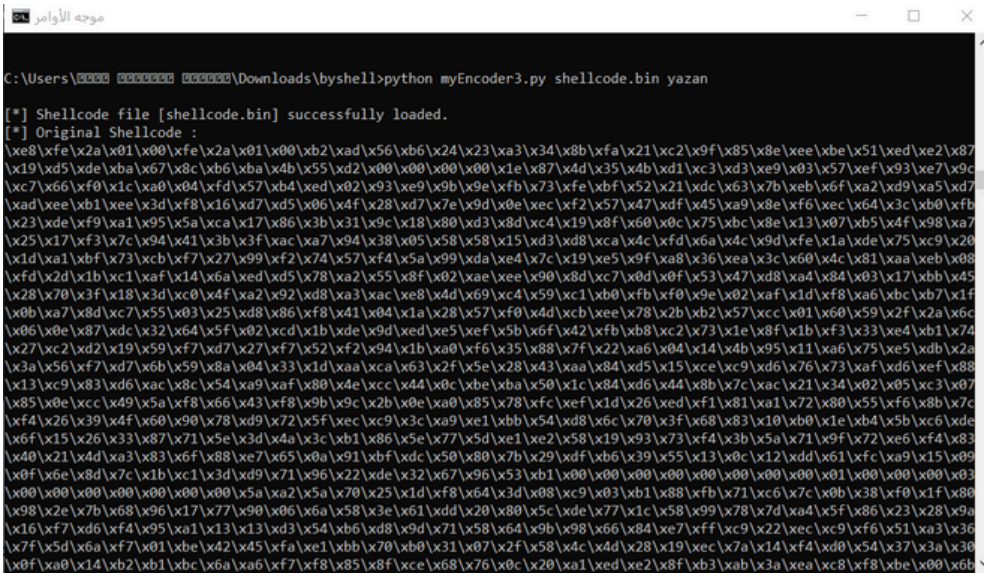

Figure 2.2 – encrypt and encode shell code

Detection Criteria

Each evasion technique was evaluated using the following success metrics:

- Microsoft EDR Detection:

  Any alert, process block, or quarantine event triggered locally.
- Cloud Protection Detection:

  Any delayed detection or classification by Microsoft Security Center .
- Execution Result:

Whether the reverse shell or intended payload was successfully executed and connected without interruption.

3. Technique 1 – Early Cryo Bird Shellcode Injection

3.1 Overview

- Early Cryo Bird Injection is an advanced injection technique that leverages undocumented behavior in Windows Job Objects to achieve stealthy code execution. Rather than using suspicious flags such as CREATE_SUSPENDED or DEBUG_PROCESS, which are commonly flagged by Microsoft Defender for Endpoint (EDR), this technique initiates a target process in a pre-frozen state using the NtSetInformationJobObject API and a JOBOBJECT_FREEZE_INFORMATION structure.
- While the target process remains frozen, memory is allocated, and a DLL path or shellcode is written silently. An Asynchronous Procedure Call (APC) is queued to execute LoadLibraryW or a shellcode entry point. Once the process is "thawed" by lifting the freeze, the APC executes, injecting the payload into the target process.
- This technique avoids common detection paths by bypassing high-level monitored APIs, reducing behavioral anomalies, and mimicking legitimate process activity. It supports both DLL and shellcode injection.

3.2 Step-by-Step

- Create Job Object

A new Job Object is created using CreateJobObjectW, acting as a container for the target process.

- Enable Freeze Behavior

The Job is configured using NtSetInformationJobObject with the JOBOBJECT_FREEZE_INFORMATION structure, setting freeze to TRUE.

- Initialize Attribute List

STARTUPINFOEXW is initialized and linked to a PROC_THREAD_ATTRIBUTE_LIST to define extended process attributes.

- Bind Job to Attributes

The Job Object is linked to the attributes via UpdateProcThreadAttribute.

- Create Frozen Target Process

A target process (e.g., dllhost.exe) is created with CreateProcessW, starting frozen inside the Job.

- Allocate Memory in Target

Memory is allocated in the frozen target using NtAllocateVirtualMemoryEx.

- Write DLL Path into Memory

The encoded path to the DLL is written using NtWriteVirtualMemory.

- Queue APC to Load DLL

An APC is queued using NtQueueApcThread, targeting LoadLibraryW with the DLL path as the argument.

- Thaw the Process

The freeze flag is disabled via NtSetInformationJobObject, resuming process execution.

- Execute APC → DLL is Loaded

When the process becomes alertable, the queued APC is triggered, and the DLL or shellcode is injected.

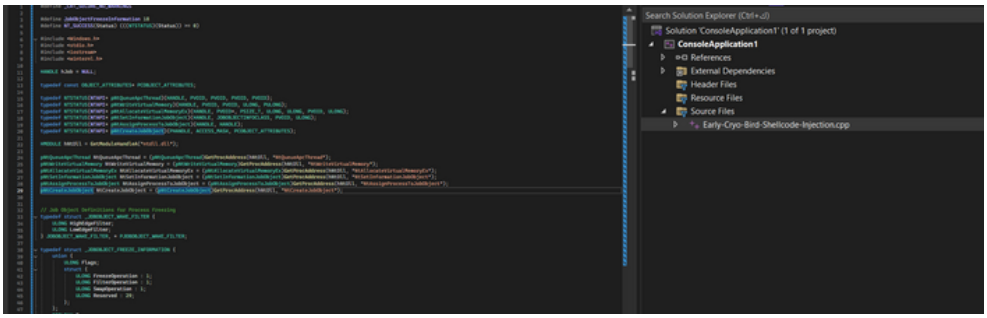

Figure 3.1 – Setup of NT Functions and Job Object Freezing Logic

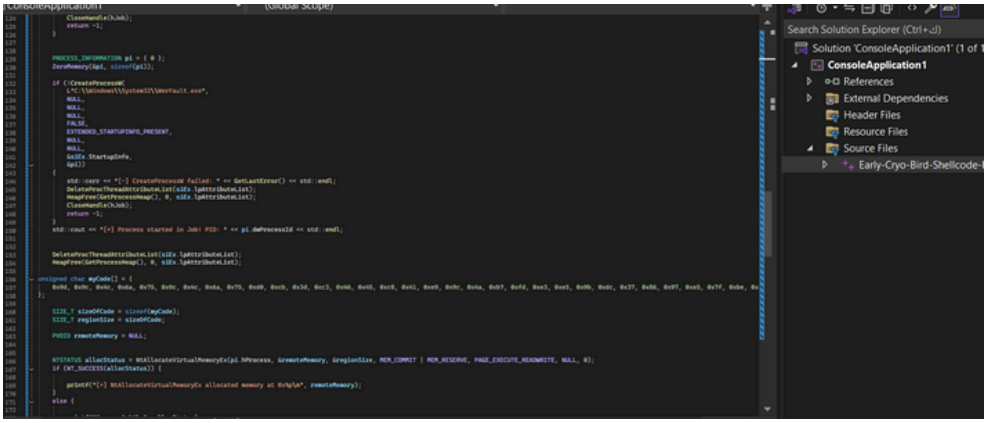

Figure 3.2 – Target Process Creation and Shellcode Injection

3.3 Results and Observations

- Malware Behavior: The injected payload was a reverse shell (encoded using XOR, converted via Donut).
- Microsoft EDR Detection: ❌ Not detected. Local real-time protection did not block the injection or execution.
- Microsoft Cloud Protection: ✅ Detected. Upon restoring internet connection, the payload was flagged by Microsoft Defender Cloud AI due to behavioral analysis (unusual memory permissions).
- Offline Bypass: When internet connectivity was disabled, the payload consistently bypassed both real-time and post-execution analysis.

Figure 3.6 shows the detection log from Microsoft Defender Security Center. The injected payload (ConsoleApplication1.exe) was flagged as Trojan:Script/Wacatac.C!ml, and several malicious behaviors were observed, including:

- Code injection into WerFault.exe
- Suspicious memory allocation and execution
- Behavioral patterns resembling Wacatac malware

The alert level was escalated after telemetry was uploaded, confirming that local Microsoft EDR did not initially detect the injection, but cloud-based ML analysis identified the threat and blocked execution.

This confirms that the technique is highly effective against local Microsoft EDR, but susceptible to cloud-based ML/AI detection when telemetry is uploaded.

Figure 3.3 – File Downloaded Without Detection by Microsoft EDR

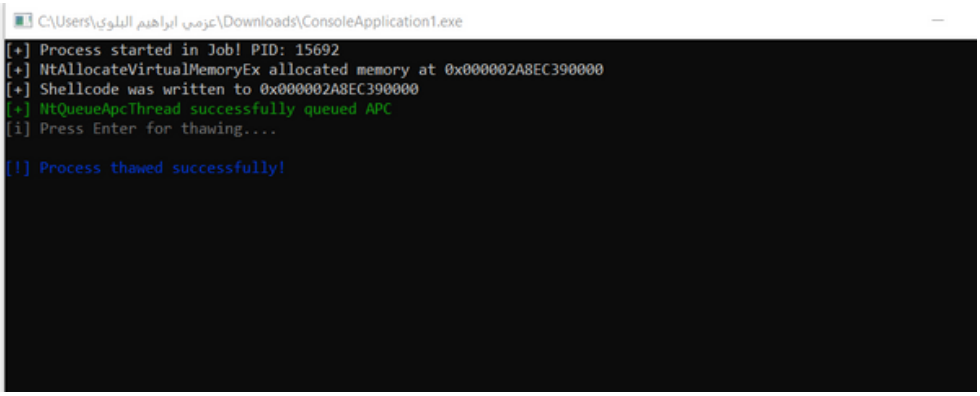

Figure 3.5 – Shellcode Injected and Executed Without Local EDR Detection

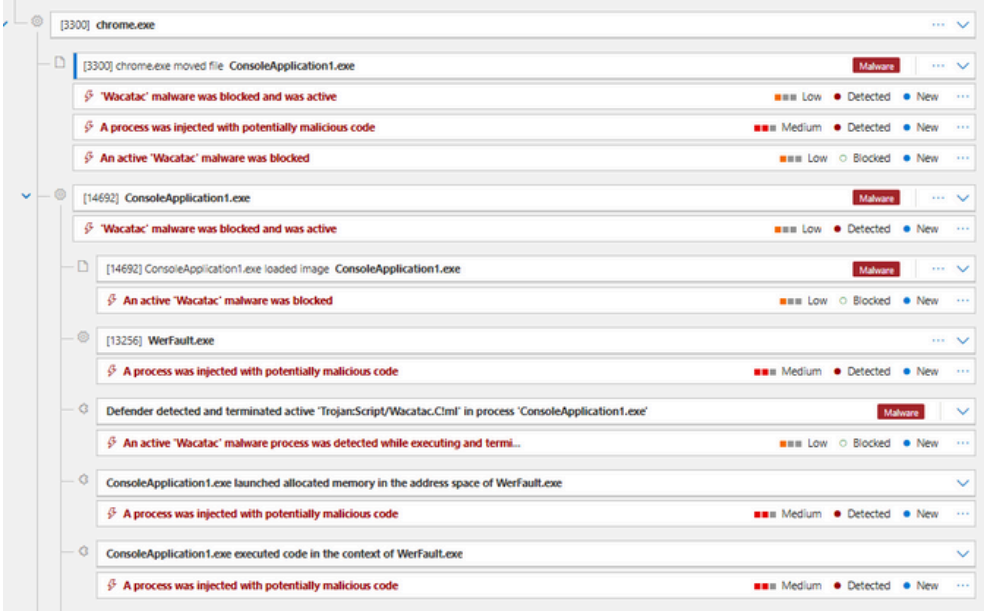

Figure 3.6 – Microsoft Defender Detection Triggered by Cloud-Based Machine Learning

4. Technique 2 – Direct Syscalls with Encrypted Shellcode

3.1 Overview

- This technique leverages direct syscalls to execute an XOR-encrypted shellcode in memory, bypassing traditional user-mode API monitoring and static analysis by Microsoft Defender for Endpoint (EDR). Instead of relying on high-level Win32 APIs such as VirtualAlloc, WriteProcessMemory, or CreateThread, the injection is performed using low-level NT system calls (e.g., NtAllocateVirtualMemory, NtWriteVirtualMemory, NtCreateThreadEx) via the SysWhispers2 framework.
- The main objective of this technique is to avoid:
- IAT (Import Address Table) scanning
- User-mode hooking
- Win32 API monitoring
- EDR rule-based detections triggered by standard injection patterns
- To further obfuscate the payload, the shellcode was XOR-encrypted and Base64-encoded, then decoded in memory right before execution.

3.2 Step-by-Step

- Generate Shellcode:

A reverse shell EXE (from a previous project) was converted into shellcode using Donut.

- Encrypt the Payload:

The shellcode was XOR-encrypted and Base64-encoded using a custom script (myEncoder3.py), making static signature detection ineffective.

- Prepare the Loader:

A custom C++ loader was built using SysWhispers2, which allows invoking NT system calls directly (bypassing user-mode hooks).

- Allocate Memory (Direct Syscall):

Memory is allocated inside the current process using NtAllocateVirtualMemory.

- Write Decrypted Shellcode:

The XOR-decoded shellcode is written to the allocated memory via NtWriteVirtualMemory.

- Execute Shellcode:

Execution is triggered using NtCreateThreadEx directly—no call to CreateThread or any high-level API.

- Payload Activated:

The reverse shell connects to the attacker's listener (e.g., netcat or msfconsole), completing the post-exploitation phase.

Figure 4.1 – code phase 1

Figure 4.2 – code phase 2

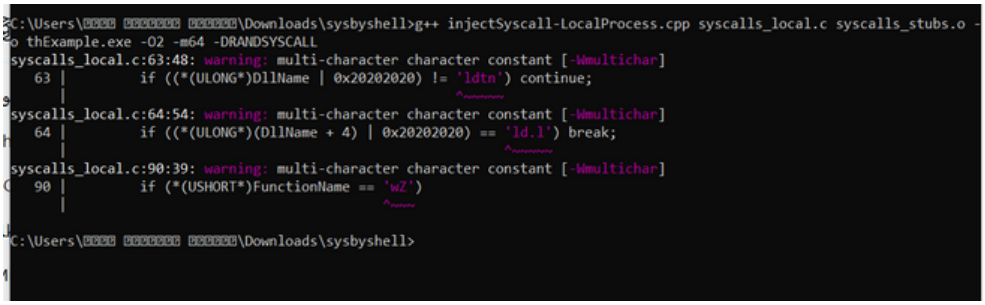

Figure 4.3 – compile it

Microsoft EDR Detection: Not detected

No alert or blocking was triggered during memory allocation, injection, or shellcode execution.

Microsoft Cloud Protection: Detected

After telemetry was uploaded, the payload was flagged based on behavioral analysis and was classified as malicious in the Defender Security Center.

Offline Execution: Bypassed successfully

When internet access was disabled, the technique ran without interruption or post-analysis detection.

5. Technique 3 – ZigStrike Injection via XLL and DLL

3.1 Overview

- This technique utilizes the ZigStrike framework — a modular payload generator and shellcode loader written in Zig — designed specifically to bypass EDR and sandbox environments using multiple evasion strategies. The framework provides a web-based interface to generate payloads in various formats such as XLL (Excel Add-in) and DLL, with multiple injection options including manual mapping and thread hijacking.
- In this evaluation, two variations of ZigStrike were tested:
- (A) XLL payload with manual mapping + anti-sandbox features
- (B) DLL payload injected via a custom C++ executable + packed using WinRAR SFX dropper

3.2 Step-by-Step

A. XLL + Manual Mapping (Excel-based Execution)

Generated a payload using ZigStrike's web interface with the following options:

- Output format: .xll (Excel Add-in)
- Injection: Manual Mapping
- Anti-analysis: Enabled (e.g., Anti-VM, TPM check, Domain join check)
- Delivered the .xll file to the target machine.

When opened in Microsoft Excel, the XLL executed the embedded shellcode via manual mapping.

- The shellcode connected to the attacker's listener, completing the reverse shell execution.

B. DLL + SFX Dropper Execution

- Generated a payload as a .dll using ZigStrike.
- Built a custom C++ injector executable to load the DLL into memory.
- Combined both the injector EXE and DLL into a WinRAR SFX archive configured for silent execution.

Upon execution, the injector loaded the DLL into memory and executed the payload.

Figure 5.1 – zigstrik webpage

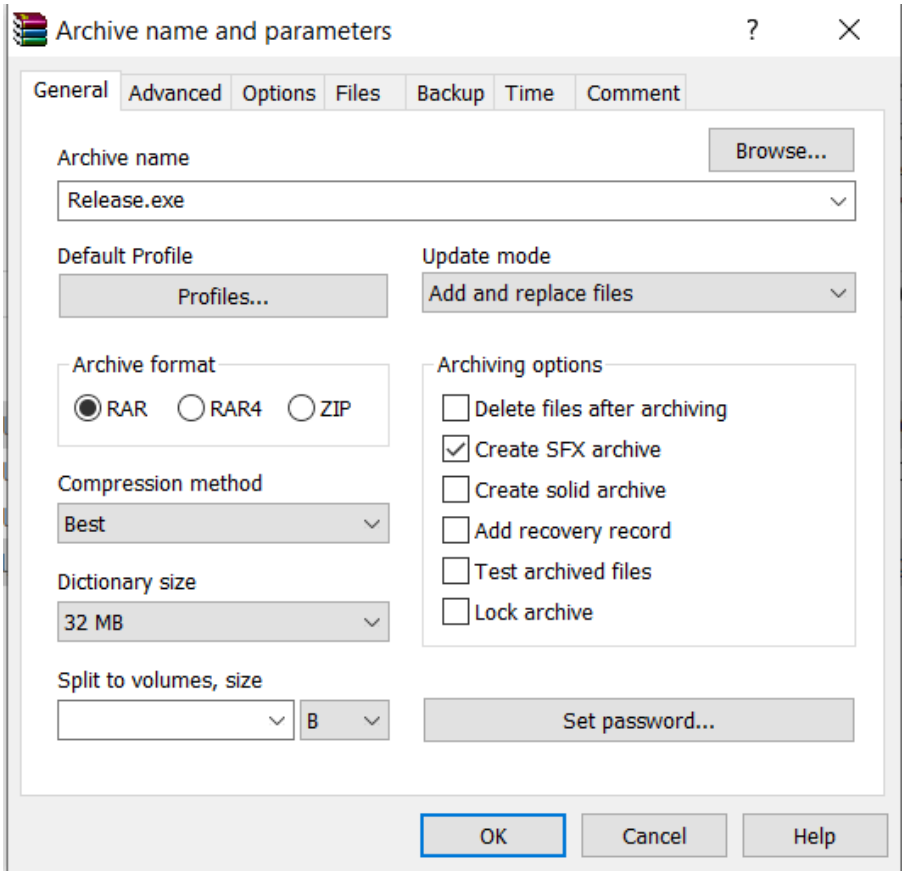

Figure 5.2 – winrar sfx

5.3 Results and Observations

During testing of the ZigStrike payloads, the results varied depending on the delivery method and output format used. The first variation involved generating an XLL payload using the ZigStrike web interface with manual mapping and anti-analysis features enabled. Upon delivery to the target system, the XLL file was blocked by Microsoft Excel, which refused to load the add-in due to it originating from an untrusted source. This was not a simple warning but an active block, preventing the payload from executing unless manually unblocked by the user via the Trust Center settings. Notably, Microsoft Defender EDR did not detect or flag the file itself, indicating that the blocking decision came from Office's built-in security policy rather than malware analysis.

In contrast, the second variation utilized a DLL payload generated by ZigStrike, which was injected using a custom C++ loader and packed within a WinRAR SFX executable. This approach was immediately detected and blocked by Microsoft Defender EDR. The detection was triggered by suspicious process behavior, including manual DLL mapping, memory allocation, and execution patterns commonly associated with reflective DLL injection. Upon telemetry submission, Microsoft Cloud Protection further escalated the threat classification.

These results suggest that while ZigStrike's manual mapping and anti-sandbox features can bypass endpoint detection, execution may still be blocked by other platform-specific security policies (like Office Trust Center). Furthermore, DLL-based execution remains highly detectable, even when obfuscated or packed.

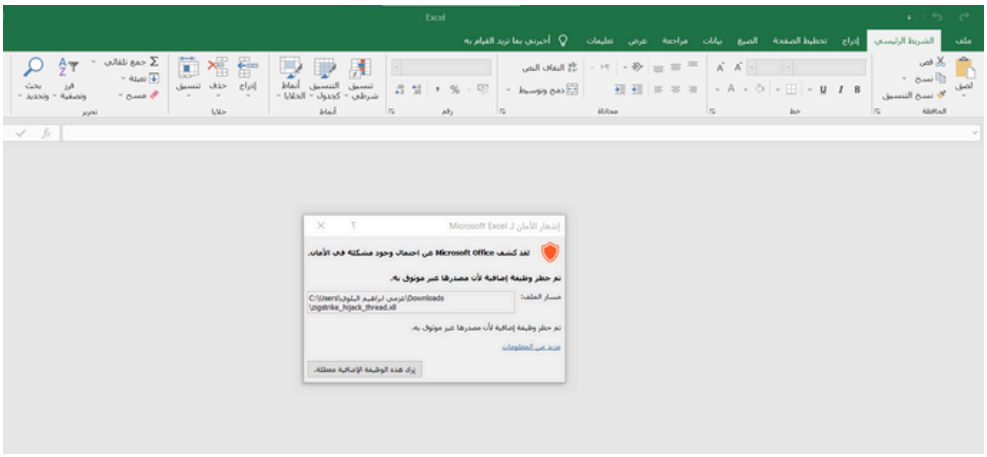

Figure 5.3: Microsoft Excel blocking the XLL file zigstrike_hijack_thread.xll due to being from an untrusted source.

Figure 5.4: Microsoft Defender for Endpoint preventing execution of the RAR file byshell.rar, detected as Sabsik malware.

6. Technique 4 – Native Messaging API via Covert C2

6.1 Overview

This method leverages the Native Messaging API — a legitimate browser feature designed to allow communication between a browser extension and a native desktop application. In this technique, the API was weaponized to establish stealthy post-exploitation communication via a custom C2 framework named Covert C2 (C3).
Unlike traditional browser-based attacks that steal cookies or redirect users, this method allows direct communication with the OS, enabling command execution and in-memory payload loading via a registered native app.
The method stands out due to its:

- Lack of external C2 connection during initial execution (fully local)
- Compatibility with Chrome and Edge (Native Messaging is supported)
- Zero detection by the tested EDRs, despite minimal evasion

This proof-of-concept (PoC) was inspired by EarthKitsune APT's 2023 activity but introduces direct OS-level interaction.

3.2 Step-by-Step

- Register Native Messaging Host:
  A manifest file is created and registered in the Windows Registry to link the browser extension with a native app (usually located in AppData).
- Develop Browser Extension:
  The extension is responsible for sending commands or encrypted payloads in JSON format to the native messaging host.
- Implement Native App (C++):
  The native app reads JSON input from stdin, decrypts the payload (e.g., XOR/Base64), and prepares it for execution.
- In-Memory Execution:
  The shellcode is executed directly from memory using low-level system calls (VirtualAlloc, NtWriteVirtualMemory, NtCreateThreadEx).
- Communication Loop:
  The extension continuously sends tasks, and the native app responds or executes, forming a local covert channel.
- No Network Artifacts:
  Since communication is purely local (extension ↔ native app), no network traffic is generated, making it highly evasive.

Figure 6.1 shows the custom extension developed for Microsoft Edge, configured to communicate with the native app via manifest.json.

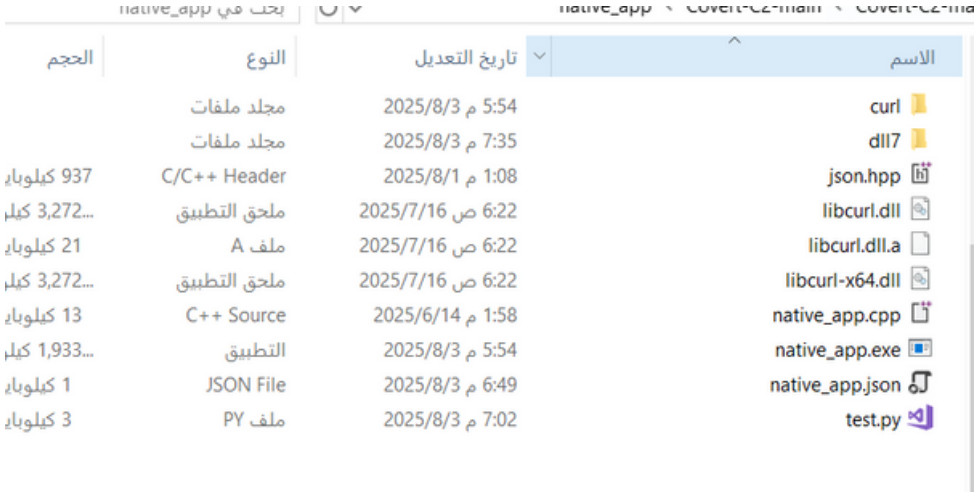

Figure 6.2 displays the Native App that acts as a communication bridge, executing received commands and sending results back to the extension.

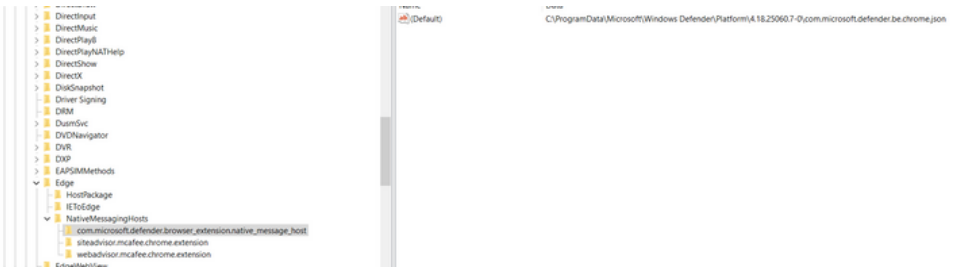

Figure 6.3 shows the registry key configuration used to register the native app with the browser, ensuring persistence and proper linking with the extension.

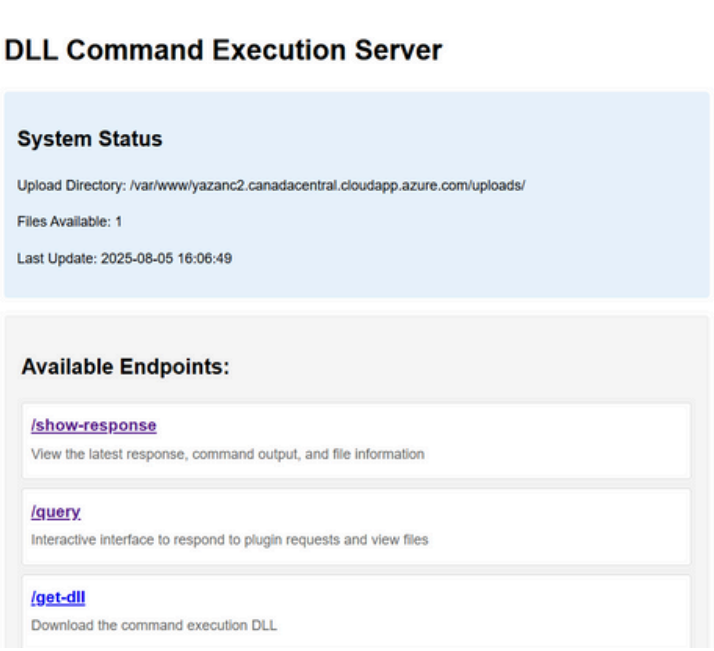

Figure 6.4 presents the webserver interface, which is used to send commands and receive outputs via HTTP/3 with QUIC, providing encrypted traffic between attacker and victim.

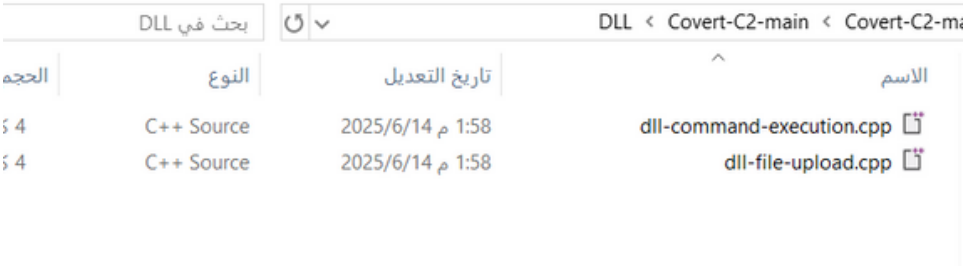

Figure 6.5 shows the signed DLL file used for evasion purposes inside the extension folder to bypass unsigned code restrictions.

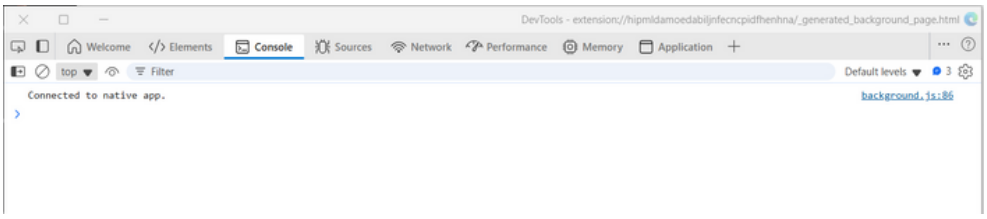

Figure 6.6 shows the Edge background activity confirming the extension runs silently and is actively communicating with the native app.

6.3 Results and Observations

During the test, the Native Messaging API successfully enabled bi-directional communication between the browser extension and the native executable, executing system-level commands without triggering any alerts. Notably, Microsoft Defender for Endpoint—including both its EDR component and cloud-based AI/ML protection—did not generate any warnings, detections, or telemetry flags during the entire communication and execution process.

This indicates a complete bypass of Microsoft EDR for this particular method, showcasing a critical blind spot in the platform's behavioral and signature-based analysis.

However, despite the stealthiness, this technique suffers from major limitations in real-world scenarios. It requires pre-installation steps, including manual setup of the native app, registry modifications, and loading the extension in developer mode. These prerequisites make it impractical and less effective as a post-exploitation technique, since an attacker would rarely have the opportunity to install and configure such components directly on the target without prior access or user interaction.

As such, while the method remains stealthy under current detection systems, it lacks the flexibility and feasibility required for scalable or automated post-exploitation.

Figure 5.3.1: Successful Command Execution via Covert C2 Web Interface

7. Comparative Analysis

This section compares the four tested techniques in terms of stealth, detection by Microsoft EDR and cloud-based AI/ML protections, setup complexity, and practical feasibility. Each technique leverages a different method of payload delivery or execution, which resulted in varying outcomes when tested in a controlled environment against Microsoft's modern defenses.

The table below summarizes the observed characteristics of each approach:

Technique	Detected by Microsoft EDR?	Shellcode Execution?	Cloud Evasion?	Notes
Early Cryo Bird	❌ Not Detected	✅	Detected ✅	Very stealthy, runs in memory
Direct Syscalls	❌ Not Detected	✅	Detected ✅	Requires syscall mapping
XLL Macro	❌ Not Detected ✅ Detect by microsoft office	✅	❌ Not Detected	Detected as suspicious document
dll inj	Detected ✅	✅	Detected ✅	Detected via install
Native Messaging API	❌ Not Detected	(via commands)	❌ Not Detected	Requires browser extension setup

8. Conclusion

This research explored five advanced techniques to bypass Microsoft Defender for Endpoint (EDR) and its cloud-based AI/ML protection. The evaluation covered various evasion strategies ranging from low-level memory injection to browser-based covert channels. The results highlight both the strengths and blind spots of modern endpoint protection systems:

Early Cryo Bird Shellcode Injection bypassed Microsoft Defender EDR, but it was detected by Cloud Protection due to machine learning heuristics that flagged anomalous memory behavior.

Direct Syscalls with Encrypted Shellcode utilized low-level NT system calls (e.g., NtAllocateVirtualMemory, NtWriteVirtualMemory, NtCreateThreadEx) through SysWhispers2, allowing execution of XOR-encrypted shellcode directly in memory. This method successfully bypassed both EDR and Cloud Protection, as it avoided user-mode hooks, IAT scanning, and common Win32 APIs—highlighting the effectiveness of direct syscall injection in evading behavioral detection.

Malicious XLL Macro evaded both EDR and Cloud AI; however, upon execution, Microsoft Office blocked the file due to its built-in protection mechanisms for untrusted add-ins. Thus, while technically undetected, it is practically neutralized by Office's trust center settings.

DLL Injection via Reflective Loading was immediately flagged by Microsoft Defender EDR, indicating that such techniques are now well-covered by static and behavioral signature rules and are no longer stealthy in default environments.

Native Messaging API via Covert C2 allowed stealthy, bi-directional communication between a browser extension and a native executable on the host, without triggering any alerts from either EDR or Cloud-based protection. Despite its stealth, the technique requires considerable preparation and installation steps (e.g., extension deployment, native app setup, registry configuration), which limits its practicality in fast-paced attack scenarios.

Key Takeaways:

EDR can be bypassed using advanced in-memory or system-level evasion, but Cloud AI detection adds a robust second layer that catches unusual runtime behaviors.
Direct syscalls and encrypted payloads remain effective against most detection mechanisms.
Techniques that leverage trusted software functionality (e.g., Native Messaging API, XLL macros) offer stealth but may face usability constraints.
Static injection methods like DLL injection are increasingly obsolete without additional obfuscation or layering.
Ultimately, red teams and attackers must weigh stealth versus operational complexity, while defenders should enhance detection for non-standard execution paths, native system call abuse, and legitimate application misuse.

7. References

- **Early Cryo Bird Injections – PoC Implementation**
  https://github.com/zero2504/Early-Cryo-Bird-Injections
- **Covert C2 – Native Messaging API Post-Exploitation Framework**
  https://github.com/efchatz/Covert-C2
- **ZigStrike Shellcode Loader – Encrypted Shellcode with Evasion**
  https://github.com/gavz/ZigStrike_shellcode
- **Bypassing Windows Defender in 2025 – Direct Syscalls & XOR Encryption**
  https://www.hackmosphere.fr/en/bypass-windows-defender-antivirus-in-2025-evasion-techniques-using-direct-syscalls-and-xor-encryption-part-2
- **SysWhispers2 – Direct System Calls for Red Teaming**
  https://github.com/jthuraisamy/SysWhispers2
- **Reflective DLL Injection by Stephen Fewer**
  https://github.com/stephenfewer/ReflectiveDLLInjection