

Behind the Refund: From GST Phishing to Remcos RAT Through a Multi-Stage .NET Infection Chain

 seqrite.com/blog/behind-the-refund-from-gst-phishing-to-remcos-rat-through-a-multi-stage-net-infection-chain

Vaibhav Billade

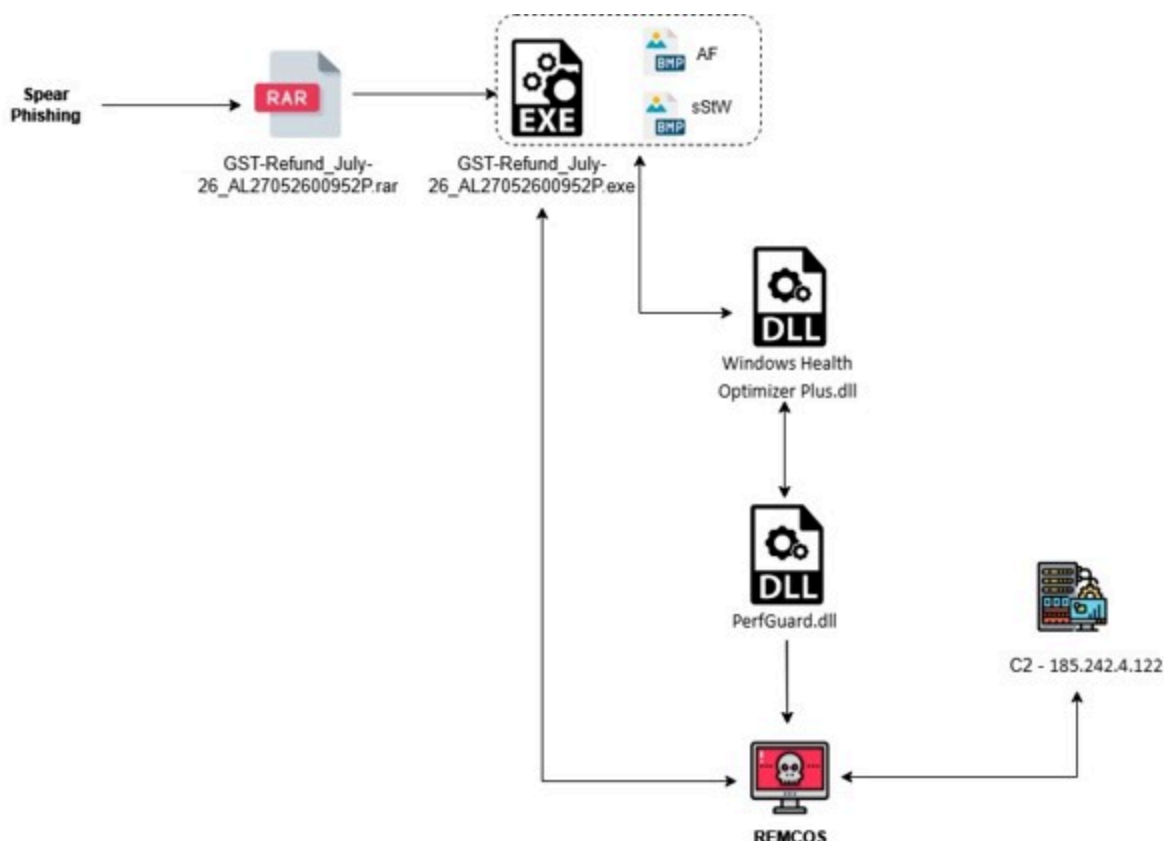
July 17, 2026

[Home](#) / [Technical](#) / Behind the Refund: From GST Phishing to Remcos RAT Through a Multi-Stage .NET Infection Chain

Introduction

Seqrite Labs recently identified a malware distribution campaign that abused the credibility of government institutions to increase infection success rates. The threat actors impersonated legitimate government departments and distributed malicious emails disguised as official notifications related to taxation, refunds, compliance requirements, and regulatory matters. By leveraging recognizable government branding, urgency, and financial incentives, the campaign was designed to persuade recipients to open malicious attachments or interact with embedded content. This activity highlights the continued effectiveness of government-themed social engineering techniques in facilitating malware delivery and compromising targeted users.

Infection Chain:



Initial Findings about Campaign:

A sophisticated phishing campaign targeting Indian businesses and taxpayers has been observed leveraging GST-related themes to distribute malware. First identified on VirusTotal on July 06, 2026, the campaign impersonates legitimate government communications and uses tax-related lures to increase user interaction. The threat actors employ convincing documents and filenames that closely resemble official GST notifications, making it difficult for recipients to distinguish malicious content from legitimate government correspondence. The primary objective of the campaign appears to be malware deployment and the theft of sensitive information, highlighting the continued abuse of government-themed social engineering techniques to target the Indian financial and taxation ecosystem.

Initial mail:

In the analyzed case, the threat actor distributed a phishing email that spoofed the Government of India GST (Goods and Services Tax) infrastructure. The email originated from the address donotreply.@gst.gov.in and used the subject line "Refund Application GST RFD-01 filed vide ARN AL27052600952P". The message was crafted to appear as an official GST refund notification, attempting to convince recipients that a refund application had been

processed or required review. To enhance credibility, the email included an archive attachment named GST-Refund_July-26_AL27052600952P.rar, which referenced the same Application Reference Number (ARN) contained in the subject line. The use of matching identifiers across the email content and attachment is a common social engineering tactic designed to reinforce legitimacy and reduce suspicion.

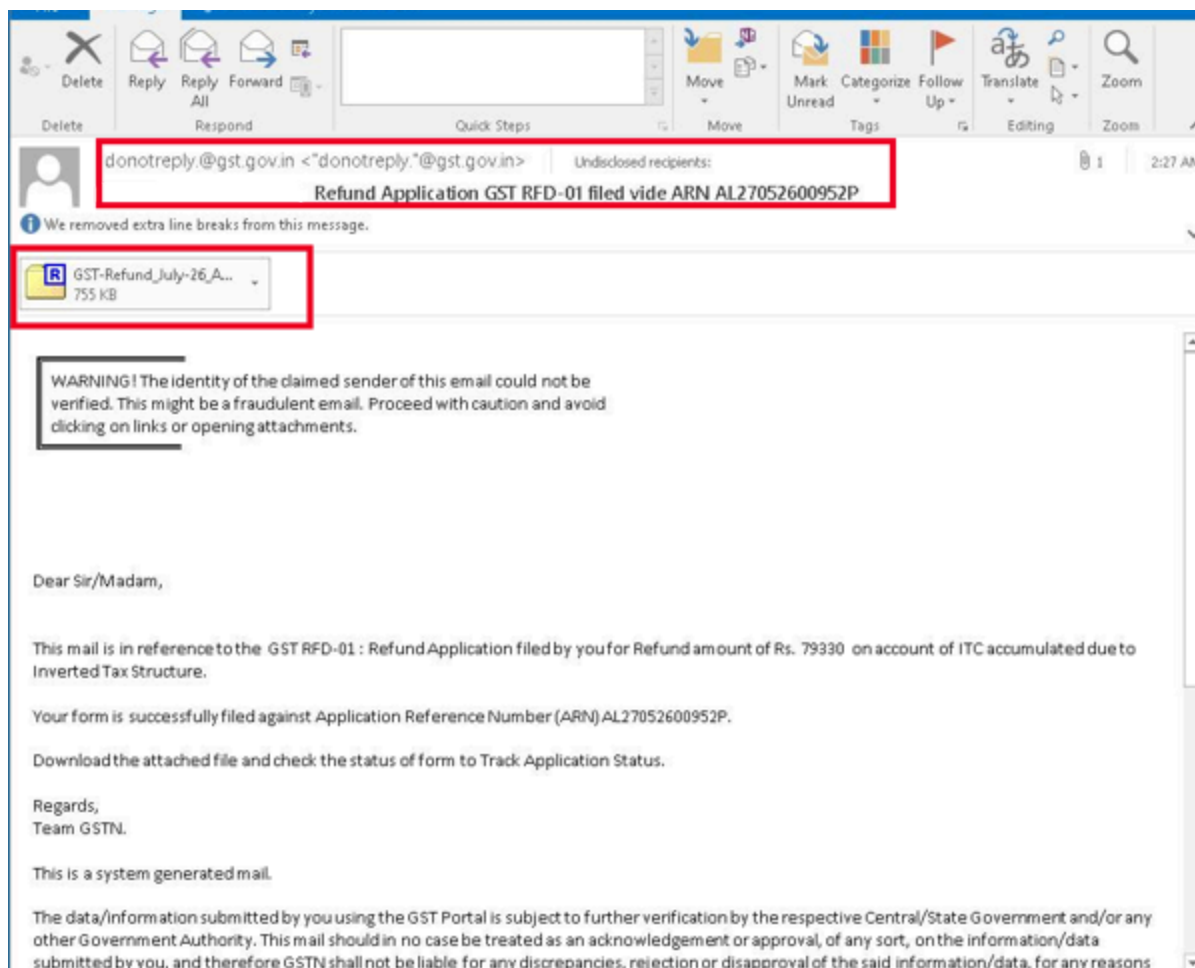


Fig: Email

Technical Analysis:

Upon extraction, the archive contained an executable file masquerading as a GST refund-related document or application. Initial examination revealed that the payload is a .NET-based executable, indicating that it was developed using the Microsoft .NET framework. Threat actors frequently utilize .NET binaries because they are relatively easy to develop, support rapid deployment of malicious functionality, and often include obfuscation mechanisms that complicate static analysis. The executable serves as the primary malicious payload delivered through the phishing campaign and represents the next stage of the attack chain following successful user interaction with the attachment.

Name	Date modified	Type	Size
GST-Refund_July-26_AL27052600952P.com	7/9/2026 4:52 AM	MS-DOS Applicati...	799 KB

Fig: Extracted Zip

Analysis of the extracted executable identified it as a Microsoft .NET Framework-based binary. Unlike native Windows Portable Executable (PE) files compiled directly into machine code, .NET applications contain Intermediate Language code that is executed by the Common Language Runtime.

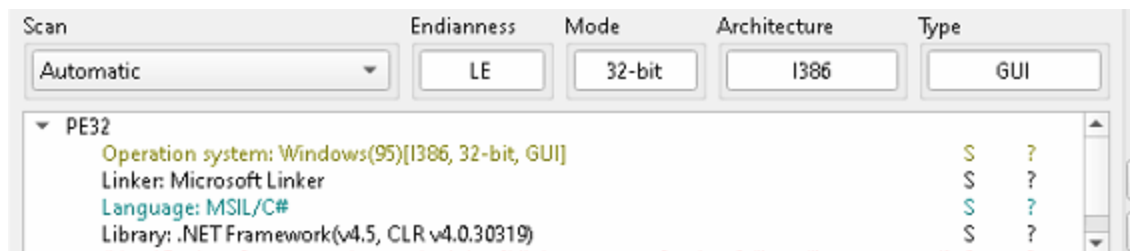


Fig: .NET initial Loader

The analysis begins with a .NET binary that contains a bitmap (AF) embedded in its resources. Rather than storing the next-stage payload as a normal DLL resource, the author hides it inside the image by encoding the payload bytes into the RGB values of the bitmap's pixels.

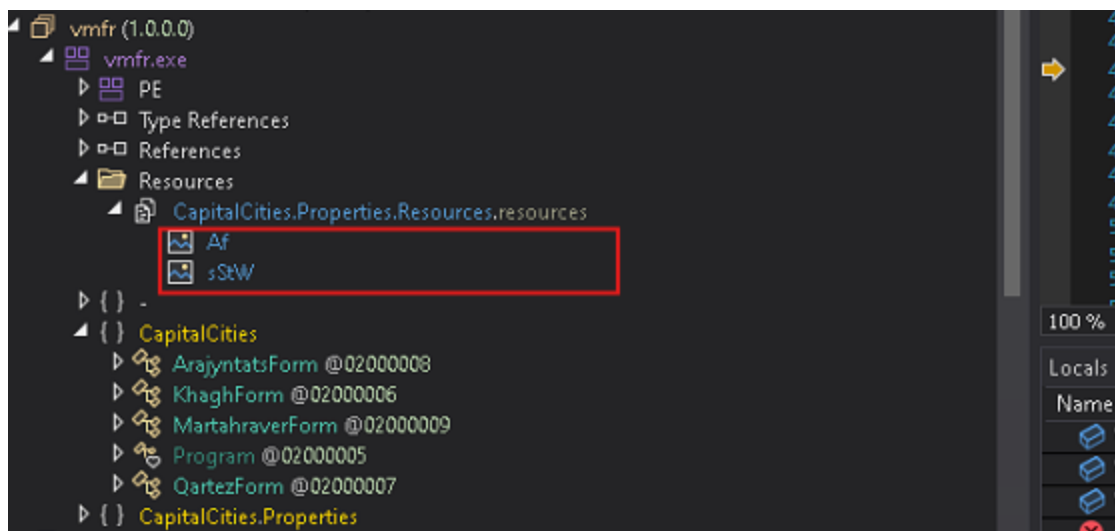


Fig: Encrypted Bitmap containing Next stage loaders

When the application starts, it calls the ServiceChromatic() function, passing the embedded bitmap as input. Although the function contains a large amount of obfuscating junk code, its actual purpose is straightforward: it iterates through the bitmap in column-major order, reads the Red, Green, and Blue values from each pixel, and reconstructs the original byte stream. The function extracts the first 64,000 bytes from the image and returns them as a byte[].

The recovered byte array is then passed directly to `Assembly.Load()`, causing the hidden .NET assembly to be loaded entirely from memory without being written to disk. In this sample, the extracted payload is the next-stage assembly, `Windows Health Optimizer Plus.dll`, which is subsequently executed via reflection.

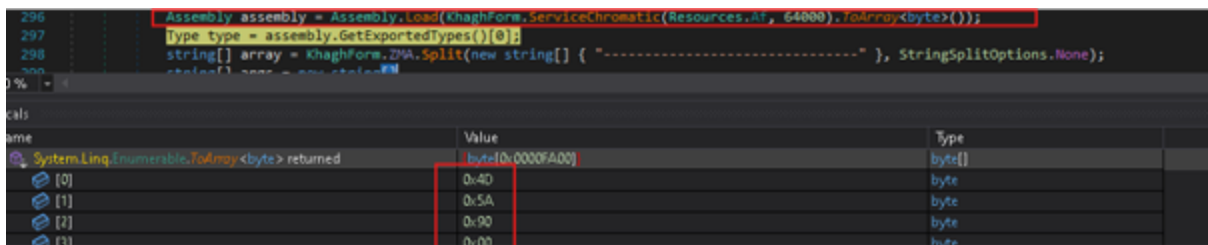


Fig: Decrypted Loader dll

Second-Stage loader Analysis:

Following the successful extraction and in-memory loading of the second-stage loader, **Windows Health Optimizer Plus.dll**, the malware invokes the method `v3.aq()` with three parameters: `v3.aq("73537457", "6C6E6A", "CapitalCities")`. The first two arguments are hexadecimal-encoded strings. When converted from hexadecimal to ASCII, `73537457` resolves to `"sStW"`, and `6C6E6A` resolves to `"inj"`. These decoded strings are subsequently used by the second-stage assembly as command or operation identifiers. Their values indicate that the malware instructs the next stage to process the embedded bitmap payload (`sStW`) and perform an injection-related operation (`inj`), suggesting that these parameters serve as execution instructions controlling the behavior of the next-stage payload.

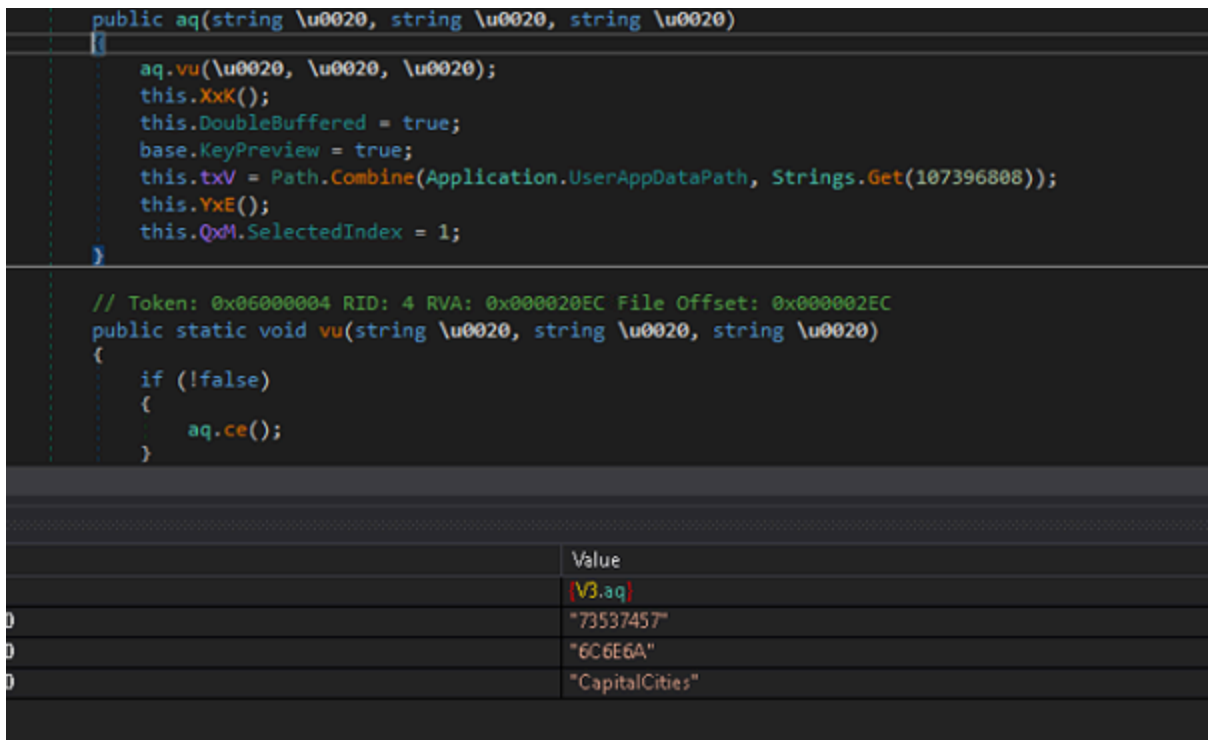


Fig: Arguments passed on second stage loader

Bitmap-Based Payload Extraction:

It then employs a bitmap-based payload extraction technique by abusing pixel storage to conceal embedded data. Rather than using the image solely for visual content, it reads the raw ARGB values of individual pixels and reconstructs the hidden binary payload from the extracted byte stream. The bitmap effectively acts as a carrier for the concealed data, allowing the payload to remain hidden within an otherwise legitimate image format.

The extracted payload is processed through a custom XOR-based decryption routine, where the supplied string parameter is converted into UTF-16 Big Endian bytes and used as XOR key material along with an additional value derived from the encrypted buffer. After decryption, the payload is reconstructed directly in memory, revealing `perfgurd.dll`. The malware loads the DLL from its encoded representation without writing it to disk, enabling a fileless execution chain that reduces detection opportunities.

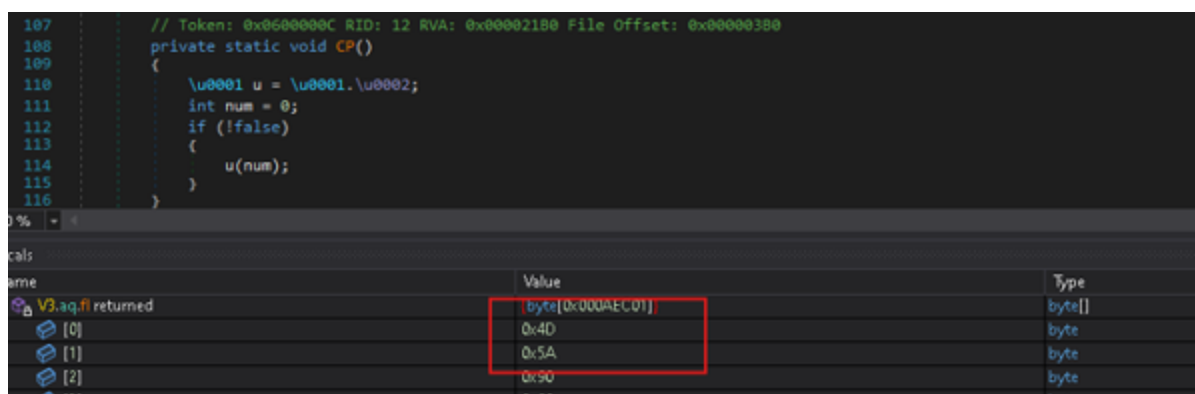


Fig: Third-Stage Loader

Third-Stage Loader Analysis – `perfgurd.dll`

The method below acts as a dynamic execution wrapper that invokes a specific method from a loaded assembly without directly referencing its name. It first initializes the variable `num` to 29, but if the process is running in a 64-bit environment (`IntPtr.Size == 8`), it changes the value to 15. This conditional selection suggests that the assembly contains separate methods for 32-bit and 64-bit execution paths.

The parameter `object_0` is expected to be a loaded .NET Assembly object. The code retrieves all types within the assembly using `GetTypes()`, selects the type at index 20, and then retrieves its methods using `GetMethods()`. Depending on the system architecture, it selects either method 29 or method 15 from that type and invokes it using reflection via `Invoke(null, null)`.

Because both arguments to `Invoke()` are null, the targeted method is expected to be a static method that accepts no parameters. This technique allows the malware to hide the actual

method name and execution flow from static analysis tools, as execution is based entirely on array indexes rather than explicit identifiers. In practice, this function serves as a dispatcher that transfers execution to the next stage of the malware, with the selected method likely containing the primary malicious functionality or payload initialization routine.

```
private static void (object \u0020)
{
    int num2;
    int num = (num2 = 29);
    int num3;
    if (num != 0)
    {
        if (!false)
        {
            num3 = num;
        }
        if (false)
        {
            return;
        }
        if (\u0002.\u0003() != 8)
        {
            goto IL_0022;
        }
        if (6 == 0)
        {
            return;
        }
        num2 = 15;
    }
    if (8 != 0)
    {
        num3 = num2;
    }
    IL_0022:
    \u0005.\u0006(\u0004.\u0005(\u0003.\u0004((Assembly)\u0020)[20])[num3], null, null);
}
```

*((Assembly)object_0).GetTypes()
[20].GetMethod()[num].Invoke(null, null);*

Fig: Architecture based call

Based on the architecture of the victim system, the malware invokes one of two methods from the PerfGuard.Networking.EfficientServer class within perfgurd.dll. For 32-bit systems, the method at index 29, WaitForAdvancedController(), is executed, while on 64-bit systems, the method at index 15, WaitForGlobalAggregator(), is invoked. As shown below, the malware uses method indexes rather than explicit method names during execution, a technique that helps obscure the execution flow and complicates static analysis.

```

Assembly:
PerfGuard, Version=10.8.5.0, Culture=neutral, PublicKeyToken=null

Type[20]:
PerfGuard.Networking.EfficientServer

=====
Method index: 29
=====
Name      : WaitForAdvancedController
Declaring : PerfGuard.Networking.EfficientServer
Return    : System.Void
Token     : 0x60000F9

=====
Method index: 15
=====
Name      : WaitForGlobalAggregator
Declaring : PerfGuard.Networking.EfficientServer
Return    : System.Double
Token     : 0x600014B

```

Fig: invoked methods based on architecture

The loader first checks whether the process is running with administrative privileges. If elevated access is not available, it leverages cmstp.exe, a legitimate Windows utility, to spawn the main process with elevated privileges. By abusing this trusted system binary, the malware attempts to bypass normal execution restrictions and continue execution under an administrative context.

To establish persistence, the malware creates a copy of itself at C:\Users\admin\AppData\Roaming with the filename RVKLmPcYcNO.exe, as shown in the figure below. It then generates a PowerShell script (.ps1) in the temporary directory, which is responsible for launching the copied executable in hidden mode, thereby reducing its visibility to the user. To ensure automatic execution after user login, the malware creates a Run registry entry under HKCU\Software\Microsoft\Windows\CurrentVersion\Run that points to the PowerShell script. As a result, the PowerShell script is executed whenever the user logs in, which in turn launches the copied malware executable and maintains persistence across system reboots.

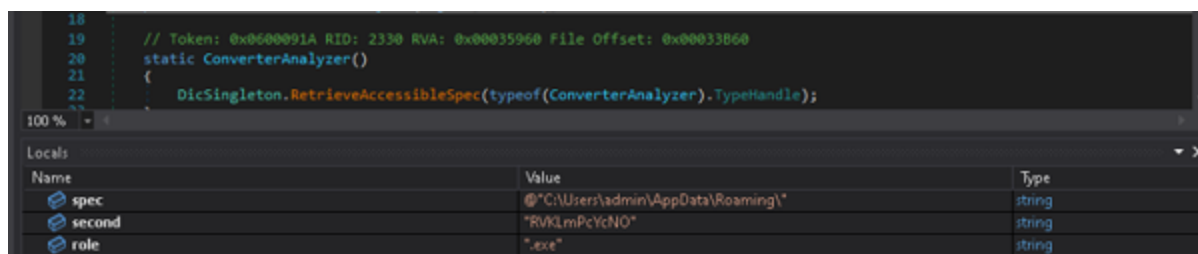


Fig: Copy of Executable in AppData folder

```

# module_load
$rlqg = [System.Guid]::NewGuid().ToString()
$efene = [char]78+[char]111+[char]80+[char]114+[char]111+[char]102+[char]105+[char]108+[char]101
$rlqg = [System.Guid]::NewGuid().ToString()
$ajv4l = [char]69+[char]120+[char]101+[char]99+[char]117+[char]116+[char]105+[char]111+[char]110+[char]80+[char]111+[char]108+[char]105+[ch
$wbsi3 = [char]66+[char]121+[char]112+[char]97+[char]115+[char]115
$dlrip = (1..3 | ForEach-Object { $_ * 0 })
$oung5 = [char]87+[char]105+[char]110+[char]100+[char]111+[char]119+[char]83+[char]116+[char]121+[char]108+[char]101
$gwek0 = [char]72+[char]105+[char]100+[char]100+[char]101+[char]110
$03ez5 = [char]83+[char]116+[char]97+[char]114+[char]116+[char]45+[char]80+[char]114+[char]111+[char]99+[char]101+[char]115+[char]115
$rlqg = [System.Guid]::NewGuid().ToString()
$xxfzs = @(67,58,92,85,115,101,114,115,92,97,100,109,105,110,92,65,112,112,68,97,116,97,92,82,111,97,109,105,110,103,92,82,86,75,76,109,80,
$1almq = ($xxfzs | ForEach-Object { [char]$_ }) -join ''
$pkbip = Get-Date
4 $03ez5 -FilePath $1almq

```

Fig: Ps1 used for persistence

The third-stage loader decrypts and reconstructs the final payload directly within its own process memory. After decryption, the recovered payload is loaded and executed from the existing process context, avoiding the need to launch a separate executable process. This in-memory execution technique helps reduce filesystem artifacts and improves the stealth of the payload execution. The decrypted payload extracted during analysis is shown below.

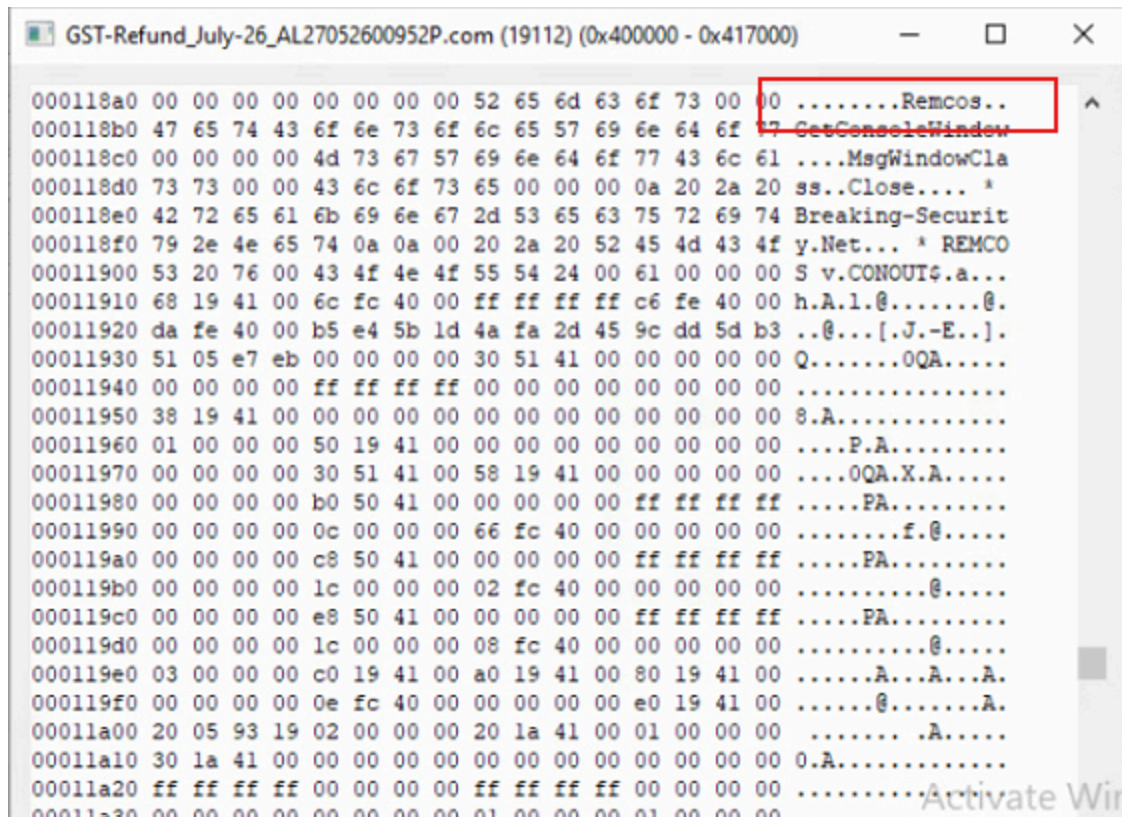
```

10 // Token: 0x06000A00 RID: 2560 RVA: 0x00035F28 File Offset: 0x00034128
11 public static void ExamineAttachedWatcher(string i, byte[] cont, RunnerDecorator A_2)
12 {
13     A_2(i, cont);
14 }
15
16 // Token: 0x06000A01 RID: 2561

```

Name	Value	Type
cont	byte[0:00017000]	byte[]
[0]	0x40	byte
[1]	0x5A	byte
[2]	0x90	byte
[3]	0x00	byte
[4]	0x03	byte
[5]	0x00	byte

Final Payload Deployment – Remcos RAT



Analysis of the final stage revealed that the malware ultimately deploys Remcos RAT (Remote Access Trojan). Remcos is a commercially available remote administration tool that is frequently abused by threat actors to establish persistent access to compromised systems.

The multi-stage loader architecture observed in this campaign is designed to conceal the final payload and complicate detection. The initial executable, Windows Health Optimizer Plus.dll, and perfgurd.dll collectively function as staged loaders that decrypt, reconstruct, and execute Remcos entirely in memory.

Once executed, Remcos provides the attacker with extensive control over the compromised host.

Remcos Capabilities:

The analyzed sample provides capabilities commonly associated with Remcos RAT, including:

- Remote command execution.
- File upload and download.
- File system management.
- Keylogging and keystroke monitoring.

- Credential harvesting.
- System reconnaissance.
- Process and service manipulation.
- Registry modification.
- Persistence establishment.
- Screenshot capture.
- Remote desktop functionality.
- Monitoring of user activity.
- Collection of system and network information.
- Communication with attacker-controlled Command-and-Control (C2) infrastructure.
- Deployment of additional payloads.

These capabilities enable threat actors to maintain long-term access, monitor victim activity, steal sensitive information, and conduct further post-compromise operations.

Conclusion:

This campaign represents a sophisticated GST-themed phishing operation designed to deliver Remcos RAT through a multi-stage .NET malware framework. The attack begins with a fraudulent GST refund notification that leverages Government of India branding and tax-related themes to increase user trust and encourage execution of a malicious attachment.

Infrastructure analysis identified multiple randomized subdomains under *aofmokighoig.hath.network* resolving to infrastructure associated with Remcos RAT command-and-control activity. The use of dynamic DNS services, including *hath.network* and *synology.me*, together with rotating infrastructure hosted by M247 Europe SRL, is consistent with commodity malware operations and financially motivated cybercrime. The GST refund phishing further suggests deliberate targeting of Indian users and organizations. While the available indicators support attribution to a persistent cybercrime campaign cluster, the current evidence is insufficient to confidently associate the activity with any publicly tracked threat actor. Based on the observed infrastructure, malware characteristics, and targeting, the campaign is assessed to be operated by an India-focused Remcos operator or an Initial Access Broker utilizing disposable dynamic DNS infrastructure.

Overall, the combination of phishing-based initial access, staged .NET loaders, bitmap-based payload concealment, fileless execution, and resilient command-and-control infrastructure demonstrates a deliberate effort to evade security controls while establishing persistent, unauthorized access to victim systems. The campaign reflects the continued evolution of commodity malware delivery techniques and highlights the effectiveness of layered infection chains in enabling credential theft, remote access, and broader post-compromise operations.

Seqrte Coverage:

Trojan.Agent

Trojan.AgentCiR

Trojan.LoaderCiR

Trojan.Mauvaise.SL1

IOCs:

3757dccb2adae65ccdf8d5e5c948b927	Eml
07d7d21c2c0920d198efb9ea54900a80	GST-Refund_July-26_AL27052600952P.rar
20476F3A51DFDDF3DC0603FC7858D894	GST-Refund_July-26_AL27052600952P.com
7842D12D9E37C75076133BE5B9904CB2	Windows Health Optimizer Plus.dll
CC34D9760394104AD47877A0D57E9C63	PerfGuard.dll
2a34bdd25b404737ee5d3b52bf0b3b70	RemcosRAT

C2:

185.242.4.122

MITRE ATT&CK:

Tactic	Techniques Mapped
Initial Access	T1566.001
Execution	T1204.002, T1059.001, T1129
Persistence	T1547.001, T1036
Privilege Escalation	T1548.002

Defense Evasion	T1027, T1027.013, T1140, T1218.003, T1620, T1055
Credential Access	T1056.001, T1555
Discovery	T1082, T1016, T1057, T1083
Collection	T1113, T1005, T1039
Command and Control	T1071.001, T1105
Lateral Movement	T1021
Impact	T1489

Authors:

Vaibhav Billade

Rumana Siddiqui

Related Posts

