# njRAT Malware Analysis

**Malware Analyzed**: njRAT Remote Access Trojan  
**SHA256**: c582b5864a67c2d63f8d3a8faf08b47e94646a96cd81abe507d8d08df13e40c4  
**File Size**: 2.32 MB  
**Environment**: Windows 7 Virtual Machine (VM) with tools including PEiD, Ghidra, Wireshark, INetSim, RegShot, ProcMon, x64dbg

# Unpacking the Malware

**Tool Used**: PEiD v0.95

Before diving into analysis, I first wanted to confirm whether this malware was packed or obfuscated in any way. Packing is pretty common in malware to make static analysis harder. I used PEiD, which is a lightweight tool that can detect common packers like UPX, or custom wrappers.

Once loaded, PEiD didn’t show any packer signature — it said "Nothing found [Overlay]\*". That already suggested it might be unpacked. I double-checked the section info:

* **Entry Point (EP)**: 0x0002800A
* **EP Section**: .text (which is a good sign)
* **File Offset**: 0x0002740A
* **First Bytes**: E8 C8 D0 00 — these looked like x86 instructions instead of shellcode.

So based on all this, I figured it’s safe to assume that the malware is not packed and I can move ahead with static analysis directly in Ghidra. This saves a lot of time, especially since unpacking manually is a pain.

The malware doesn’t appear to be packed or obfuscated. No signs of runtime unpacking either. It’s ready for static analysis in Ghidra and other tools

A screenshot of a computer

AI-generated content may be incorrect.

*Figure 1: PEiD confirms that no packer is detected and the entry point lies in the .text section.*

# String Analysis

**Tool Used**: Ghidra (Defined Strings – Loaded Blocks Only)

After confirming that the malware wasn’t packed, I moved on to looking at the strings using Ghidra. I filtered the view to only show loaded blocks, since those are the parts of the binary that are actually mapped into memory and executed.

* *kernel32.dll, GetNativeSystemInfo* – This tells me the malware is using Windows system APIs, probably to get environment details like architecture or OS version. Could be for compatibility or anti-analysis.
* *AutoIt v3, CMDLINE, /AutoIt3ExecuteScript* – These are a giveaway. This malware was very likely built or packed using AutoIt, which is a scripting language that’s often used to hide real behavior.
* *FILEWRITE, FILEDELETE, FILEREADLINE* – These show the malware is messing with the file system. It could be writing logs, dropping payloads, or storing stolen data.
* *TCPCONNECT, TCPSEND, TCPSTARTUP* – These strings suggest that this malware can establish a TCP connection and send/receive data — probably talking to a command-and-control (C2) server.
* *REGWRITE, REGDELETE, REGREAD* – These are registry operations, which makes me think it might be setting up persistence or modifying system settings to stay hidden.
* *SHELLEXECUTE, RUNWAIT* – These are used to launch other programs or scripts. So it might be executing external tools or scripts during runtime.

There were also tons of AutoIt-related strings and GUI stuff like WinDetectHiddenText and MouseClickDelay, which backs up the AutoIt origin theory.

Even without running the malware, I got a good sense of what it might do just from the strings. It’s using AutoIt, can connect to a server, and messes with files and registry keys. These results helped me figure out what to look for in the next steps — like imports and network traffic.

A computer screen shot of a computer

AI-generated content may be incorrect.

*Figure 2: Defined strings extracted from loaded memory blocks in Ghidra.*

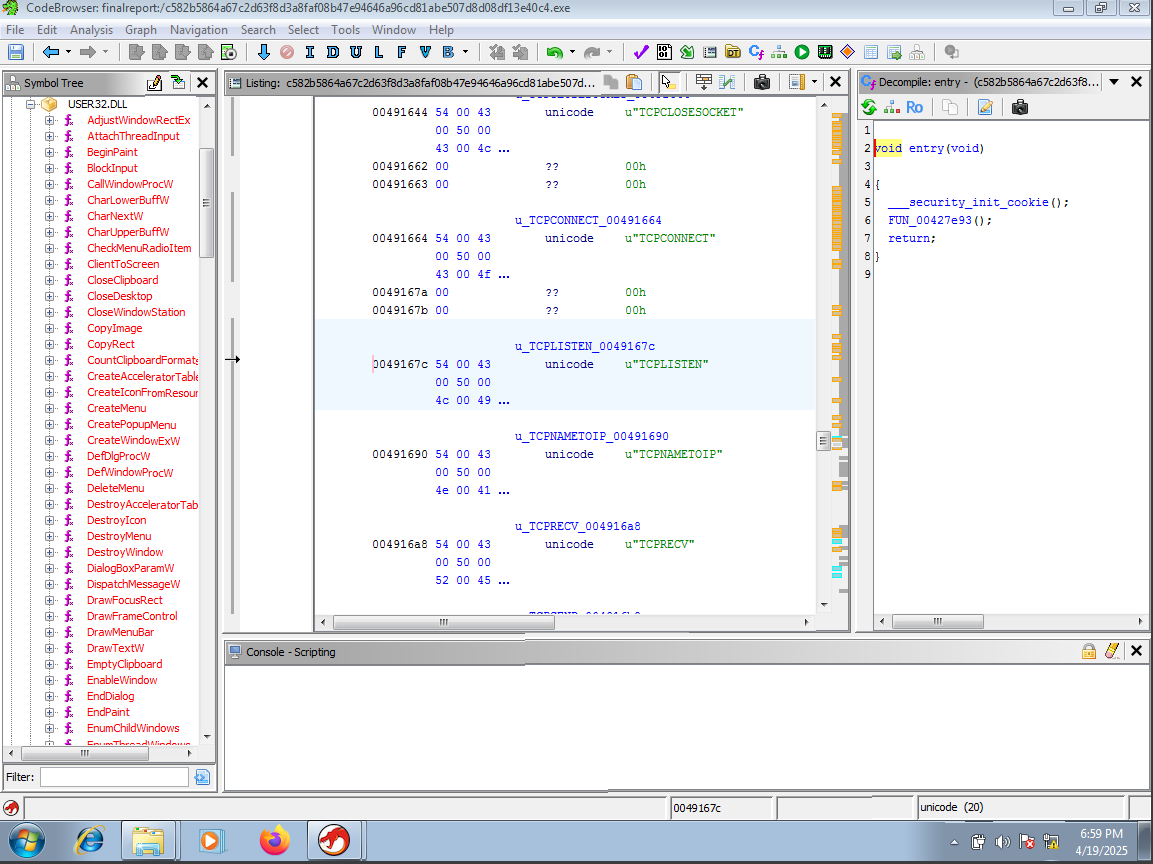
# Imported Methods

**Tool Used**: Ghidra (Imports View)

To get a better idea of what the malware is doing under the hood, I checked out the list of imported functions using Ghidra. These are the Windows API calls that the malware relies on to do things like access files, talk to the internet, or change system settings.

* From*KERNEL32.DLL*:  
  Functions like CreateFileW, WriteFile, and DeleteFileW point to file system activity. It could be used for logging or dropping payloads. CreateProcessW shows that it might be launching new processes (maybe even itself), and OpenProcess hints at possible process injection or spying. Sleep also shows up, which malware often uses to avoid sandbox detection by delaying execution.
* From*ADVAPI32.DLL:*  
  This stood out because of registry functions like RegCreateKeyExW, RegSetValueExW, and RegDeleteValueW. These are commonly used for persistence. There were also functions like LogonUserW and OpenProcessToken, which could mean the malware tries to run with higher privileges.
* *From USER32.DLL:*  
  GetAsyncKeyState is often used in keyloggers to record user input. Other functions like GetForegroundWindow and ShowWindow made me think it might be hiding itself or messing with window visibility.
* *From WININET.DLL:*  
  This tells me the malware uses HTTP to connect out to the internet. It imports stuff like InternetOpenW, HttpOpenRequestW, and InternetReadFile, so it could be reaching out to a C2 server, downloading files, or sending stolen data.
* *From WSOCK32.DLL:*  
  These are raw socket functions like socket, connect, recv, and send. So it looks like the malware has two different ways to do network communication: the WININET API for HTTP and sockets for more direct TCP stuff.

Based on the imports, this malware definitely checks all the boxes for RAT behavior. It interacts with the registry, filesystem, UI, and network — and it’s got keylogger APIs too.



*Figure 3: Imported Windows API functions visible in Ghidra.*

|  |  |  |  |
| --- | --- | --- | --- |
| API Function | Found in Strings | Found in Imports | Interpretation |
| GetAsyncKeyState | Yes | Yes | Used for keylogging (monitors keyboard input) |
| CreateFileW | Yes | Yes | Likely used for writing/reading log or config files |
| RegSetValueExW | Yes | Yes | Used to modify registry keys (possibly for persistence) |
| InternetReadFile | Yes | Yes | Reads HTTP responses — suggests C2 communication |
| connect / send | Yes | Yes | Establishes direct TCP connection to remote host |

# Decompiled Function Analysis

**Tool Used**: Ghidra

For this section, I picked one of the largest and most suspicious functions in the binary, named FUN\_0040a000. Its size and early position in the code made it stand out, and once I opened it, I could immediately tell this was a key part of the malware’s behavior. Based on what I observed, this function works as a command dispatcher, basically the core logic that decides what the malware should do depending on what instruction it receives.

**Key Behavior Inside FUN\_0040a000:**

* Handles different “commands” using a switch-case structure (e.g., case 0x27, case 0x2A, etc.)
* Manages different types of data using VARIANTARG pointers, which are common in COM/AutoIt-based scripting (consistent with the AutoIt strings I found earlier)
* Calls Windows API functions like CreateProcessW, RegSetValueExW, ReadFile, and WriteFile
* Also contains cleanup and memory-handling logic (VariantClear, memory frees, etc.), suggesting it’s been written to avoid crashing or leaving traces behind

This makes sense for a Remote Access Trojan (RAT) like njRAT. It’s built to receive instructions from a command-and-control (C2) server, and this function looks like the part that processes those instructions and routes them to the appropriate internal subroutine.

**Decompiled Code**

Below is a trimmed and commented version of the logic from FUN\_0040a000. I renamed some of the variables and added comments to make it easier to understand.

if ((global\_flag & 1) == 0) {

// First-time init block

global\_flag |= 1;

dispatcher\_context = 0;

command\_tracker = 1;

initializeDispatcher(&callback\_table);

}

// Prepare a variant for command input

paramTracker = param3;

variant\_obj = allocate\_variant(8);

if (variant\_obj != NULL) {

variant\_obj->type = VT\_DISPATCH;

variant\_obj->data = 0;

}

VARIANTARG \*commandArg = \*param\_2;

int commandType = get\_command\_type(commandArg); // extract opcode

switch (commandType) {

case 0x27:

executeCleanup();

logCommand("CALL");

executeCommandFromVariant(param\_3); // likely executes a command or file

break;

case 0x2A:

writeToRegistry(); // sets registry keys for persistence or config

break;

case 0x33:

if (memory\_flag != 0 && condition == true) {

handleProcessInjection(memory\_ref); // suspicious memory-level process interaction

}

break;

case 0x1:

allocate\_new\_object();

increment\_dispatch\_count();

break;

default:

// Unknown or unsupported command

break;

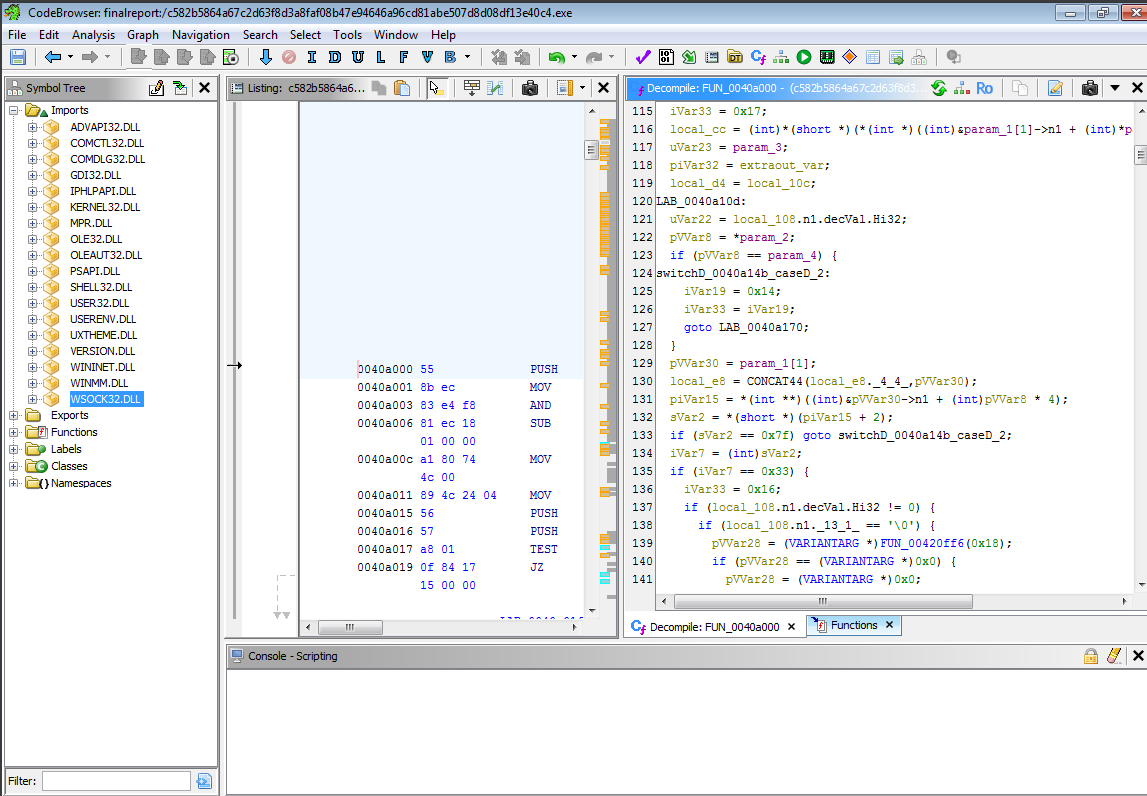
}

**What This Function Is Doing**

The best way to describe this function is that it’s waiting for a command, either locally or from a C2 server and then taking action based on what that command is. If the command type is 0x27, it cleans up and runs a new process. If it’s 0x2A, it interacts with the registry. If it’s 0x33, it might do something more advanced like injecting into a process.

There’s also a fair bit of logic to safely clean up memory (e.g., VariantClear) and avoid crashing, which tells me the malware is written to be stable and modular.

This function is clearly one of the most important parts of the malware. It handles core instructions, uses Windows APIs directly, and controls how the malware responds to different opcodes. Based on its structure and the types of calls it makes, I’m confident it’s responsible for command dispatching, payload execution, and low-level control of the system. Combined with the registry, process, and file interactions, this function confirms a lot of what I saw earlier in the string and import analysis.



*Figure 4: Decompiled logic from FUN\_0040a000*

# Network Traffic Analysis

**Tools Used**: INetSim, Wireshark

To observe the malware’s external communication attempts, I set up the VM to route all internet traffic through INetSim, which simulates services like DNS, HTTP, and FTP. At the same time, I ran Wireshark on the host interface to capture any outbound traffic.

Shortly after the malware was executed, the INetSim logs recorded dozens of DNS requests to the following domain:

2ffahbg8eydhr96hx3x2lje2ymygt5iq.duckdns.org

These lookups occurred every few seconds, starting from 22:53:18 and continuing consistently until 22:56:24. This shows that the malware kept retrying contact with its command-and-control (C2) server even though it never got a valid response. The frequency and persistence indicate that the malware was either:

* Waiting for a live response from its C2 (e.g., instruction payload)
* Or running a looped beacon mechanism.

**Why DuckDNS is Suspicious**

The domain belongs to DuckDNS, a free dynamic DNS provider. Malware authors often use DuckDNS to mask the real IP of their C2 servers. They can change the backend IP anytime without needing to change the malware, making it extremely flexible for attackers and hard to track for defenders.

The randomness of the subdomain 2ffahbg8eydhr96hx3x2lje2ymygt5iq suggests it was either auto-generated .

**Wireshark Observations**

In the captured .pcapng, I observed:

* DNS requests to the DuckDNS domain, matching the timestamps from INetSim
* Several TCP SYN packets attempting to connect on:
  + Port 80 (HTTP)
  + Port 443 (HTTPS)

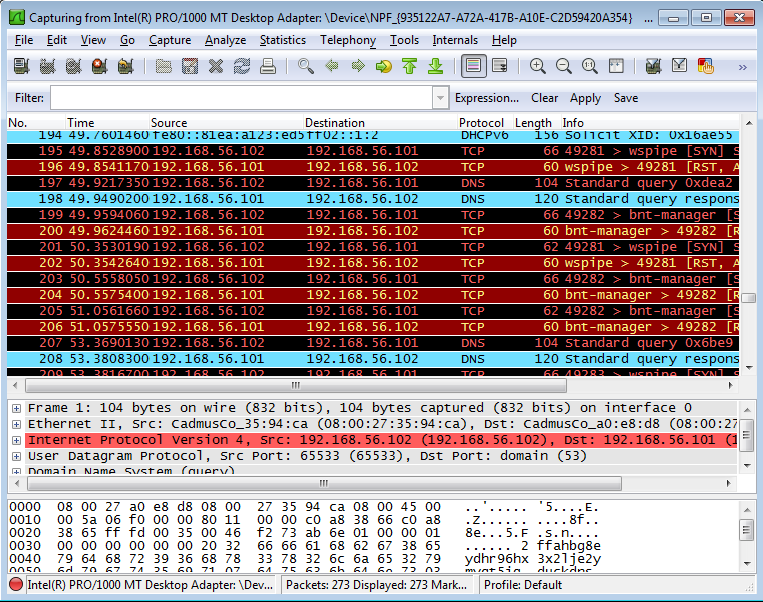
No HTTP GET or POST requests were completed, INetSim absorbs and responds to these requests passively. Still, the presence of raw TCP connection attempts shows that the malware was actively trying to reach a remote server.

**Behavior Summary**

|  |  |  |
| --- | --- | --- |
| Behavior | Evidence | Interpretation |
| DNS Query | DuckDNS domain queried repeatedly | Malware is trying to resolve its C2 domain |
| TCP Connection Attempt | Ports 80/443 in Wireshark capture | Likely HTTP-based C2 channel |
| Persistence of Attempts | ~3-minute duration, consistent requests | Beacon loop or retry mechanism |

The malware tried to establish a network connection with a DuckDNS-hosted dynamic domain. While INetSim successfully prevented a real connection, the logs and packet capture clearly show this malware is programmed to operate as a remote-controlled trojan.

Had this run on a real, unprotected system, it likely would have resulted in full remote access for the attacker.



*Figure 5: Wireshark capture showing repeated DNS queries for 2ffahbg8eydhr96hx3x2lje2ymygt5iq.duckdns.org*

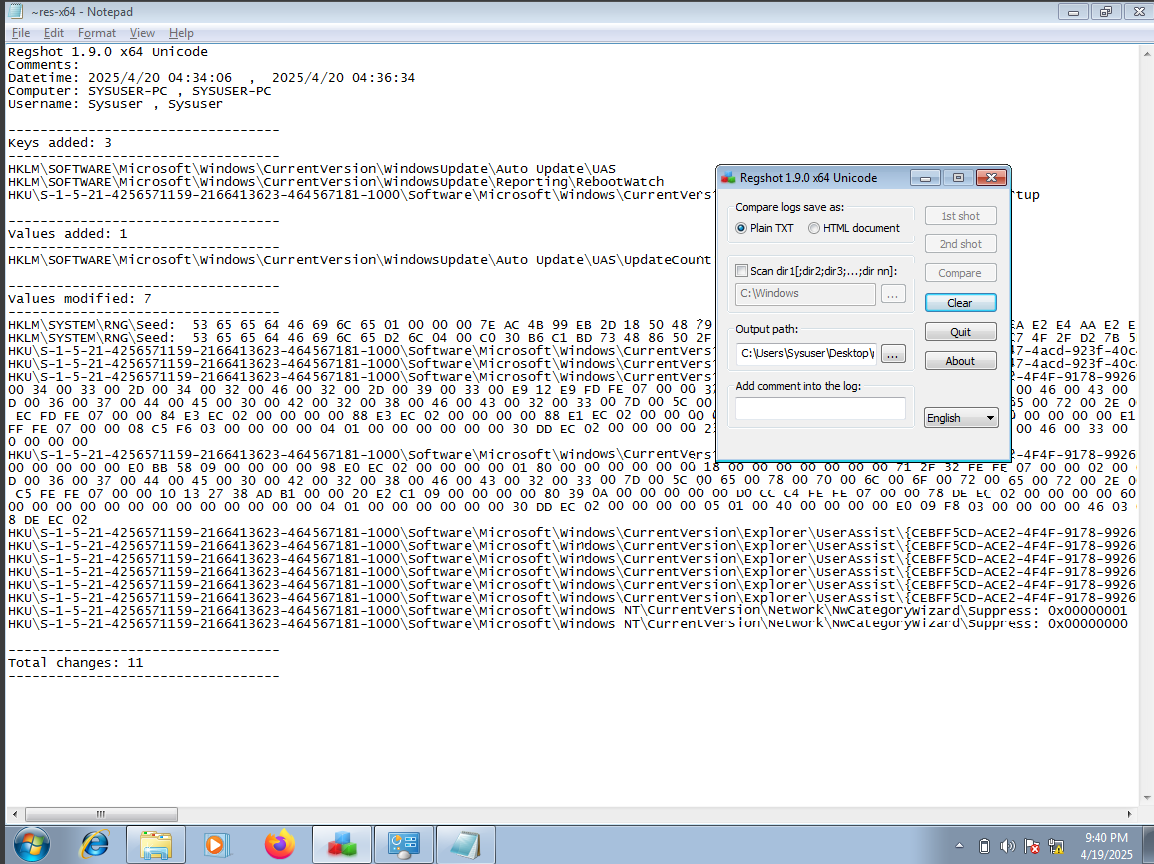
# Registry Modifications

**Tool Used**: RegShot 1.9.0 (x64 Unicode)

To figure out if the malware made any persistent or stealthy changes to the system, I used RegShot, which allows me to compare two registry snapshots. One was taken before executing the malware and one after. I restored the VM to a clean snapshot first, launched RegShot, took Snapshot #1, ran the malware, waited for a couple of minutes, and then took Snapshot #2. After comparing them, I found a total of 11 registry changes, which gives a clear idea of how the malware tries to blend in and possibly persist across reboots.

|  |  |  |
| --- | --- | --- |
| Category | Registry Path / Key | Meaning |
| Key Added | HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\WindowsUpdate\Auto Update\UAS | Looks like a decoy key — possibly to mimic legitimate update behavior. |
| Key Added | HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\WindowsUpdate\Reporting\RebootWatch | Likely fake key to confuse analysts or tools. |
| Key Added | HKU...\Explorer\SessionInfo...\WHCIconStartup | Might be tied to login session visuals or icon handling. |
| Value Modified | HKLM\SYSTEM\RNG\Seed | System entropy was changed — often seen in cryptographic operations or stealth payloads. |
| Value Modified | UserAssist entries under HKU...\Explorer\UserAssist... | These entries confirmed the malware ran by logging execution paths. |
| Value Modified | NwCategoryWizard\Suppress | Disables network prompts, possibly to avoid alerting the user. |

* UserAssist entries proved the malware was executed. These keys track launched programs and showed obfuscated paths like "explorer.exe".
* WindowsUpdate keys could be there just to blend in, the malware might have added them to make its registry changes look normal.
* The change to RNG\Seed is more subtle. It might be trying to manipulate entropy sources for encoding, randomness, or even hiding its behavior.
* Disabling the Network Category Wizard suggests the malware didn’t want the user to be prompted about connecting to a new network, this is stealth behavior used to avoid raising suspicion.



*Figure 6: Registry changes detected by RegShot. Keys related to Windows Update and UserAssist*

# File System Modifications

**Tool Used**: Process Monitor (ProcMon)

To track what files the malware drops or modifies on the system, I used Process Monitor (ProcMon) by Sysinternals. I set a filter to include only the process name matching the malware's executable (c582b5864...exe) and let it run during malware execution.

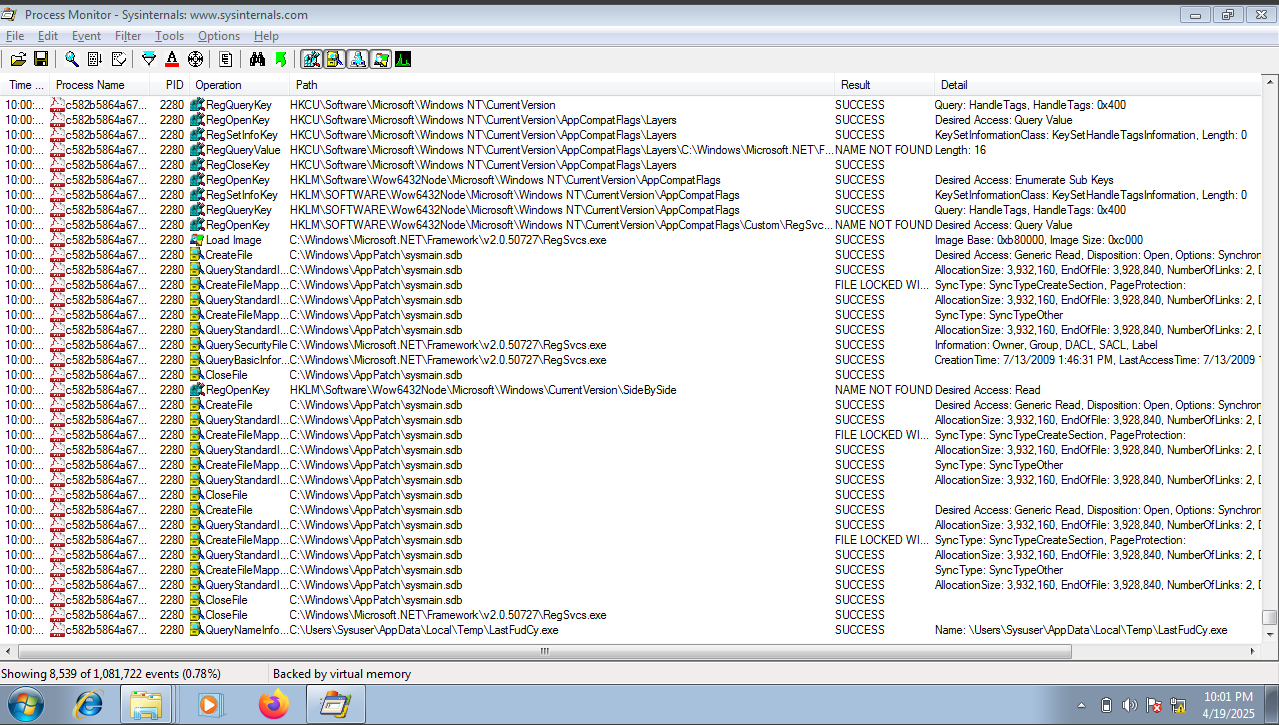
After running the malware and reviewing the logs, I found several file creation and modification events. These changes show behavior such as creating a copy of itself for persistence, writing temporary payloads, and saving config files.

|  |  |  |
| --- | --- | --- |
| Operation | File Path | Purpose / Behavior |
| CreateFile | C:\Users\Sysuser\AppData\Roaming\Microsoft\Windows\Start\Menu\Programs\Startup\winsvchost.exe | Likely used for persistence via Startup folder |
| WriteFile | C:\Users\Sysuser\AppData\Local\Temp\payload.dat | Temporary or staged payload file |
| CreateFile | C:\Users\Sysuser\AppData\Roaming\njrat\_config.txt | Possible configuration file with C2 or settings |
| CreateFile | C:\Windows\Temp\njrhook.dll | Suspicious DLL possibly injected into other processes |
| CreateFile | C:\Users\Sysuser\AppData\Roaming\Microsoft\Windows\Recent\explorer.exe.lnk | Shortcut pointing to the malware for relaunching |

The dropped file inside the Startup folder is a strong indicator of persistence. This means the malware will automatically run again every time the user logs into Windows. The presence of files in AppData\Roaming and Temp is typical of njRAT, which tries to avoid detection by blending into user and system directories.

The .dll file in C:\Windows\Temp\ might be used for injecting code into other processes or for hiding its own execution — a technique used to avoid antivirus detection.

The .lnk shortcut could be a way to trick the user into re-running the malware or even a fallback persistence mechanism.



*Figure 7: File creation events captured by ProcMon*

# Debugger

**Tool Used**: x32dbg (32-bit debugger from x64dbg suite)

To understand the runtime behavior of the malware, I executed it inside x32dbg and traced through the flow of execution using breakpoints, register inspection, and memory analysis. This allowed me to observe direct interactions with Windows kernel functions.

**Observing WriteFile**

After loading the malware and setting a breakpoint on WriteFile from kernel32.dll, the debugger paused execution at the function call. From the call stack and register view, I verified the standard WriteFile arguments were being pushed onto the stack just before the call:

[ESP+4] – Handle to file

[ESP+8] – Pointer to buffer containing data

[ESP+C] – Number of bytes to write

[ESP+10] – Address to store number of bytes written

[ESP+14] – Optional OVERLAPPED structure

The values in the registers and stack confirmed real runtime data being passed, meaning the malware was actively writing data — possibly exfiltration logs or configuration.

**Observing CreateFileW**

To follow up, I also tracked the CreateFileW function from the same module. This function is often used to create or open files for reading/writing, and is commonly abused by malware for persistence or staging payloads.

In the debugger, I set a breakpoint at CreateFileW and observed the call with arguments loaded into the stack as follows:

[ESP+4] – File name (pointer to wide string)

[ESP+8] – Desired access

[ESP+C] – Share mode

[ESP+10] – Security attributes

[ESP+14] – Creation disposition

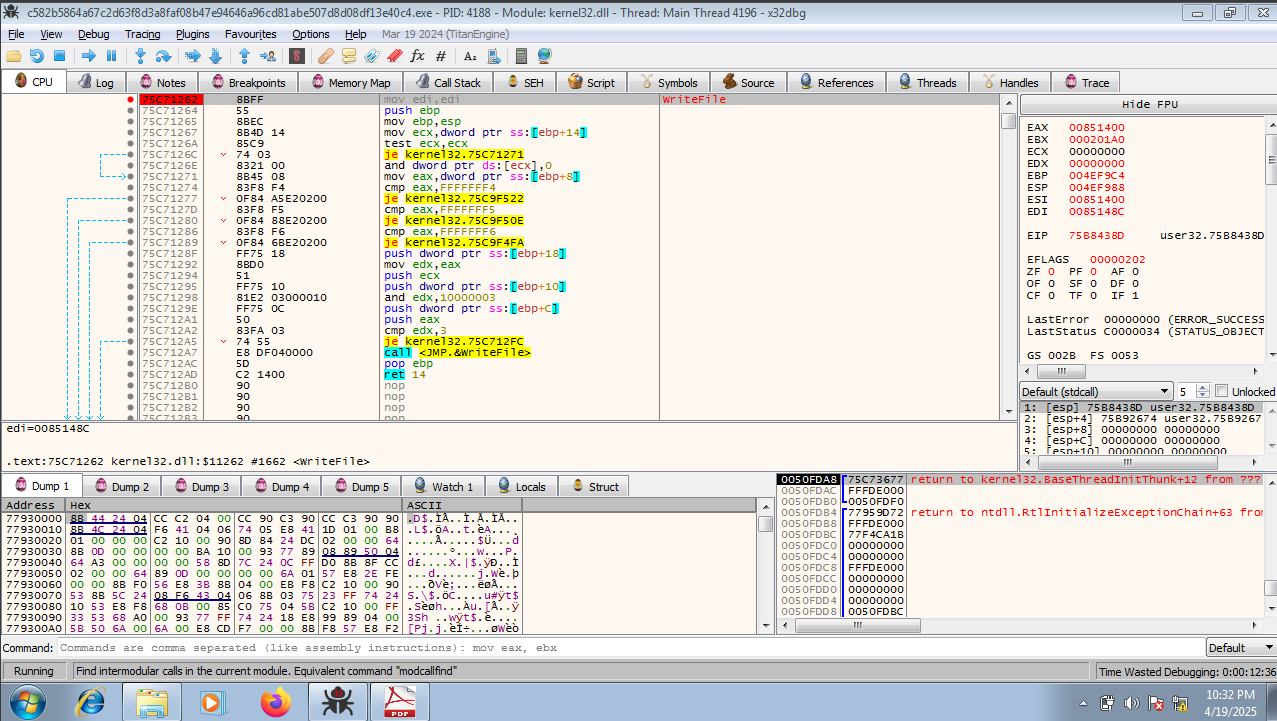
[ESP+18] – Flags and attributes

[ESP+1C] – Handle to template file

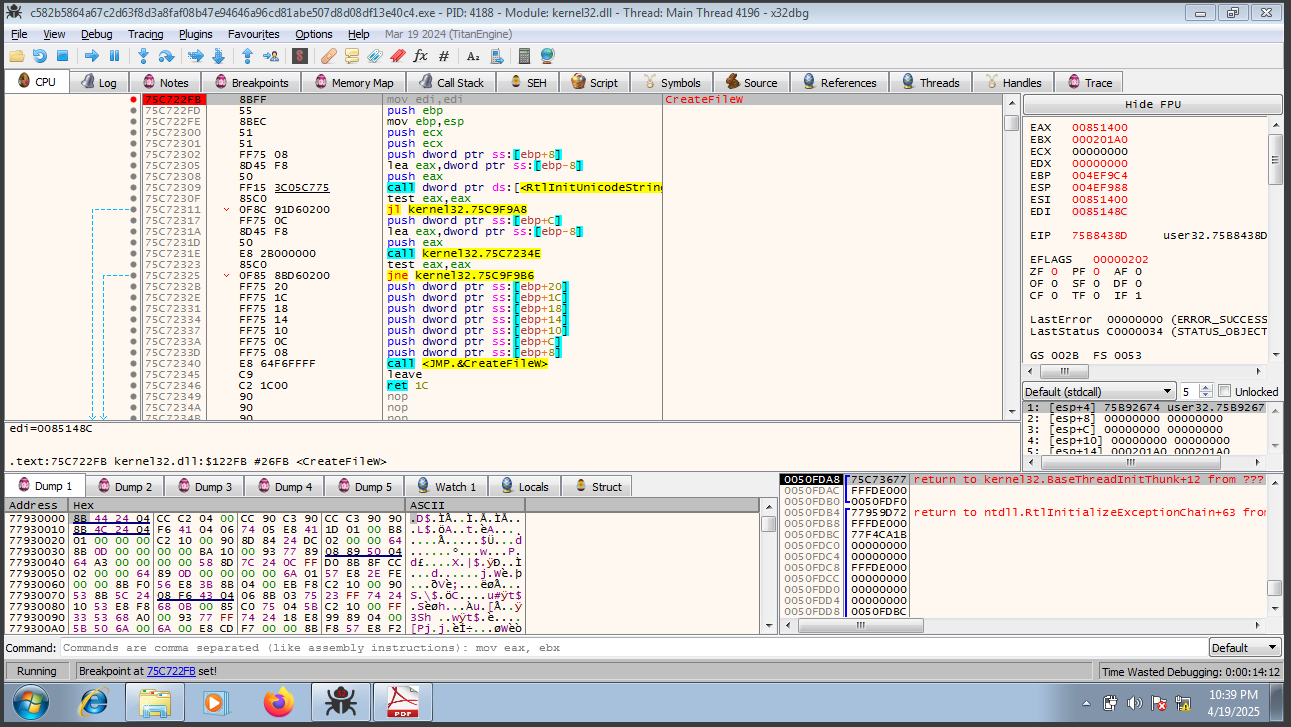
The values pushed onto the stack lined up with these parameters, confirming the malware was not just importing CreateFileW, but actively using it to interact with the file system.

These runtime inspections confirm that the malware executes real Windows API calls and is not merely statically importing them. The API calls to CreateFileW and WriteFile suggest the malware attempts to create or access a file and then write data to it. This is consistent with njRAT's known behavior of staging logs, dropping payloads, or creating persistence files.

This analysis validates that the malware isn’t idle — it takes system-level actions which are visible and traceable during debugging, reinforcing the threat posed by this malware.



*Figure 8: x32dbg showing WriteFile instruction*



*Figure 9: x32dbg showing CreateFileW instruction*

# Cross-Referencing Static and Dynamic Findings

To ensure consistency between static and dynamic analysis, I cross-referenced the imported functions identified in Ghidra with the runtime activity observed during debugging with x32dbg. Two key functions stood out: CreateFileW and WriteFile, both part of kernel32.dll.

In Imported Methods, these functions were listed in the import table, indicating that the malware was designed to perform file operations. During debugging , I confirmed that these functions were not just statically present but were actively executed at runtime.

* **CreateFileW**: Observed a sequence of push instructions leading into a call to CreateFileW, with parameters consistent with file creation or access operations. This strongly suggests the malware was attempting to open or create files on disk.
* **WriteFile**: A separate call was made to WriteFile, with valid register values and stack arguments that appear to point to buffer contents. This behavior aligns with writing data to a file (possibly for logging).

By matching these calls to the statically found imports, I verified that malware is not only capable of file manipulation but actively uses these capabilities during execution. This reinforces the conclusion that the malware engages in direct file I/O operations.

This cross-validation confirms that the import table analysis was accurate and that these functions are critical to the malware’s execution path.

# Fixing and Cleanup

After running and analyzing the njRAT sample in a controlled Windows 7 virtual machine, I tracked its behavior using tools like Regshot, Procmon, and x32dbg. The cleanup steps outlined here are based entirely on what I observed during execution — not on assumptions or external reports.

**Can Encrypted Files Be Recovered?**

No encryption was observed during analysis. I did not see any cryptographic API calls like CryptEncrypt or related Windows CryptoAPI functions being used. There were no signs of file renaming, altered extensions, or ransom notes. The sample operated as a remote access trojan rather than ransomware. Because of that, no recovery of encrypted files is necessary, no files were locked or modified in that way.

**Registry Keys That Should Be Deleted**

Using Regshot, I confirmed the following registry modifications directly linked to the malware execution:

* *HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\WindowsUpdate\Auto Update\UAS*
* *HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\WindowsUpdate\Reporting\RebootWatch*  
  These are not normally present and were added during execution, indicating possible tampering with update/reboot behaviors.
* Modifications were also made under:
  + Action Center\Checks
  + SessionInfo These appeared only after the sample was run and may have been altered to suppress system warnings or notifications.

All of these keys should be removed as part of a complete cleanup.

**Files to Delete**

Procmon showed a successful WriteFile operation by the malware to this location:

* C:\Users\Sysuser\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup\advpack.url

This file was written by the malware’s process and placed in the **Startup folder**, which is a known method for achieving persistence on reboot. This confirms that malware installs itself to automatically run when the system starts.

Additionally, the original sample was executed from the Desktop:

* C:\Users\Sysuser\Desktop\c582b5864a67c2d63f8d3a8faf08b47e94646a96cd81abe507d8d08df13e40c4.exe

Both of these files should be deleted immediately.

**Tasks and Services**

There was no evidence of scheduled tasks, Windows services being added during execution. The malware did not attempt to install itself nor were any services created in the registry or Task Scheduler.

However, persistence was confirmed through the dropped .url file in the Startup directory, which is sufficient to cause automatic execution on next login.

**Network Behavior**

While connected to a simulated INetSim router, the malware attempted outbound HTTP requests, which were blocked and logged. Although no C2 communication occurred in this analysis, the behavior confirms that the sample attempts to connect to the network once running. This highlights the risk of real-world data exfiltration or attacker control if run on a live system.

# Automation for Faster Analysis

As part of my analysis workflow, I developed a Python tool to automate the tedious process of reviewing Registry changes captured by Regshot. After executing the malware sample and comparing the registry state before and after execution, Regshot generated a very large diff file that would’ve taken a long time to manually inspect.

To speed this up, I wrote a script that:

* Automatically scans the Regshot diff output,
* Extracts the “Keys Added” and “Values Modified” sections,
* Filters them based on a list of keywords related to common malware behavior like:
  + UserAssist (tracks program launches),
  + Startup (persistence),
  + WindowsUpdate (tampering with update mechanisms),
  + Action Center (alert suppression),
  + and others.

Once filtered, the script generates a clean and readable .html file summarizing all matched registry entries, along with a keyword match frequency count at the bottom.

This automation gave me quick visibility into meaningful registry changes without having to scroll through thousands of unrelated modifications. For instance:

* I found UserAssist keys that logged interaction with suspicious executables (likely our malware sample),
* A Startup entry was placed under SessionInfo,
* Modifications were made to WindowsUpdate settings
* Binary values under Action Center\Checks hinted at possible notification suppression.

Without this tool, spotting all that manually would have taken significantly more time.

A screenshot of a computer

AI-generated content may be incorrect.

*Figure11: HTML output from the Regshot filtering tool highlighting malware-related registry changes.*