Shayan Ahmed Khan

Sr. Threat Researcher

[LinkedIn]
[Medium]
[Github]
[Website]

Secrets of commercial RATs! NanoCore dissected

This article includes the technical analysis of a commercial RAT which is easily available on black market for cheap price. NanoCore is a famous Remote Access Trojan malicious software that has its own client builder and multiple delivery methods. In this article, I will not focus on the initial delivery method which could be a malicious attachment or spear phishing. I will dive directly into the first stage malware sample.

SHA256 Hash:
1605F0E74C7088B8A2CA7190B71C83F8DC0381E57D817DF3530BDA4AC5737511
Build: x86 and dotnet (multiple stages)
Category: RAT (Remote Access Trojan)
Family: NanoCore
Version: 1.2.20

Analysis Environment:

I use FlareVM as my base VM for malware analysis and detonation. I use REMnux Box Ubuntu machine as DNS server and network simulator for the FlareVM.

1. https://github.com/mandiant/flare-vm
2. https://docs.remnux.org/install-distro/get-virtual-appliance

Tools:

- IDA Freeware
- Dnspy
- Inetsim
- Process hacker
- Procmon
- TcpView
- Wireshark
- HxD editor
- Cff-Explorer
- ResourceHacker
- Netcat
- DIE
- De4Dot
- Floss
- PE Studio
- ExeInfoPE

STAGE 1:

Generic methodology that I follow for malware analysis is:

1. Basic Static Analysis
2. Basic Dynamic Analysis (initial detonation)
3. Advanced Static and Dynamic Analysis (TTP extraction)

Basic static analysis involves looking at interesting strings and API calls. I use floss utility for string extraction process. It can also decode unicode strings and extract stack-based strings which is helpful in some cases. For looking at interesting API calls, I use PE Studio as it also provides red flags to potential malicious APIs.

Interesting strings & APIs:

- Software\Microsoft\Windows\CurrentVersion
- CreateProcessA, ShellExecuteA, RegSetValueExA, RegCreateKeyExA

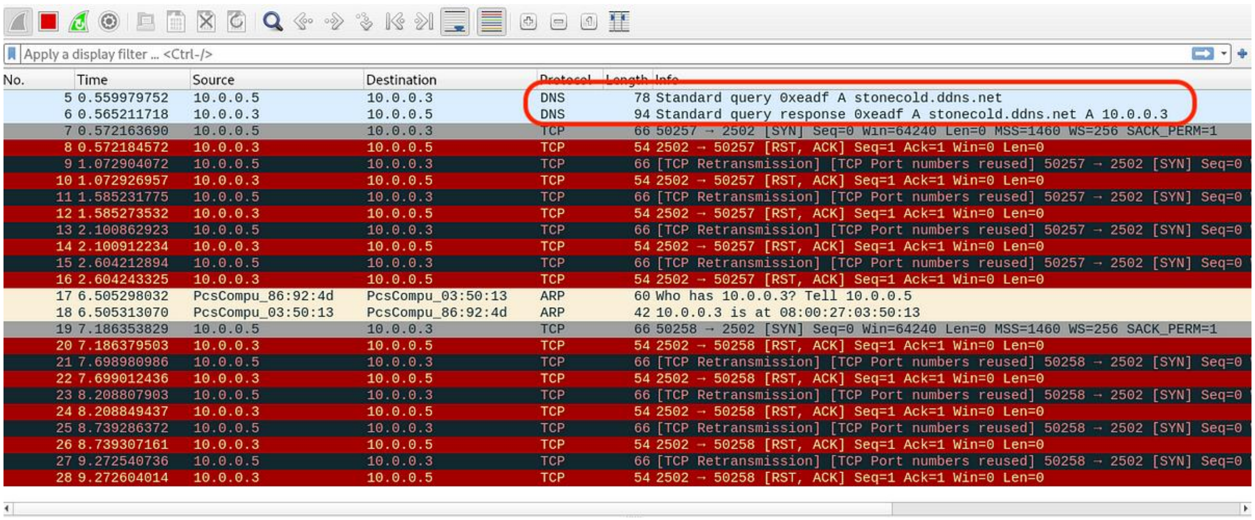
The strings show that malware might be achieving persistence using **Registry Run Keys** technique as it is also creating and setting registry keys using the APIs **RegCreateKeyExA**, **RegSetValueExA**. It is also executing something, maybe a next stage payload? using the APIs of **CreateProcessA** or **ShellExecuteA**.

Initial Detonation:

In the basic dynamic analysis, i detonate the malware in presence of **Procmon** for host- based indicators and **Wireshark** for network-based indicators. The prcomon is setup in the detonation FlareVM and the wireshark is setup at REMnux box which is simulating the network traffic using **inetsim**.

Network Indicators:

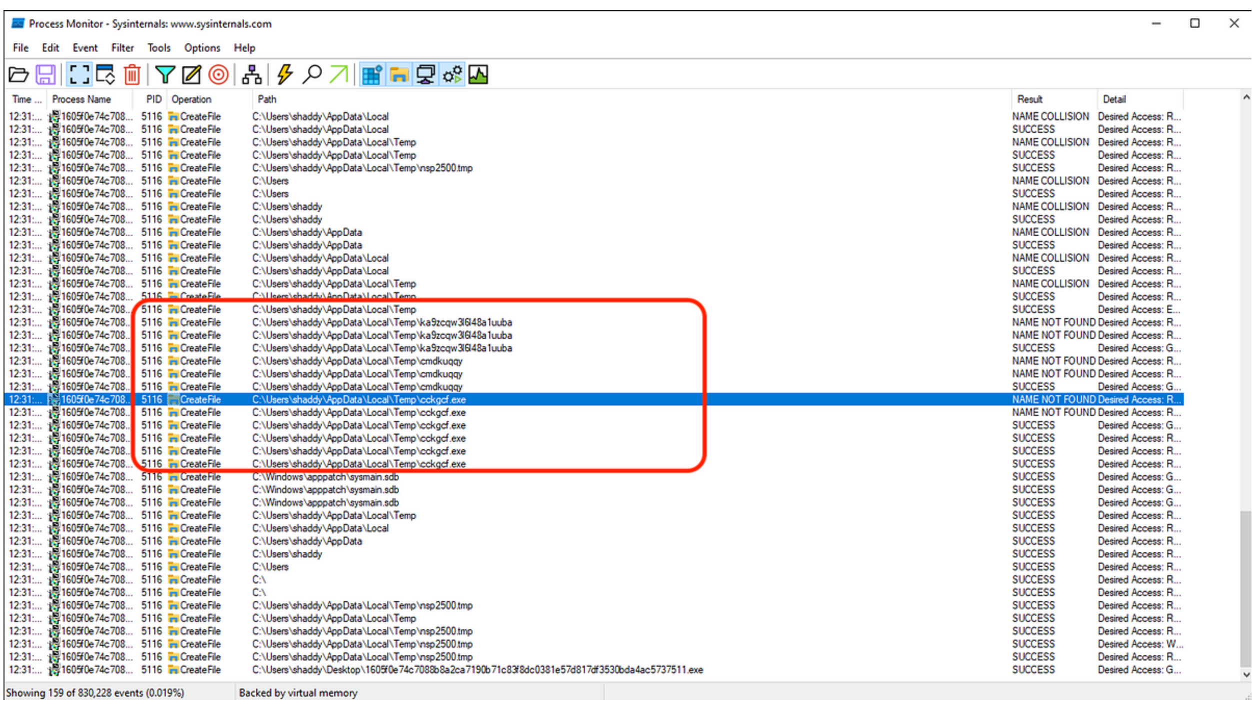
1. Contacting malicious domain: **stonecold.ddns.net**
2. Multiple **TCP packets** sent after DNS query.
3. Creating socket connection on specified port: **2502**



Network indicators wireshark packet capturing

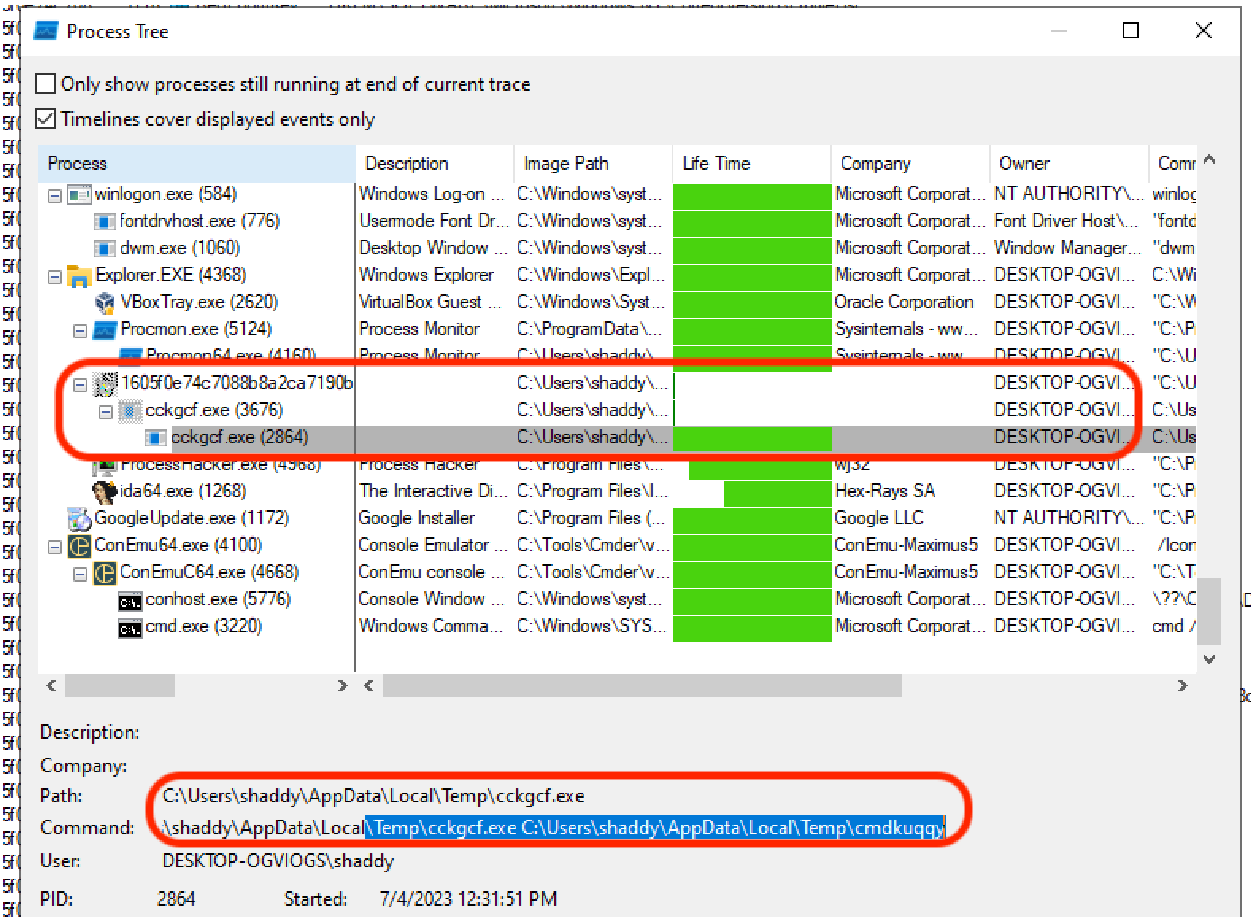
Host-based Indicators:

1. Creates multiple files in **%temp%** folder
2. Extracts **cmdkuqqy**, **cckgcf.exe** and **ka9zcqw3l6l48a1uuba**



Host-based indicators procmon logs

It looks like stage1 malware is extracting 3 files from its resources. The second stage malware is then executed with the file passed as parameter. I have checked the process tree of malware and it shows that the original sample extracted the 2nd stage malware files in %temp% and executed it as shown in the picture below:



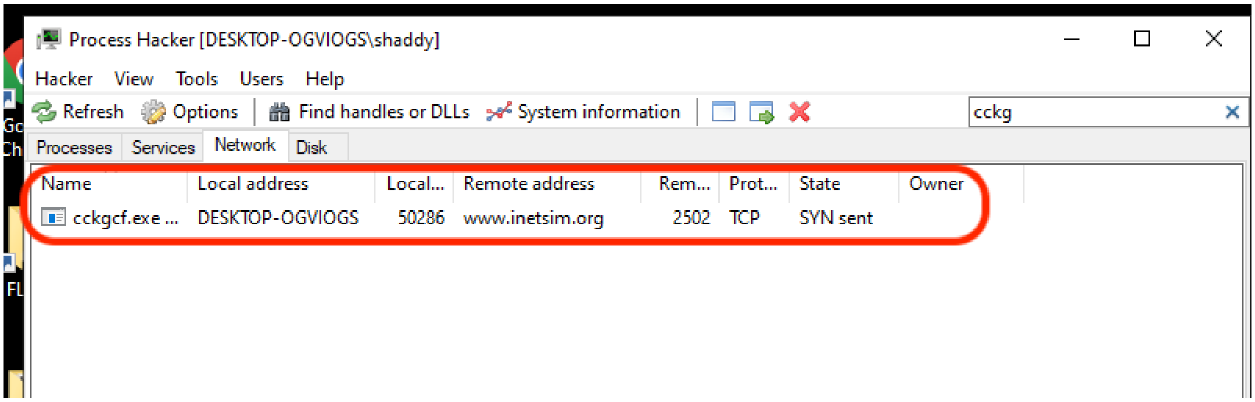
Process Tree

STAGE 2:

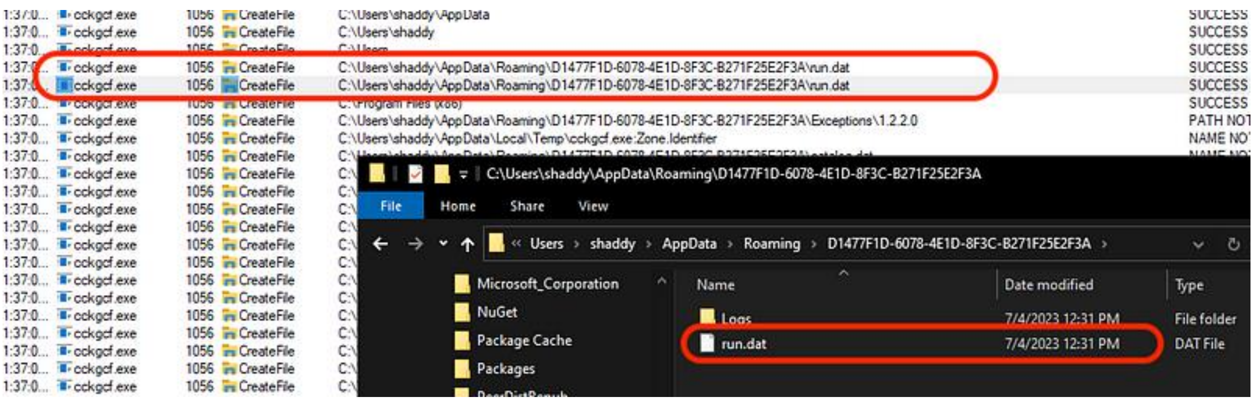
The second stage malware is **cckgcf.exe** which makes use of encrypted files **cmdkuqqy** and **ka9zcqw3l6l48a1uuba** for further malware execution. From the process tree above, it is visible that second stage sample (cckgcf.exe) launches another process of itself. This is common behavior in malware which employs defense evasion techniques to **deobfuscated/decrypt** payloads at run-time.

The indicators for stage2 malware are as follow:

1. Starts itself as child process
2. Keeps sending SYN packets to the remote **C2 server** on port **2502**
3. Creates a dat file (**run.dat**) in %Appdata% folder
4. Creates persistence of itself by using **Registry Run keys** procedure.



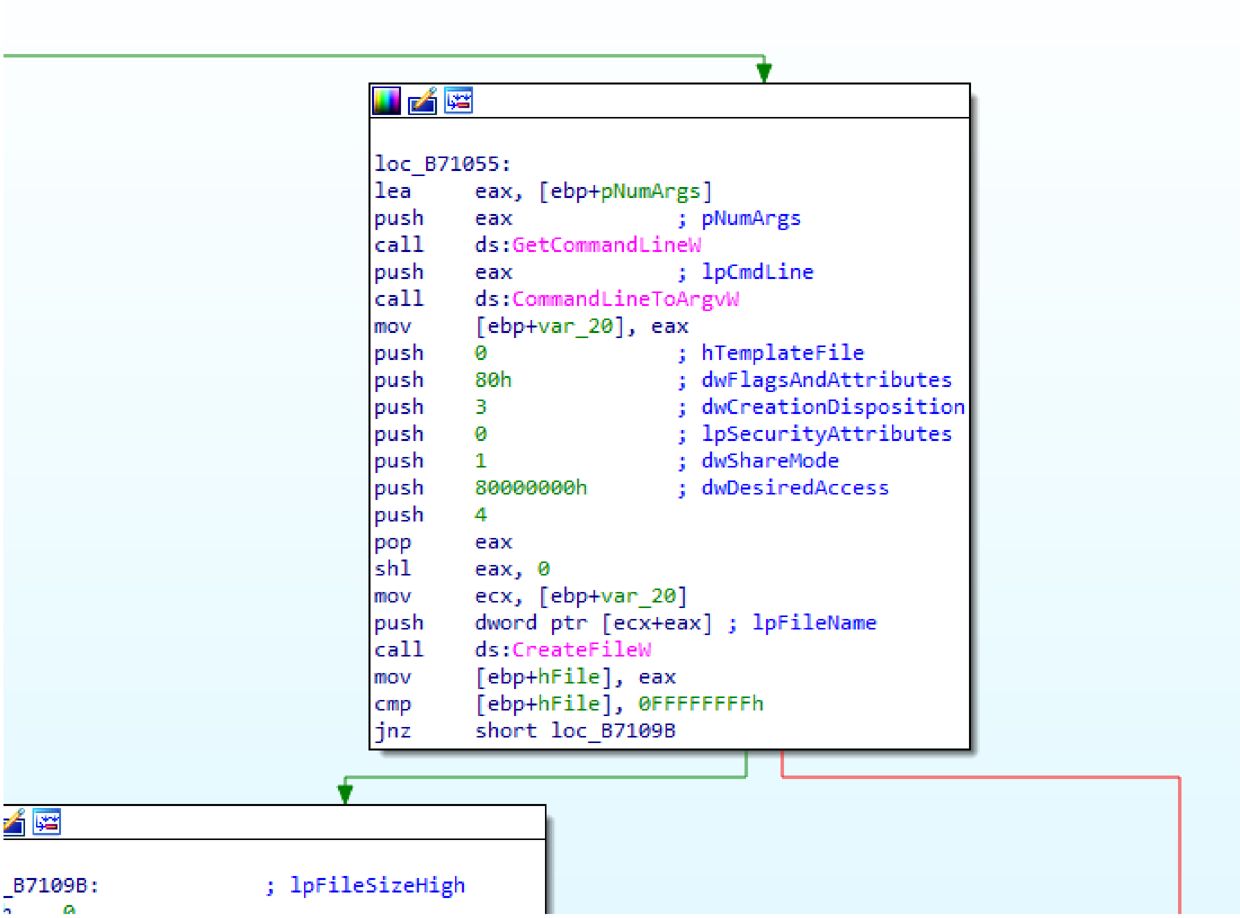
Network indicators stage2



Host indicators stage2

Advanced Static Analysis:

I use advanced static analysis by looking at the assembly of malware in IDA freeware. From the initial analysis, it looks like the stage2 malware accepts a cmdline argument for execution. If the argument is passed, then it processes further, else it exits.

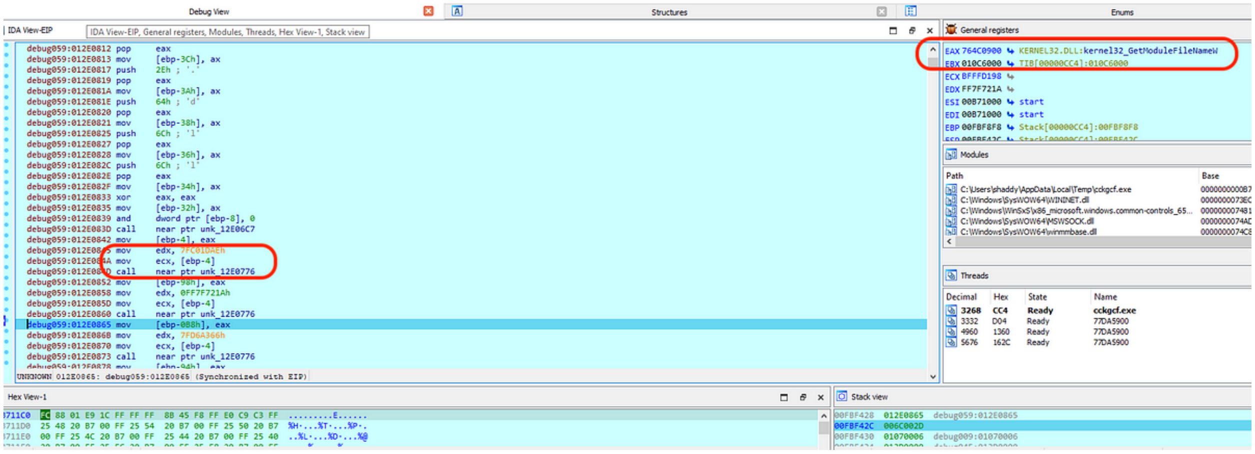


IDA freeware stage2 malware analysis

All the API calls in stage2 malware are resolved dynamically, so static analysis doesn't help here. Therefore, I've started advance dynamic analysis. I use IDA local debugger for advance dynamic analysis.

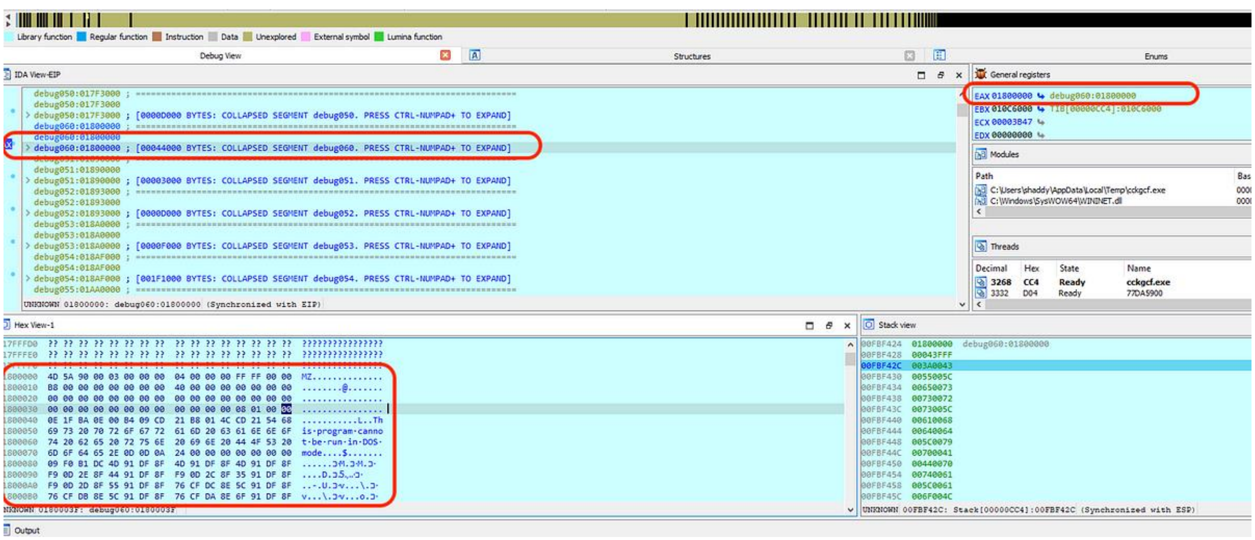
Advanced Dynamic Analysis:

Advanced dynamic analysis revealed that, there are multiple modules that are loaded into the stage2 malware which are not added by default. The libraries like shlwapi.dll and wininet.dll are included at run-time. The API calls are all obfuscated and resolved at run-time to avoid detection by anti-malware systems. The combination of **LoadLibraryA** and **GetProcAddress** is used to achieve dynamic API resolution.



Dynamic API resolution stage2

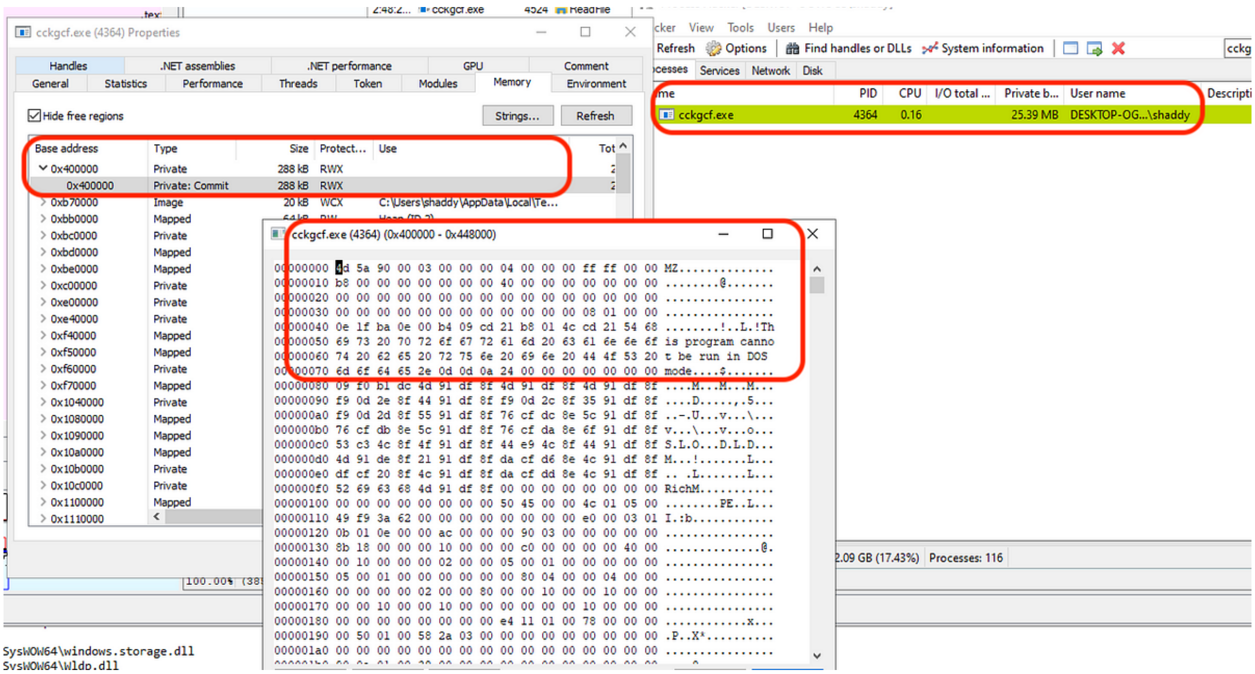
I resolved the API calls while debugging malware and located the **shellcode** that is being **decrypted** and then **injected** into the process space of malware itself. The shellcode is another portable executable binary bytes that are executed in a separate thread. The starting bytes of **4D 5A (MZ)** are the identifier of a portable executable which is shown in the screenshot below:



Shellcode injection stage2

The process injection technique that is being used is called **process hollowing**, in which a process is started in a suspended state which in this case is malware itself. Then a memory is allocated in the suspended process and shellcode is written into that memory. Finally the address of image base is changed to the starting address of shellcode and process is resumed from suspended state. Now it will start it's execution from the injected shellcode.

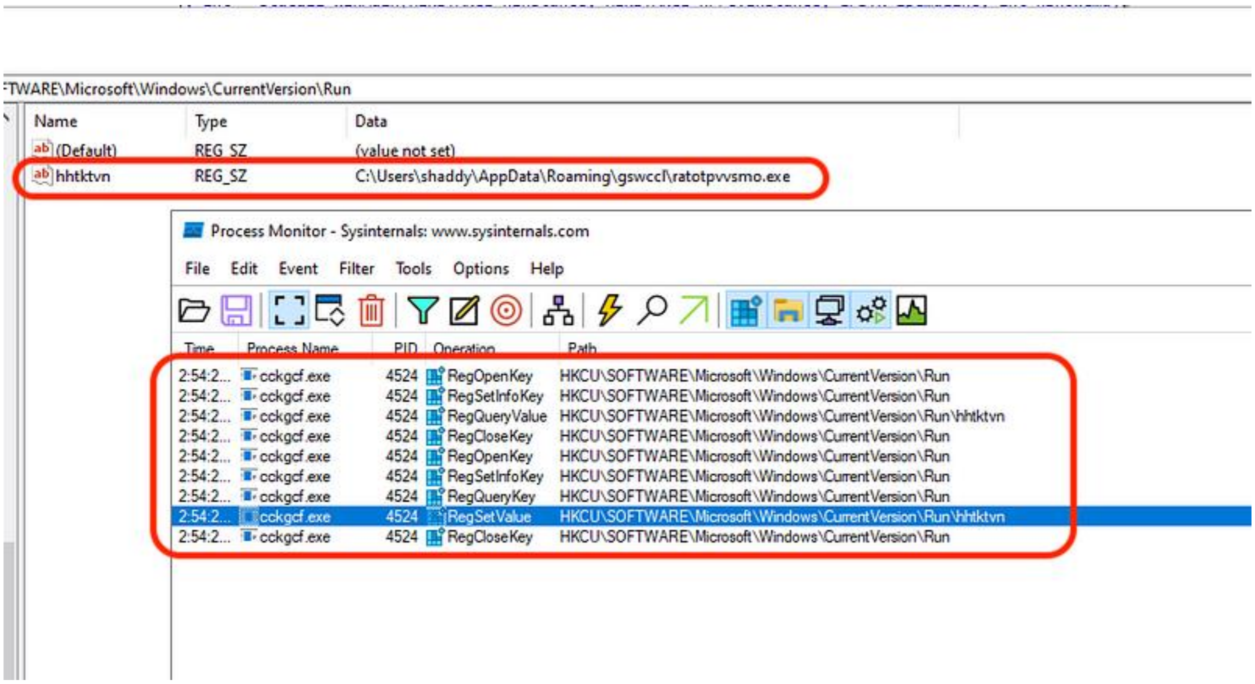
To verify memory related modification, I use process hacker which is an excellent resource to monitor the processes. Injected bytes could be found easily by looking at the memory protections of running process. For injection, a memory protection with permission of all READ, WRITE and EXECUTE are required, therefore I look for RWX memory protections which shows the injected memory bytes in a process. In the screenshot, the injected bytes are shown which are equal to the ones that I have debugged using IDA.



Shellcode Injection memory view stage2

One cool feature of process hacker is that we can directly dump shellcode from the memory to a file and since in this case, the shellcode is a whole portable executable and not a position independent shellcode therefore, I could analyze it separately as a next **stage3** malware.

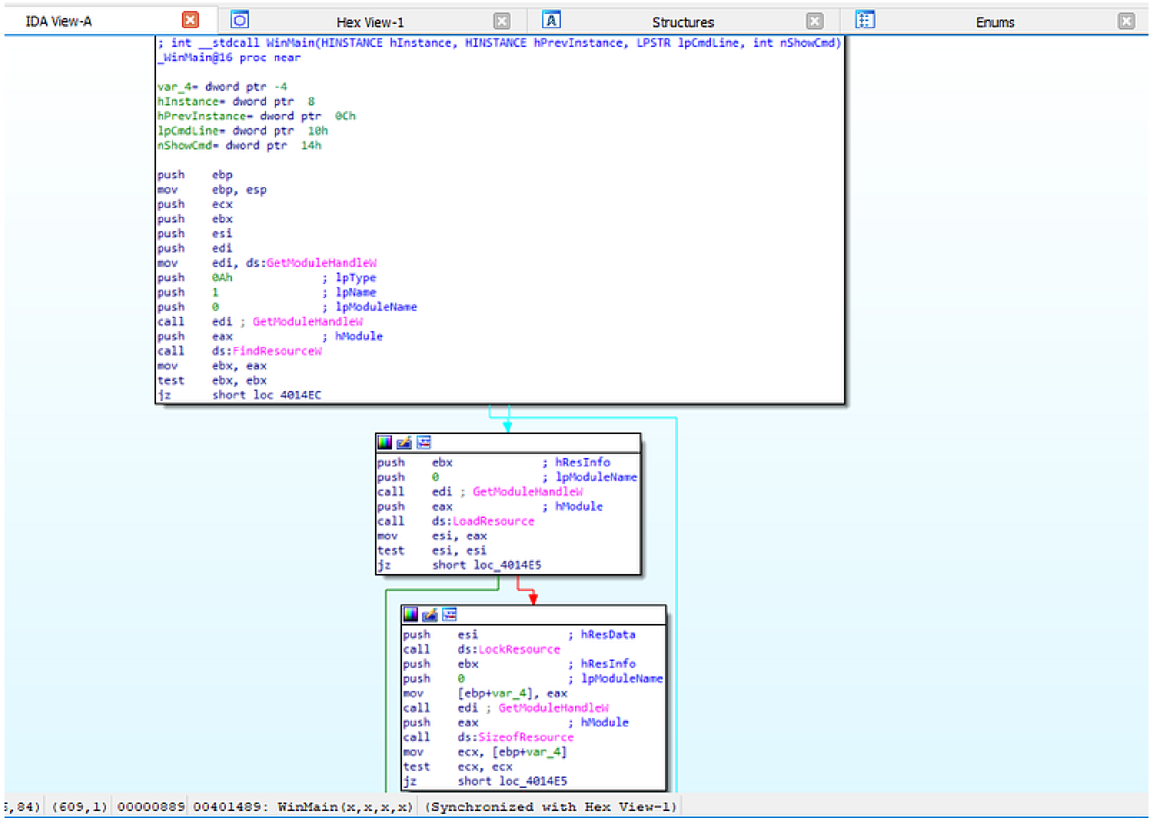
Another indicator of stage2 malware is that is persists itself by registry keys. The stage2 malware creates persistence by adding a registry key value to a binary named: **ratotpvvsmo.exe** in the %Appdata% folder called **gswccl.**



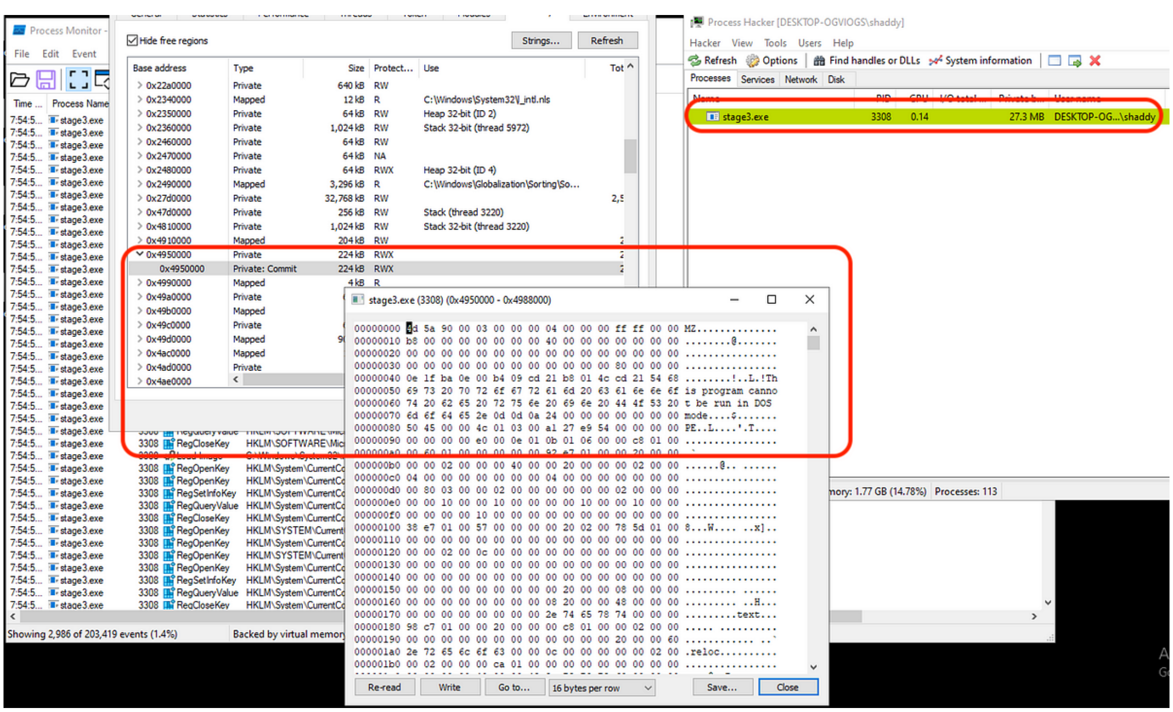
Persistence stage2

STAGE 3:

Stage3 malware that was Portable executable shellcode injected into the process space of stage2 malware is another resource extractor stage. It just repeats the cycle, extract and decode shellcode bytes from its resources and injects in itself again. This process just adds another layer of defense evasion technique.



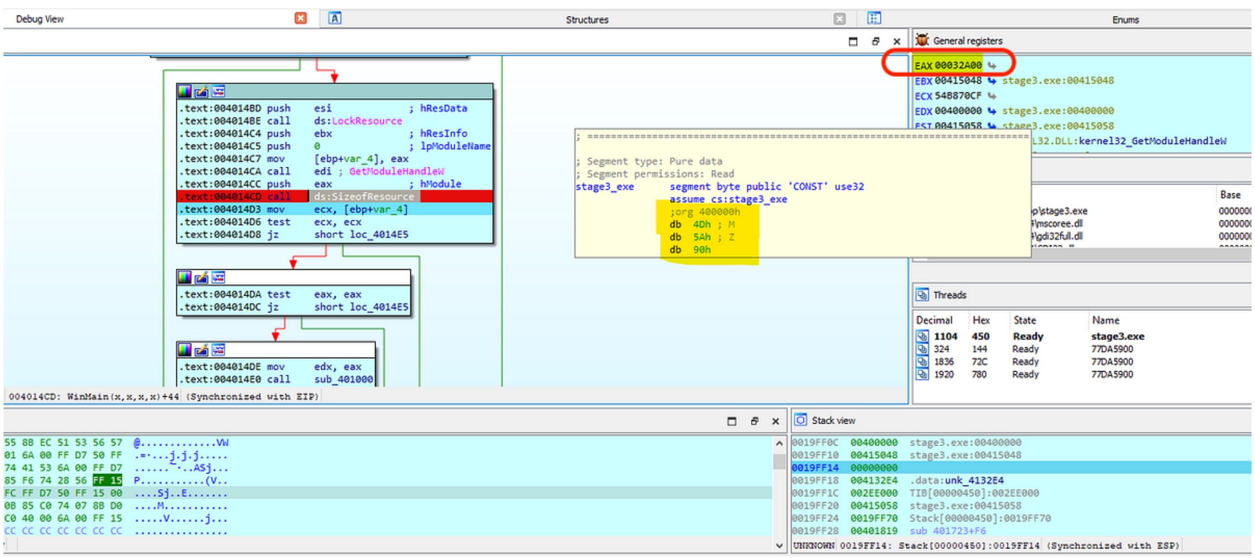
Resource extraction stage3



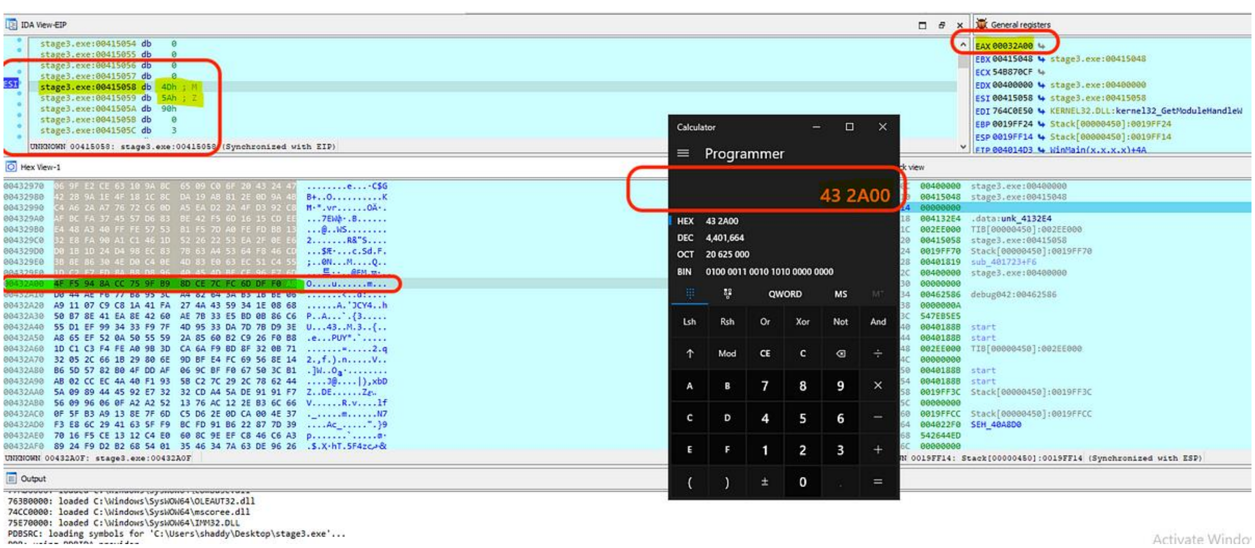
Process injection stage3

I located the shellcode again while debugging and extracting it out using process hacker.

- To locate the shellcode in the memory, I analyzed the registers and found the handle to the shellcode memory
- From then on, I only had to find the length of shellcode to copy from hex
- I used the value returned by API **SizeofResource** to calculate the size of shellcode as shown in the register **eax** which is **32A00**
- Next part is simple, I just added the value to the address space where the shellcode is starting



Shellcode size stage4



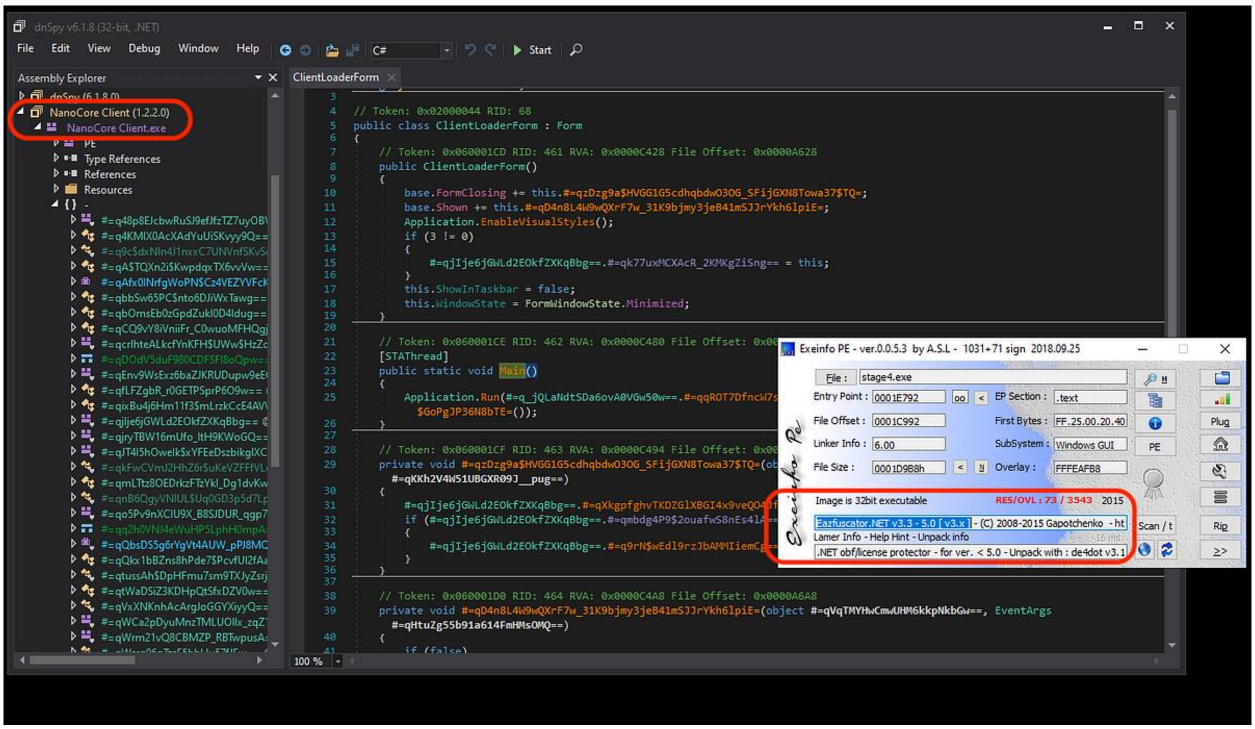
Shellcode address stage4

I dumped the shellcode from IDA freeware hex view in a binary file. It is another portable executable which could be labelled as stage4 or final stage malware.

However, extracting shellcode from resources using IDA freeware sometimes causes unknown problems, like the configurations are not being decrypted into the final stage payload. So, I used **Resource hacker** tool to dump the last stage malware and started analyzing it.

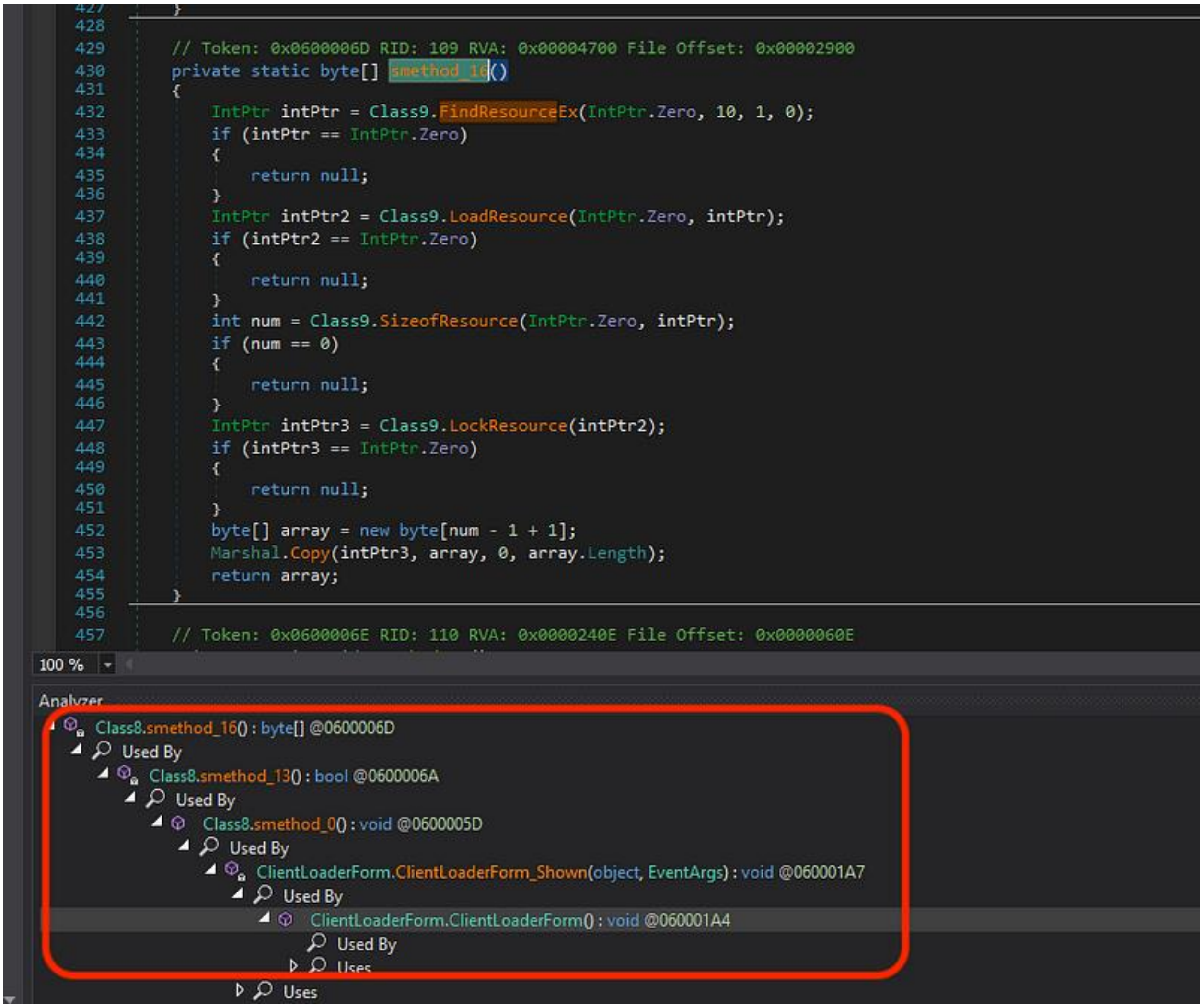
STAGE 4: NanoCore v1.2.2.0

Final stage malware is a dotNet build binary. It is a **NanoCore Client binary** of version v1.2.2.0 which is highly obfuscated. I used ExeInfoPE to identify the obfuscation. **Eazfuscator** has been used to obfuscate the final stage dotent malware. Luckily there are open-source deobfuscators available for this type of obfuscation.



Final stage obfuscated malware

Similar to all RATs, NanoCore extracts its **configuration file** and adjust its settings to the specified configuration. It extracts the configurations and extra malware plugins from the resources. The resource is encrypted for defense evasion purposes.

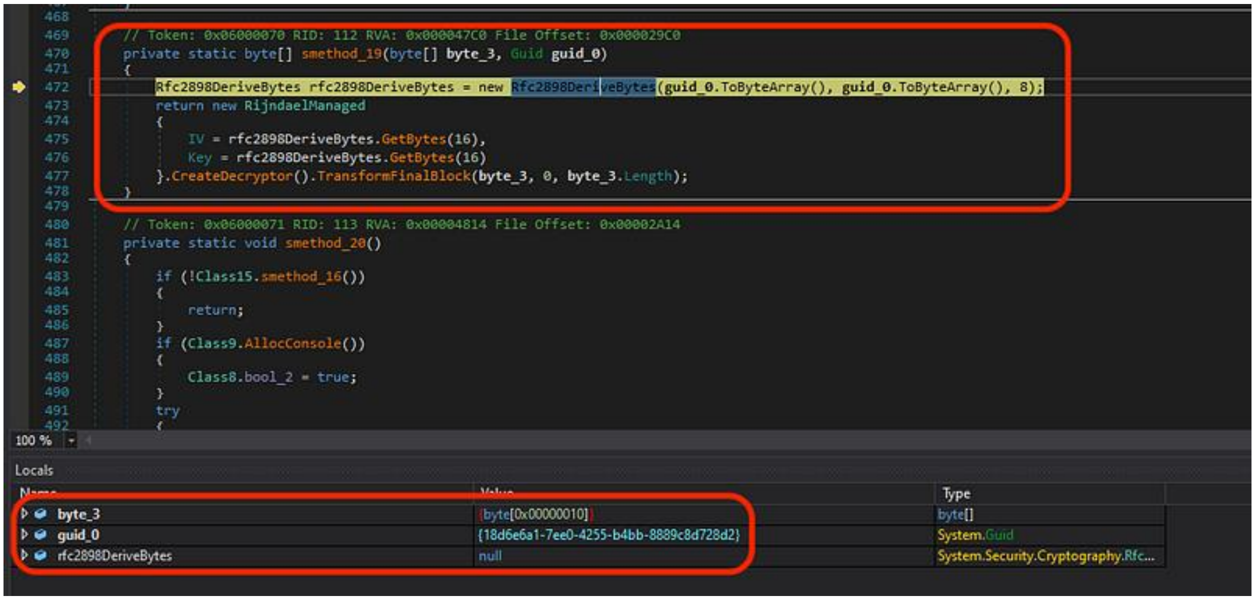


Malicious resource extraction

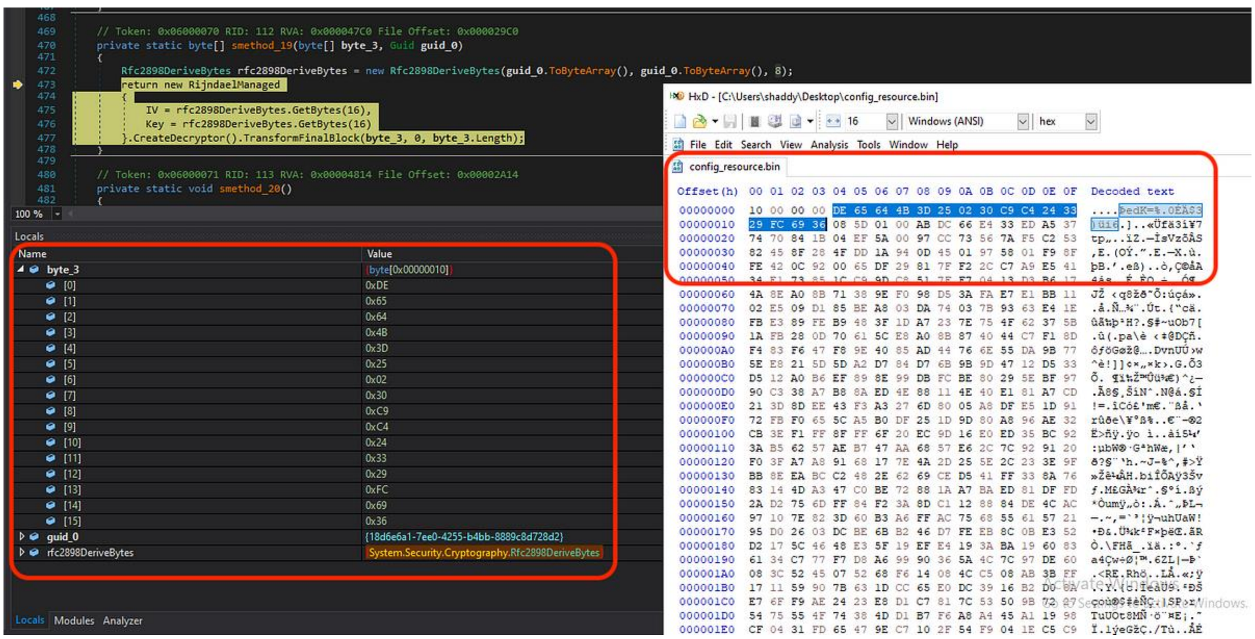
It reads first 4 bytes of this encrypted resource and gets size of decryption key in those 4 bytes from the encrypted resource. It also creates a **GUID** of the executing malicious PE binary and initiates a **decryption** routine to decrypt the key that is used to encrypt rest of the resource.

For example, the first 4 bytes are **10 00 00 00 (0x00000010)**, which in decimal means the value is **16** and that means the encrypted key is next 16 bytes in the encrypted resource. The parameters that are passed to decryption routine are:

- 16 bytes encrypted key
- GUID of itself



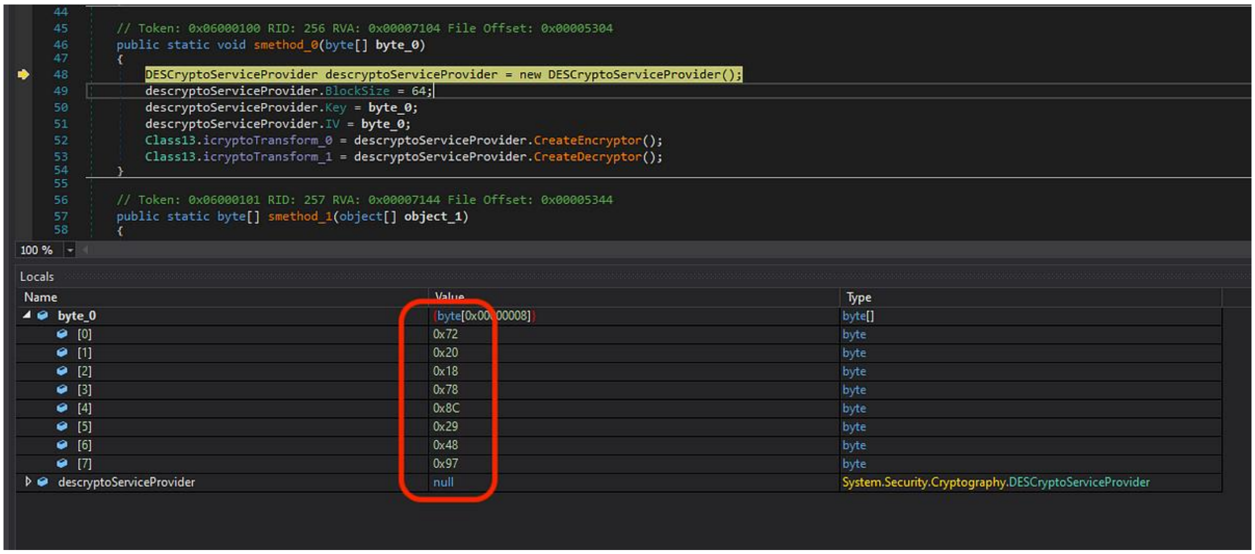
Stage4 decryption routine



Stage4 key decryption

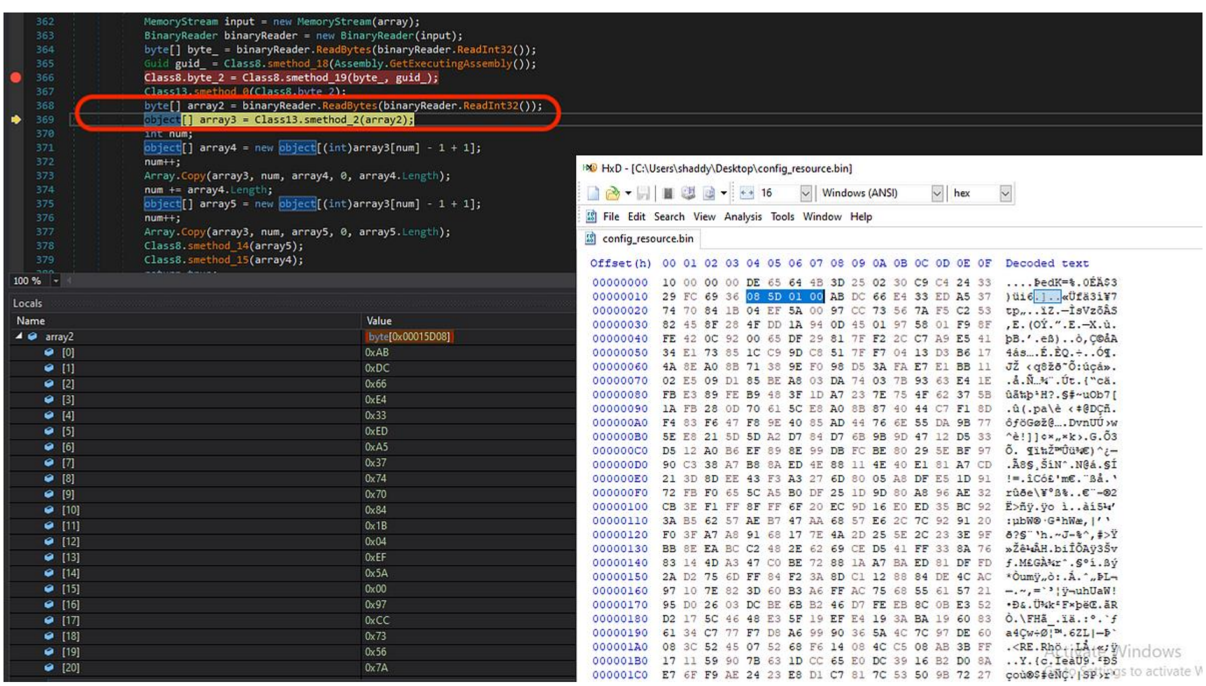
The HxD editor is displayed for easy understanding of how this decryption routine works. In the screenshot above, it is shown that first 4 bytes provides the length of encrypted key bytes that are highlighted. Those key bytes are decrypted using **Rijndael** decyptor and the key for decrypting these bytes is the GUID of malware stage4 binary.

Next, we get the **8-byte decrypted key for DES** encryptor, which is the key used to decrypt rest of the resource. So, the malware uses GUID of itself to decrypt the first 16 bytes (with rijndael) and use the decrypted 8 bytes as key and salt for DES encryption algorithm to decrypt rest of resource. As shown in the screenshot below: it will initiate encryptor and decryptor of DES using the decrypted bytes from the resource file.



Decrypted key stage4

It continues by reading the next 4 bytes and again take it as a parameter of length for reading next number of bytes for DES decryption routine. Next 4 bytes are **15D08** which is equivalent to **89352** number of bytes. Means it is then reading to the end of encrypted resource file.



Resource decryption stage4

Finally, we get the decrypted config file for NanoCore RAT. All the configuration setting are provided below:

There are two dlls that have also been decrypted, that are:

- **ClientPlugin**
- **SurveillanceExClientPlugin**

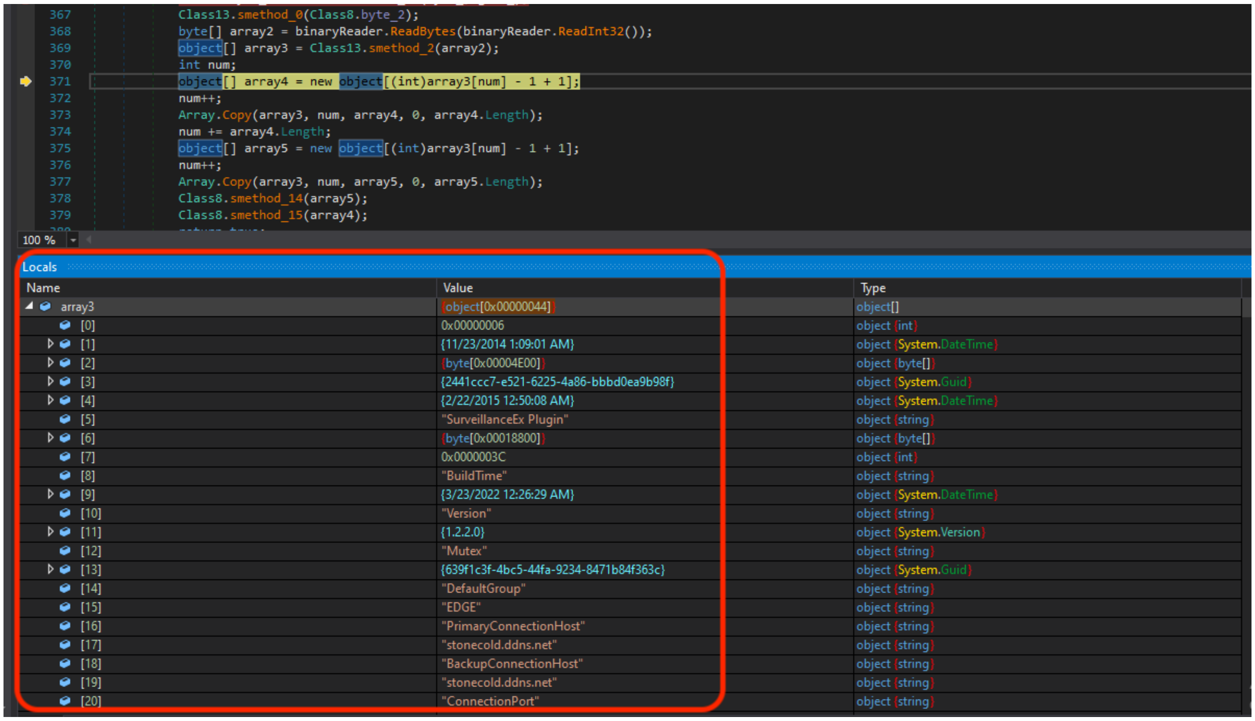
Decrypted resource is divided into two arrays:

- 1st array holds the decrypted binaries (dlls)
- 2nd array holds the configuration settings

Configuration settings:

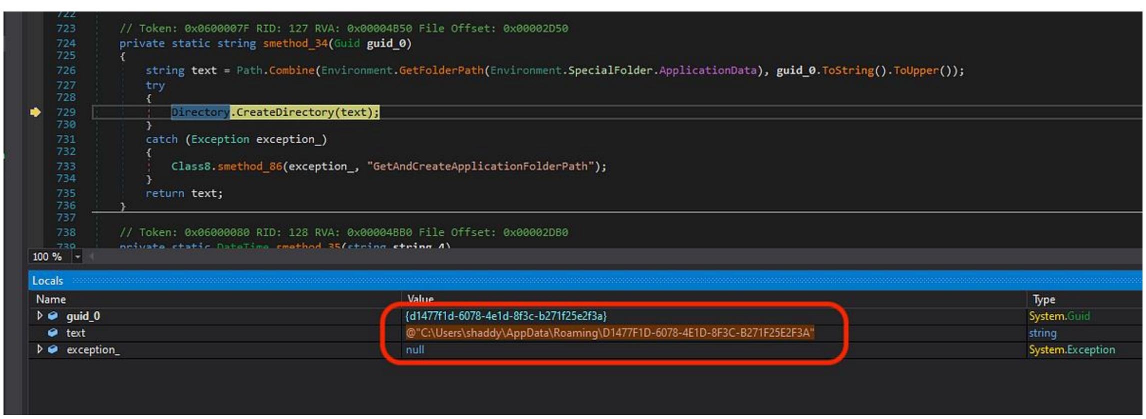
- **BuildTime**: {3/23/2022 12:26:29 AM}
- **Version**: {1.2.2.0}
- **Mutex**: {639f1c3f-4bc5–44fa-9234–8471b84f363c}
- **DefaultGroup**: EDGE
- **PrimaryConnectionHost**: stonecold.ddns.net
- **BackupConnectionHost**: stonecold.ddns.net
- **ConnectionPort**: 0x09C6
- **RunOnStartup**: false
- **RequestElevation**: false
- **BypassUserAccountControl**: false
- **ClearZoneIdentifier**: true
- **ClearAccessControl**: false
- **SetCriticalProcess**: false
- **PreventSystemSleep**: true
- **ActivateAwayMode**: false
- **EnableDebugMode**: false
- **RunDelay**: 0x00000000
- **ConnectionDelay**: 0x00000FA0
- **RestartDelay**:0x00001388
- **TimeoutInterval**: 0x00001388
- **KeepAliveTimeout**: 0x00007530
- **MutexTimeout**: 0x00001388
- **LanTimeout**: 0x000009C4
- **WanTimeout**: 0x00001F40
- **BufferSize**: 0x0000FFFF
- **MaxPacketSize**: 0x00A00000

- **GCThreshold**: 0x00A00000
- **UseCustomDnsServer**: true
- **PrimaryDnsServer**: 8.8.8.8
- **BackupDnsServer**: 8.8.4.4



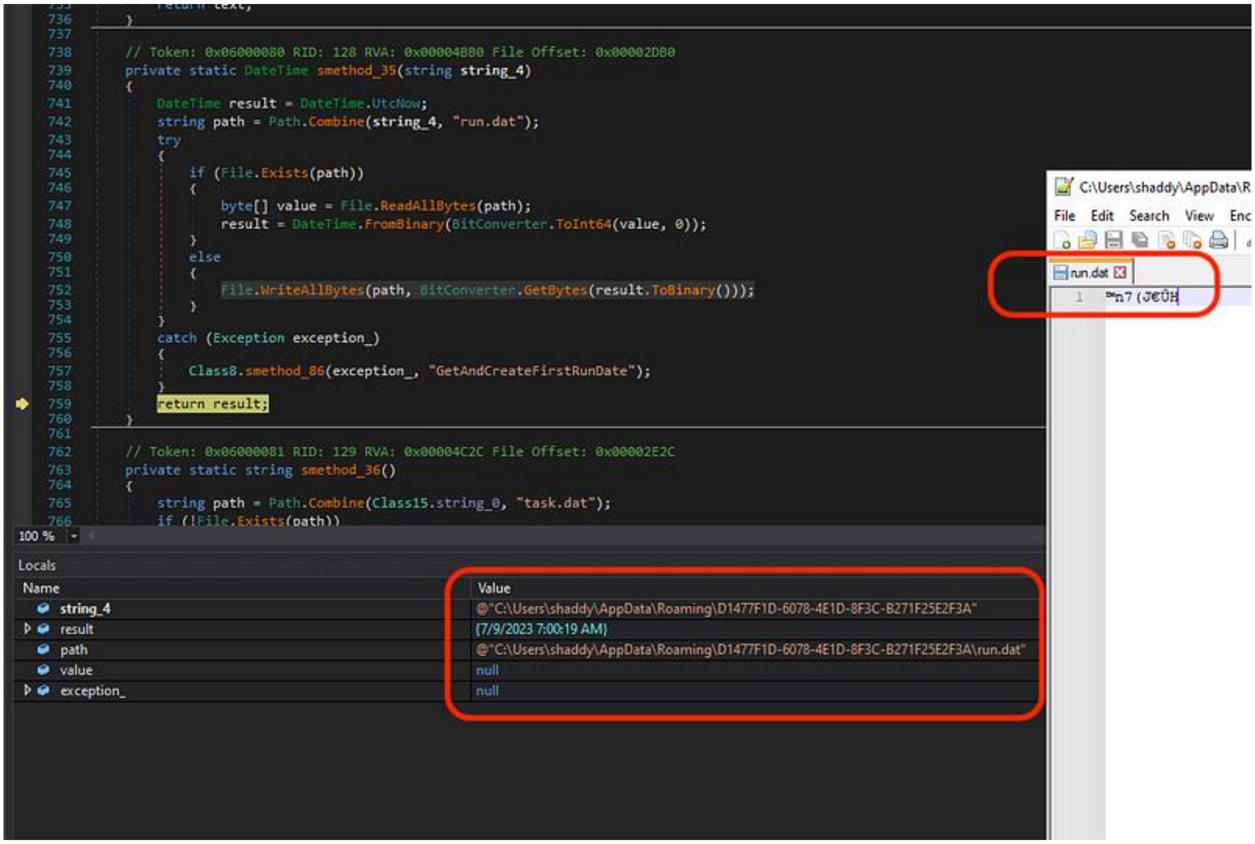
Decrypted RAT configuration

The malware adjusts its settings based on the configuration file above and then performs a series of steps as provided in RAT configuration. It then moves on to create mutex, queries the machine GUID from registries and create a folder in %appdata% with machine GUID value. This folder is the main working directory of malware.



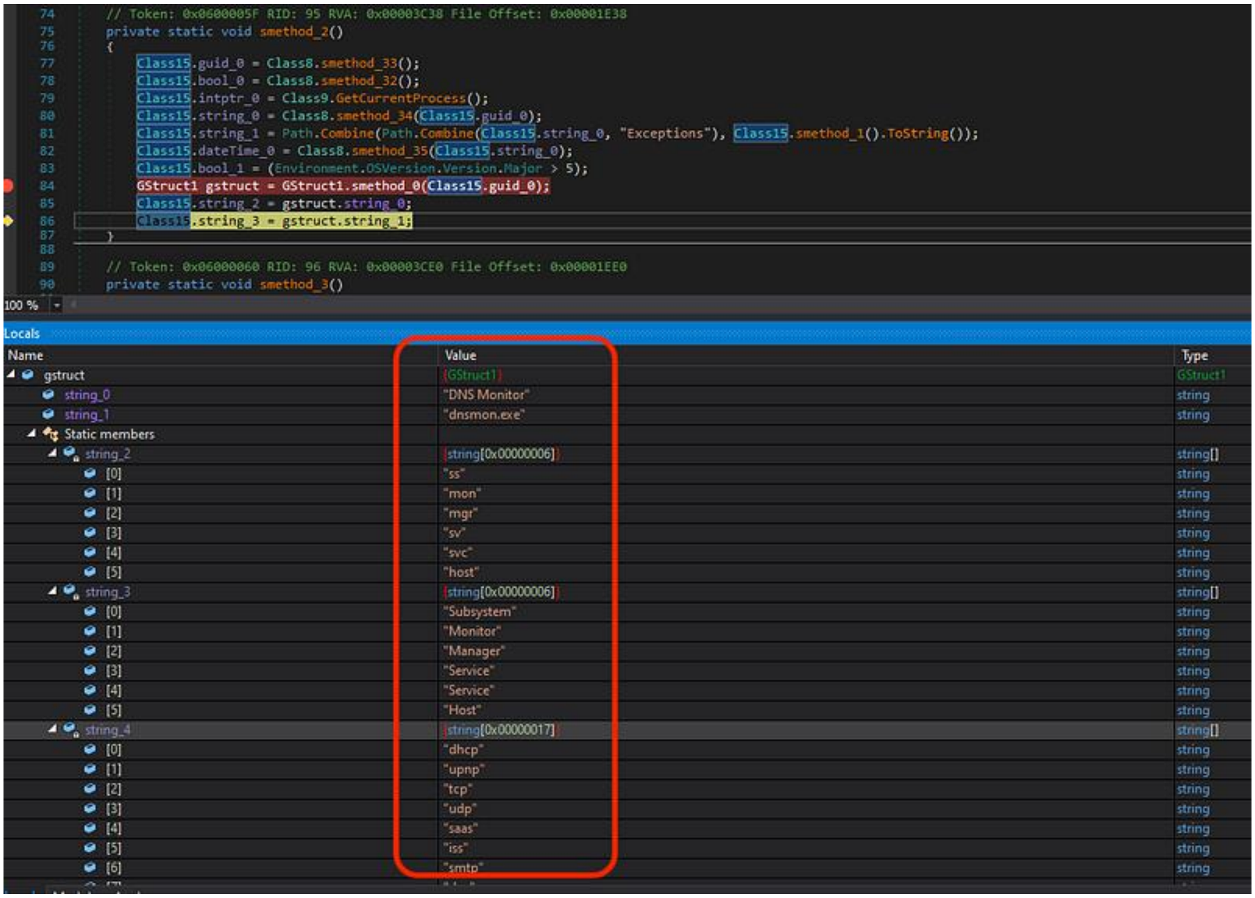
NanoCore working directory

One of the indicators that I found above, which is the creation of a "run.dat" file in the system is achieved in the next method. It gets current DateTime and save those values as bytes in **Run.dat** file. This might be used as an indicator for when the infection started in the particular system. Also, I am assuming the value of run.dat is being sent as **heartbeat** packet to the c2 server.



Indicator of NanoCore

Malware is totally dynamic. It sets up most of the strings at run-time for the malicious files. It combines different strings dynamically to avoid detection. The malware has pre-defined values in its structures based on the LOL bins (living of the land binaries) names and paths. It combines these values at run-time and sets up its malicious files and processes masquerading as windows native binaries.



LOL bins masquerading

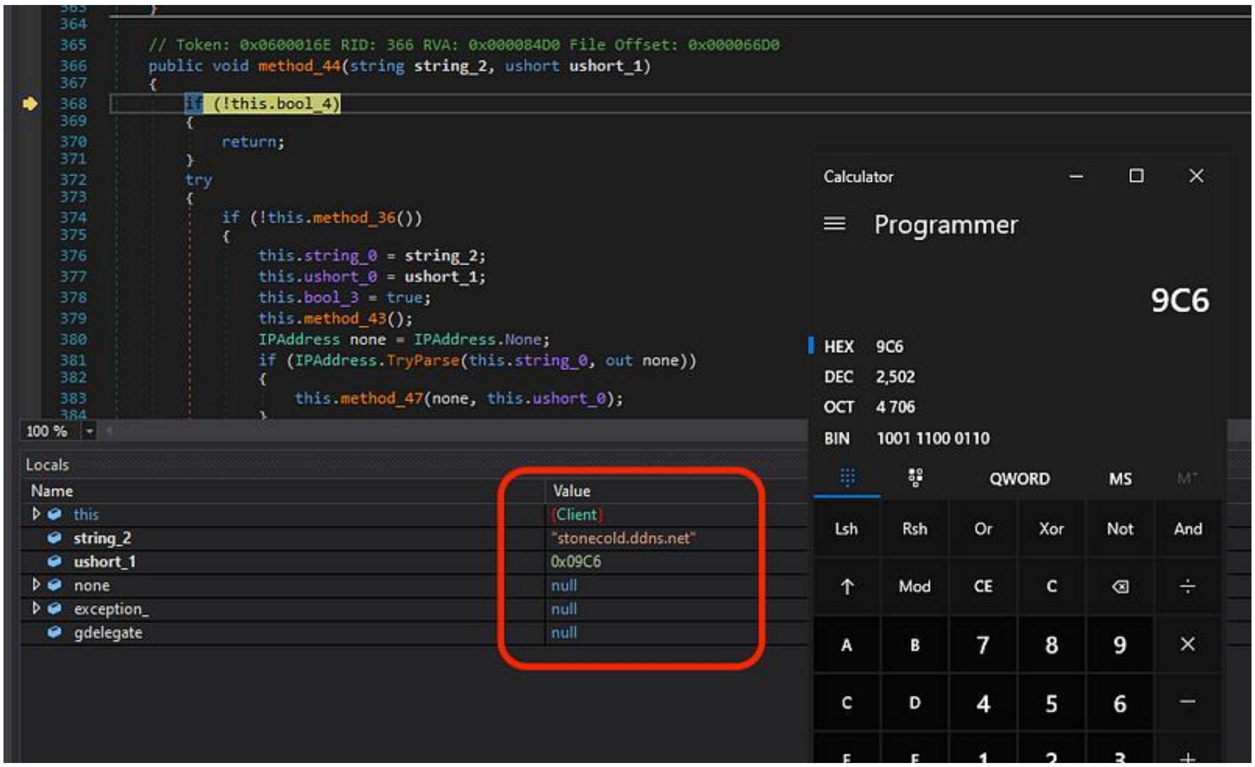
In the screenshot above, it is visible that the malware picked **DNS Monitor** and **dnsmon.exe** from the structures that are available. Next time it could pick **NTFS Manager** and **ntfsmgr.exe** as the next target.

In this sample, the RAT doesn't have everything enabled in its configuration. Therefore, it skips most of the really critical steps:

- RunOnStartup: false
- RequestElevation: false
- BypassUserAccountControl: false
- ClearZoneIdentifier: true
- ClearAccessControl: false
- SetCriticalProcess: false
- PreventSystemSleep: true
- ActivateAwayMode: false
- EnableDebugMode: false

All of the above-mentioned steps are being skipped as I further debug the malware. I later patched the malware to execute these steps as well for TTP extraction process, which i will discuss later on.

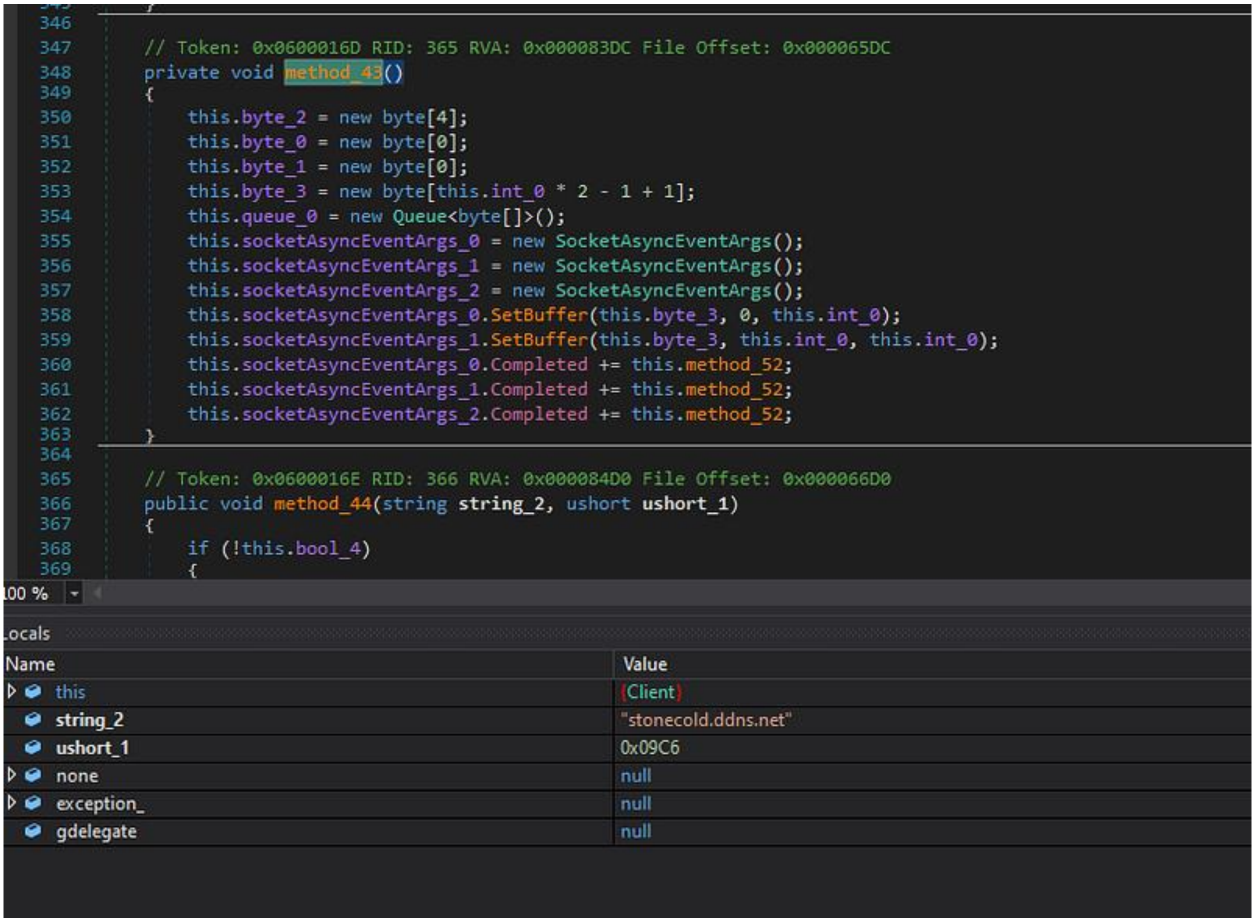
I debugged the code further. There were so many dynamic changes, like setting variable values, setting the plugins, setting Client Connection values, The connection IPs, the timeout values and much more. Finally, it was able to configure all settings and resolve the C2 server. The Domain name and the port number are being resolved to create the connection. Port number is **2502** and C2 server is **stonecold.ddns.net**.



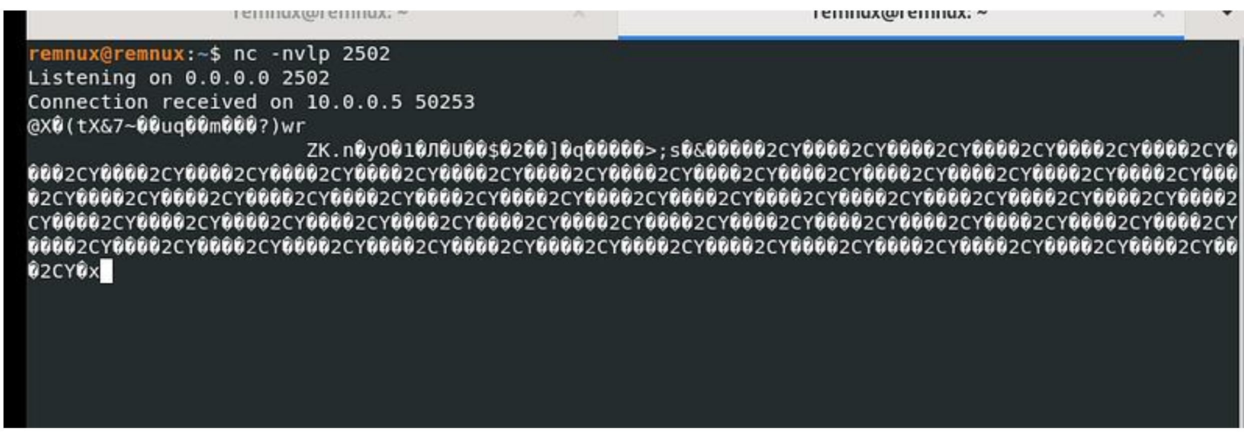
Resolving c2 server

Creates and establishes asyn sockets for the connection. Since all the code is dynamic therefore the values are being received from different methods. Then it forwards the program to asynchronously send **heartbeat** messages to the c2 server again and again until the connection is created. The c2 server is down, therefore the malware doesn’t move forward with its execution.

Using the internet simulator, we can fool the malware by showing c2 server as live, but it has some sort of authentication mechanism in place and waits for sever response to create socket. I used netcat to listen on the specified port and it keeps sending heartbeat packets as shown:



Async sockets



Netcat listening on malicious port

So C2 server is basically a DuckDNS domain. Duck DNS is a free Dynamic DNS service that associates domain names with changing IP addresses, primarily used for legitimate purposes like remote access to devices. However, malicious actors can exploit it for command and control (C2) in malware. They do this to hide the C2 server's location, maintain anonymity, evade detection, and quickly adapt to takedowns.

TTP Extraction

My work is related to TTP extraction and recreation process after the initial analysis. The project that i am working on is Breach and attack simulation and my job is to enrich its threat library with latest malware recreated in a safe exploitation manner for security testing.

From NanoCore i have identified these TTPs in my initial analysis:

1. Defense Evasion: Obfuscated Files or Information: Embedded Payloads
2. Defense Evasion: Obfuscated Files or Information: Dynamic API Resolution
3. Defense Evasion: Process Injection: Process Hollowing
4. Persistence: Boot or Logon Autostart Execution: Registry Run keys/startup folder
5. Defense Evasion: Hide Artifacts: Resource Forking
6. Defense Evasion: Subvert Trust Controls: Mark-of-the-web Bypass
7. Privilege Escalation: Scheduled Task/Job: Scheduled Task
8. Defense Evasion: Files and Directory Permissions Modifications: Windows File and Directory Permissions Modifications
9. Defense Evasion: Masquerading: Masquerade Task or Service
10. Defense Evasion: Hide Artifacts: Hidden Window
11. Command and Control: Non-Application Layer Protocol
12. Collection: Input Capture: Keylogging
13. Collection: Clipboard Data
14. Collection: Automated Collection
15. Exfiltration: Exfiltration over C2 channel

NanoCore SurveillanceExClientPlugin

Another dynamic link library that has been decrypted from the resources and being used for spying on victim is called the **SurveillanceExClientPlugin**. I dumped this module separately for static analysis and found very exciting and organized malicious code used for spying and logging user's activity.

The SurveillaneExClientPlugin does following:

- **Extracts further resources**: **Lzma** and **TLD,** first one is a custom Lzma compression plugin and the other one is Undefined
- **Process Hollowing**: There is a whole section of process hollowing code inside surveillance plugin

- **Keylogging**: Organized code for recording all types of data, including keys, clipboards, dns records etc
- **C&C**: Executes basic commands like enabling/disabling keylogging, application logging, dnslogging, get logs, delete logs, export or view logs.
- **Exfiltration**: Recorded logs are exfiltrated over to different hosts defined by malware dynamically

I have recreated most of the keylogging code used by NanoCore. It is registering a RAW input device and receives RAW input data, then maps those RAW inputs to unicode characters and logs it in a .dat file. A chunk of the simplified code is uploaded alongside this report.

Similarly, the DNS records are being logged by using the API of **DNSGetCacheDataTable**. I've created multiple test cases for each TTP listed above. However, for **security purposes** and to avoid the abuse of my code, I will not post it publicly.

In conclusion, the detailed analysis of the NanoCore Remote Access Trojan (RAT) underscores the evolving sophistication of malicious tools in the digital landscape. NanoCore RAT's multifaceted capabilities, including remote control, keylogging, file manipulation, and data exfiltration, make it a potent threat to both individuals and organizations. However, traditional signature-based detection methods often fall short in identifying such polymorphic malware due to its ability to quickly morph and evade detection.

This analysis emphasizes the urgent need for behavioral detection mechanisms in modern cybersecurity strategies. Behavioral detection, powered by machine learning and artificial intelligence, focuses on identifying patterns of behavior rather than relying solely on known signatures. This approach enables security systems to adapt and recognize novel threats like NanoCore RAT, even as they evolve to avoid traditional defenses. By continually monitoring and analyzing system behavior, security solutions equipped with behavioral detection can provide a proactive defense, offering a crucial layer of protection against emerging threats that traditional methods may miss. As cyber adversaries continue to innovate, embracing behavioral detection becomes imperative to stay one step ahead and safeguard digital assets effectively.