Teaching an old RAT new tricks

Posted on April 21, 2016 by Joseph Landry

Attackers have been successfully deploying RATs for years to remotely control users systems - giving them full access to the victim's files or resources such as cameras, recording key strokes, or downloading further malware. Traditionally RATs have been deployed when a user opens an email attachment, or downloads a file from a website or peer-to-peer network. In both cases, these vectors involve use of files to deliver the payload - which are easier to detect.

Recently we detected a more sophisticated technique that a handful of countries across Asia are actively using to infect systems with RATs. This new technique ensures that the payload/file remains in memory through its execution, never touching the disk in a de-encrypted state. In doing so, the attacker can remain out of view from antivirus technologies, and even 'next-generation' technologies that only focus on file-based threat vectors. Also, the samples analyzed have the ability detect the presence of a virtual machine to ensure it's not being analyzed in a network sandbox.

And finally it's important to highlight that the RAT itself is not new. In fact this technique can be used to deliver any "known" RAT to a victim's system. We analyzed this sample against our SentinelOne EPP to confirm is does not evade our behavior-based detection mechanisms. This is due to the fact that we're monitoring all processes at the user-space/kernel-space interface - and because all communication between the application and the kernel must be unencrypted, we detect the sample at both process-injection points.

Samples Analyzed

Main Sample

- **Format: Win32 PE .NET 2.0**
- **SHA-256 sum: b7cfc7e9551b15319c068aae966f8a9ff563b522ed9b1b42d19c122778e018c8**
- **HSA-1 sum: 3b1ac573509281cdc0b6141f8ea6ed3af393b554**
- **MD5 sum: 65752e742d643d121ee7e826ab65dc9b**
- **File size: 321024 bytes (324 kb)**

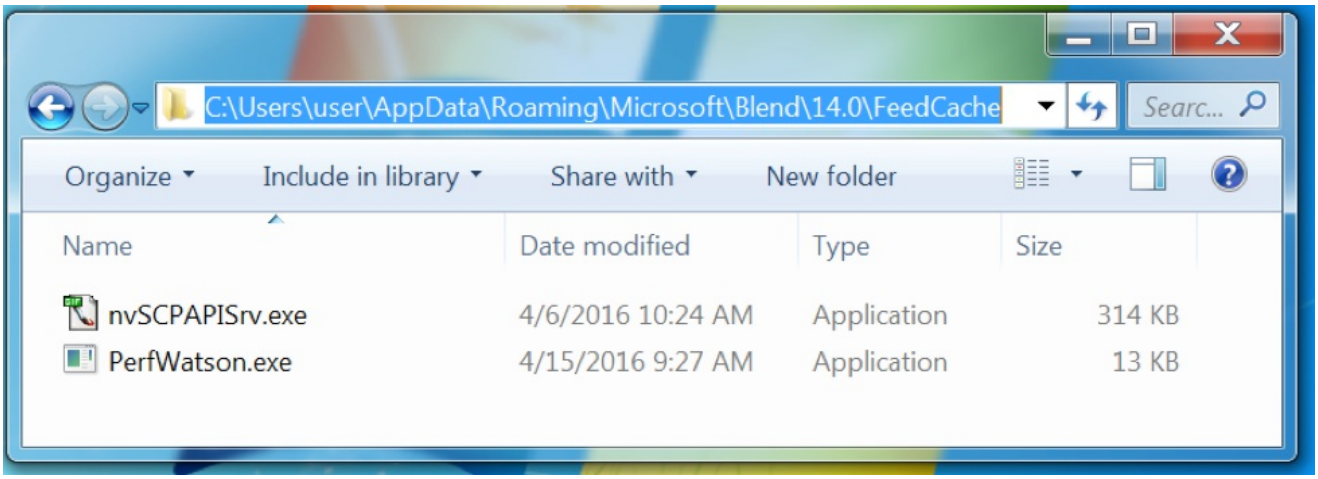
Unpacked Samples

- **Main Sample**

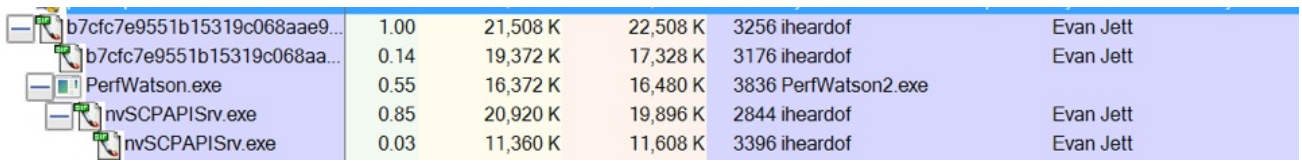
○ e5c71180f117270538487cd9b9b1b6d8 - Packed "Benchmark" DLL ▪ 9e05fb115bd4e85cfc0e32c72aa721be - Monitor (PerfWatson.exe) ○ d740ed3f33ca4cef3a6aa717f94bf52a - NanoCore RAT dumped from memory

Behavioral Analysis

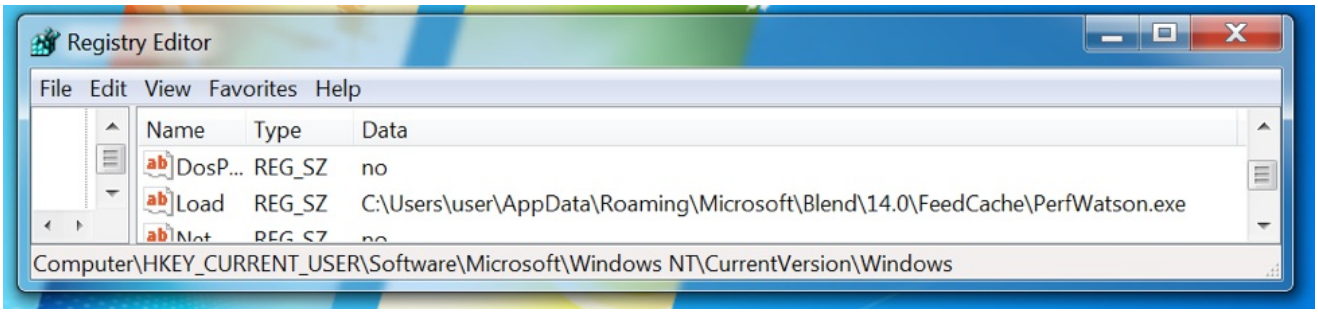
When run, the binary will copy itself to %APPDATA%\Microsoft\Blend\14.0\FeedCache\nvSCPAPISrv.exe and extracts a second binary named PerfWatson.exe



It then executes both binaries.



For persistence, a registry key is created at HKEY_CURRENT_USER\Software\Microsoft\Windows NT\CurrentVersion\Windows\Load pointing to the PerfWatson.exe binary.



Finally, the RAT tries connecting back to its control server:

- azona2015.chickenkiller.com:1617 (TCP)
- azona.chickenkiller.com:1617 (TCP)

chickenkiller.com is owned by a free dynamic DNS service.

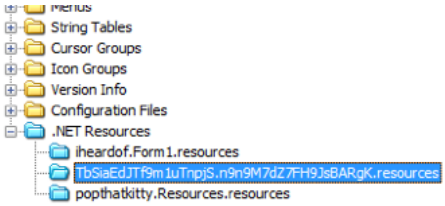
At the time of this writing, the DNS records still exist, but the address they resolve to appears to be down.

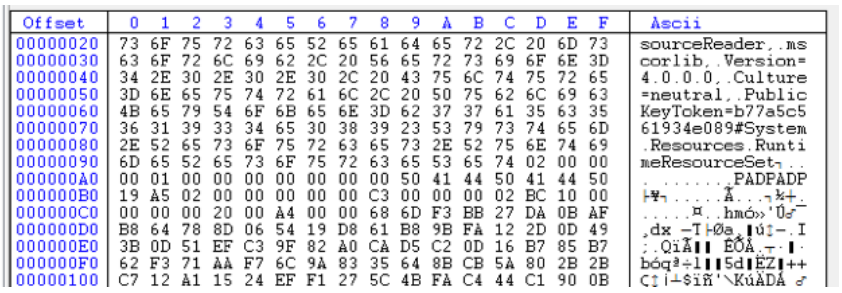
```
60 105.753484   172.16.108.128   8.8.8.8          DNS      87 Standard query 0x0be8 A azona2015.chickenkiller.com
61 105.822747   8.8.8.8          172.16.108.128   DNS     103 Standard query response 0x0be8 A azona2015.chickenkiller.com A 154.66.17.101
62 105.823277   172.16.108.128   154.66.17.101    TCP      66 49177 → 1617 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
63 108.841210   172.16.108.128   154.66.17.101    TCP      66 [TCP Retransmission] 49177 → 1617 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
64 109.566886   154.66.17.101    172.16.108.128   TCP      60 1617 → 49173 [RST, ACK] Seq=1 Ack=1 Win=64240 Len=0
```

Unpacking

"Benchmark" .NET DLL

The main executable contains an XOR encrypted .NET DLL in its .NET managed resources and the logic to unpack it. This DLL contains the logic to unpack and inject the RAT as well as monitor the application, PerfWatson.exe. This DLL is referred to as "Benchmark" because that is the .NET namespace it uses.





Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Ascii
00000020	73	6F	75	72	63	65	52	65	61	64	65	72	2C	20	6D	73	sourceReader, ms
00000030	63	6F	72	6C	69	62	2C	20	56	65	72	73	69	6F	6E	3D	corlib, Version=
00000040	34	2E	30	2E	30	2E	30	2C	20	43	75	6C	74	75	72	65	4.0.0.0, Culture
00000050	3D	6E	65	75	74	72	61	6C	2C	20	50	75	62	6C	69	63	=neutral, Public
00000060	4B	65	79	54	6F	6B	65	6E	3D	62	37	37	61	35	63	35	KeyToken=b77a5c5
00000070	36	31	39	33	34	65	30	38	39	23	53	79	73	74	65	6D	61934e089#System
00000080	2E	52	65	73	6F	75	72	63	65	73	2E	52	75	6E	74	69	.Resources.Runti
00000090	6D	65	52	65	73	6F	75	72	63	65	53	65	74	02	00	00	meResourceSet..
000000A0	00	01	00	00	00	00	00	00	00	50	41	44	50	41	44	50	.....PADPADP
000000B0	19	A5	02	00	00	00	00	00	C3	00	00	00	02	BC	10	00	¥.....Ã.....¼.
000000C0	00	00	00	20	00	A4	00	00	68	6D	F3	BB	27	DA	0B	AF	....¤..hmó»'Ú¯
000000D0	B8	64	78	8D	06	54	19	D8	61	B8	9B	FA	12	2D	0D	49	,dx..T.Øa,.ú.-.I
000000E0	3B	0D	51	EF	C3	9F	82	A0	CA	D5	C2	0D	16	B7	85	B7	;.QïÃ.. ÊÕÂ.¬·.·
000000F0	62	F3	71	AA	F7	6C	9A	83	35	64	8B	CB	5A	80	2B	2B	bóq÷l..5d.ËZ.++
00000100	C7	12	A1	15	24	EF	F1	27	5C	4B	FA	C4	44	C1	90	0B	Ç.¡.$ïñ'\KúÄDÁ..

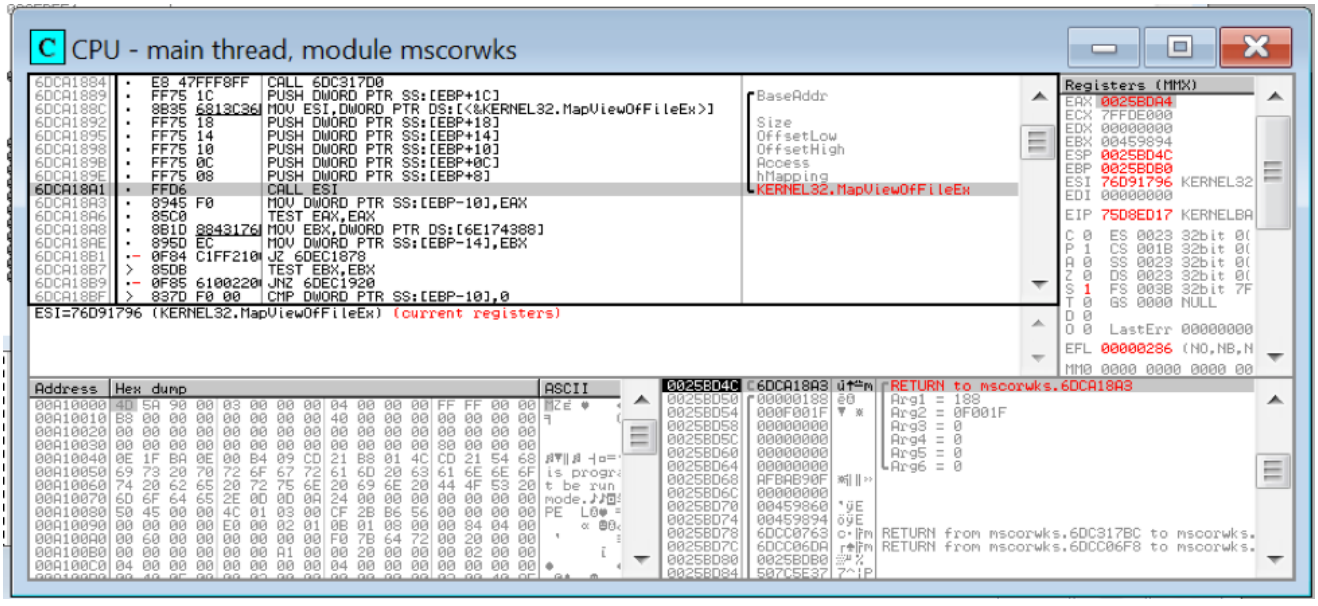
After decrypting the resource, it is linked into the process using System.Reflection.Assembly.Load(byte[]). This method is documented on MSDN here. Using this method, the DLL will never be written to the filesystem. This technique could have been chosen by the developers to evade antivirus detection.

```
internal static object LoadAssembly(object object_0)
{
    return Assembly.Load(object_0);
}
```

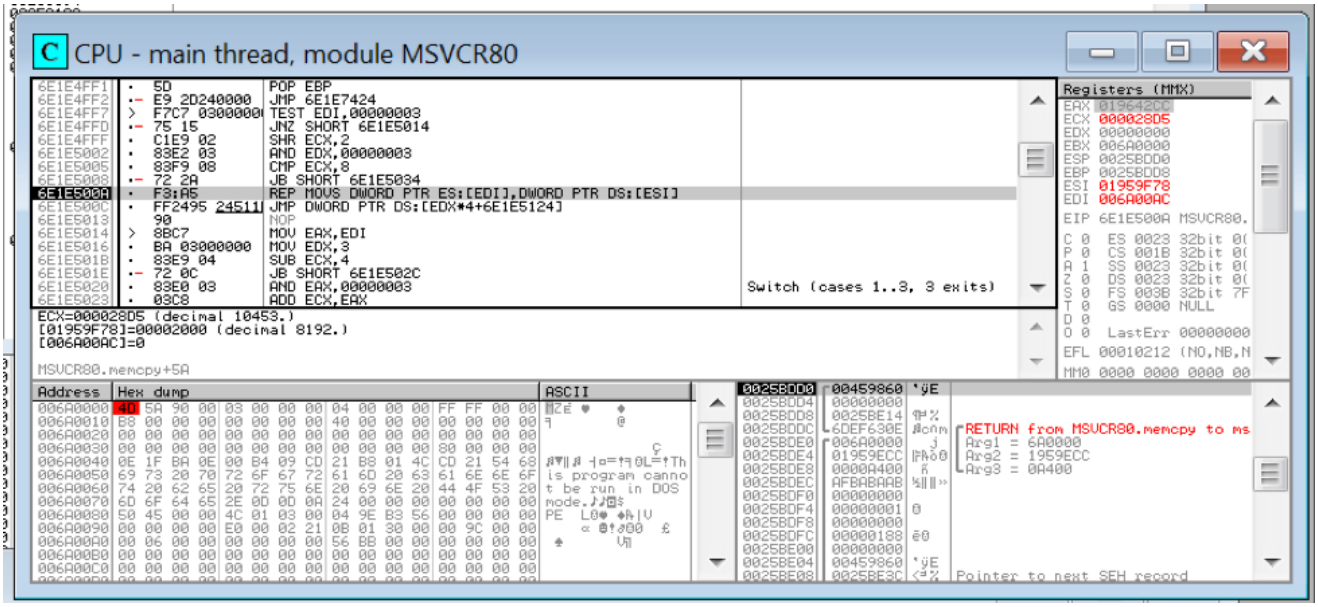
Under the hood, Assembly.Load(), uses a call to the win32 api call CreateFileMappingW() with the hFile parameter set to INVALID_HANDLE_VALUE. According to MSDN, this will create a mapped file that is backed by the paging filesystem, not the standard filesystem. A layer below CreateFileMapping, the system call NtCreateSection is invoked.

```
Stack     Data      Procedure                  Called from
0025BD4C  75D8A276  ntdll.NtCreateSection      KERNELBASE.CreateFileMappingW+0A5
0025BD50  0025BDC0    Arg1 = 25BDC0
0025BD54  000F0007    Arg2 = 0F0007
0025BD58  00000000    Arg3 = 0
0025BD5C  0025BD98    Arg4 = 25BD98
0025BD60  00000004    Arg5 = 4
0025BD64  08000000    Arg6 = 8000000
0025BD68  00000000    Arg7 = 0
0025BDA8  6DCA1878  KERNEL32.CreateFileMappingW  mscorwks.6DCA1872
0025BDAC  FFFFFFFF    hFile = INVALID_HANDLE_VALUE
0025BDB0  00000000    pSecurity = NULL
0025BDB4  00000000    Protect = 0
0025BDB8  00000000    MaxSizeHigh = 0
0025BDBC  0000A400    MaxSizeLow = 0A400
0025BDC0  00000000    Name = NULL
0025BDD0  6DEF6258  mscorwks.6DCA184D          mscorwks.6DEF6253
0025BDD4  FFFFFFFF    Arg1 = -1
0025BDD8  00000000    Arg2 = 0
0025BDDC  00000004    Arg3 = 4
0025BDE0  00000000    Arg4 = 0
0025BDE4  0000A400    Arg5 = 0A400
0025BDE8  00000000    Arg6 = 0
0025BE18  6DEF6481  mscorwks.6DEF61C9          mscorwks.6DEF647C
```

After the empty file is created, it is mapped into memory using the Win32 API call MapViewOfFileEx. The layer below this invokes the system call NtMapViewOfSection.



Now, a call to memcpy() is used to copy the decrypted DLL into the newly allocated address range.

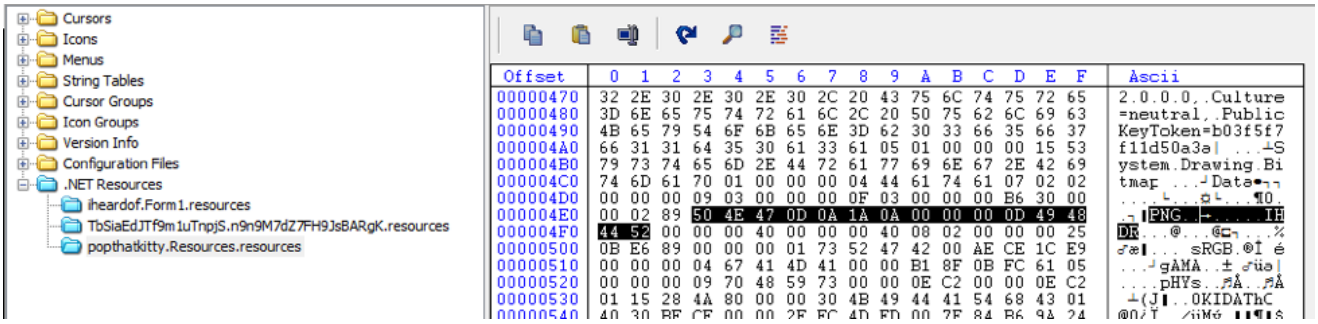


Unpacking Settings and NanoCore

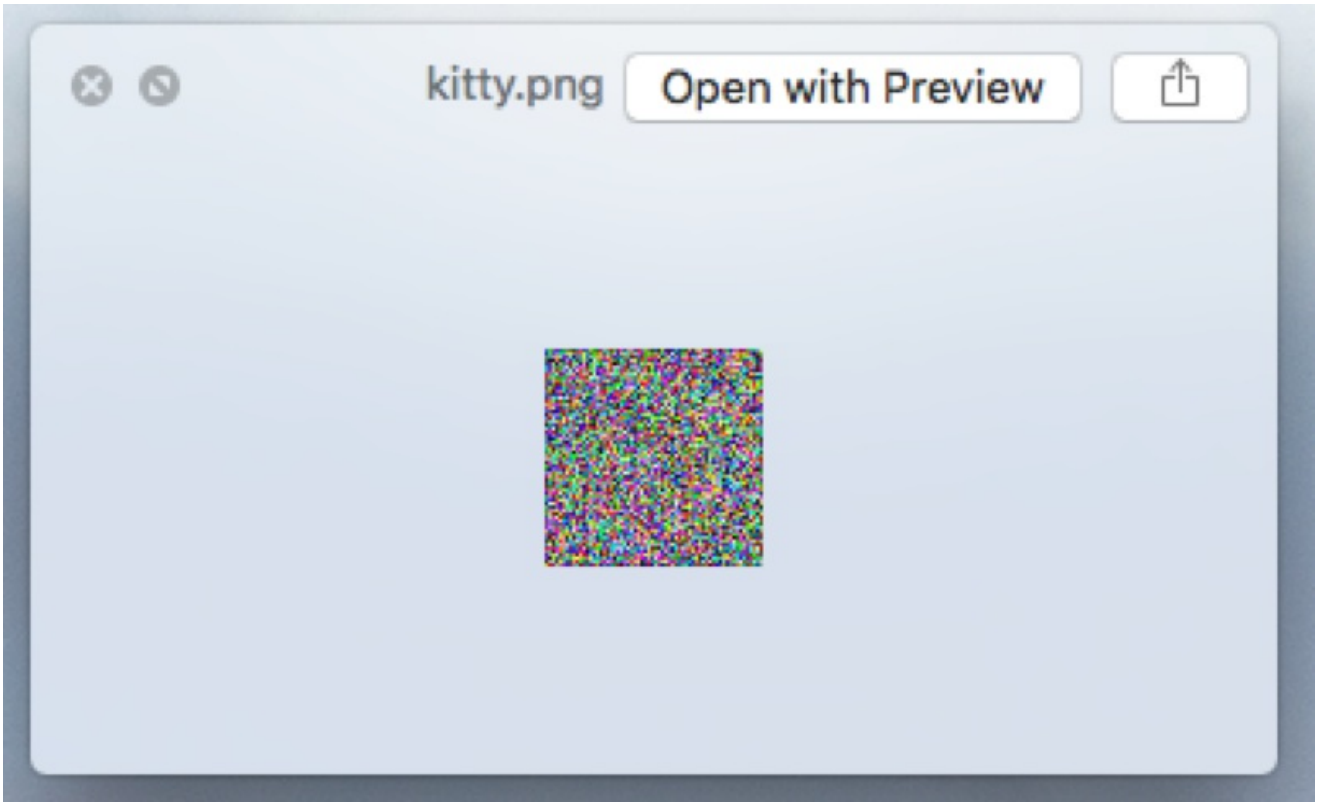
The settings for "Benchmark" and the NanoCore executable are serialized, DES encrypted, spliced, and stored across multiple PNG files as pixel data. The PNG files are concatenated and stored in the .NET managed resources of the main executable.

Some of the settings that can be configured are:

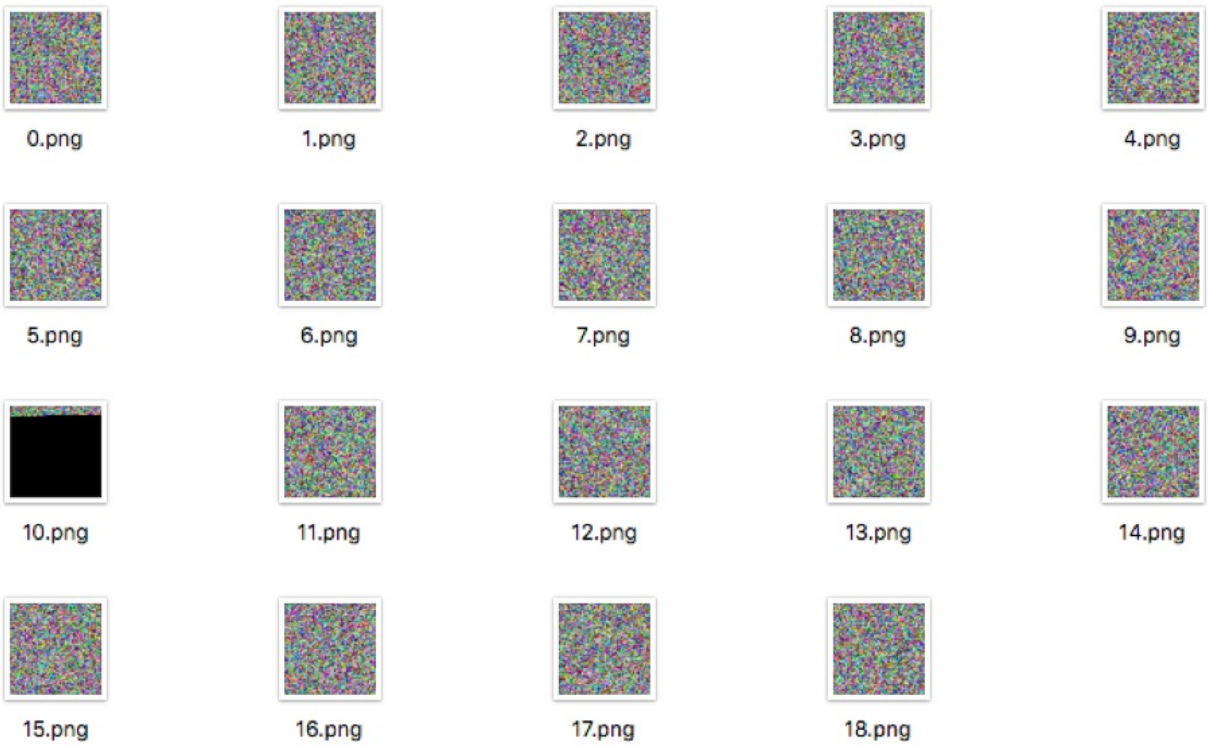
- **Exit if a virtual machine is detected**
- **Paths and filenames to store PerfWatson.exe and NanoCore**
- **Display a message box to the user**
- **Delete ":Zone.Identifier" information for files from NTFS ADS.**
- **Download an encrypted file from the Internet, decrypt it, and run it.**
- **Monitor the Injected process**



After viewing one of these images, it is obvious they are not used to conveying visual information to a human eye.



After writing a short python script, I was able to extract all 19 PNG files. If you have robot eyes, you can see a cat.



Here is a C# decompilation of the method used to extract the information out of the pixel data.

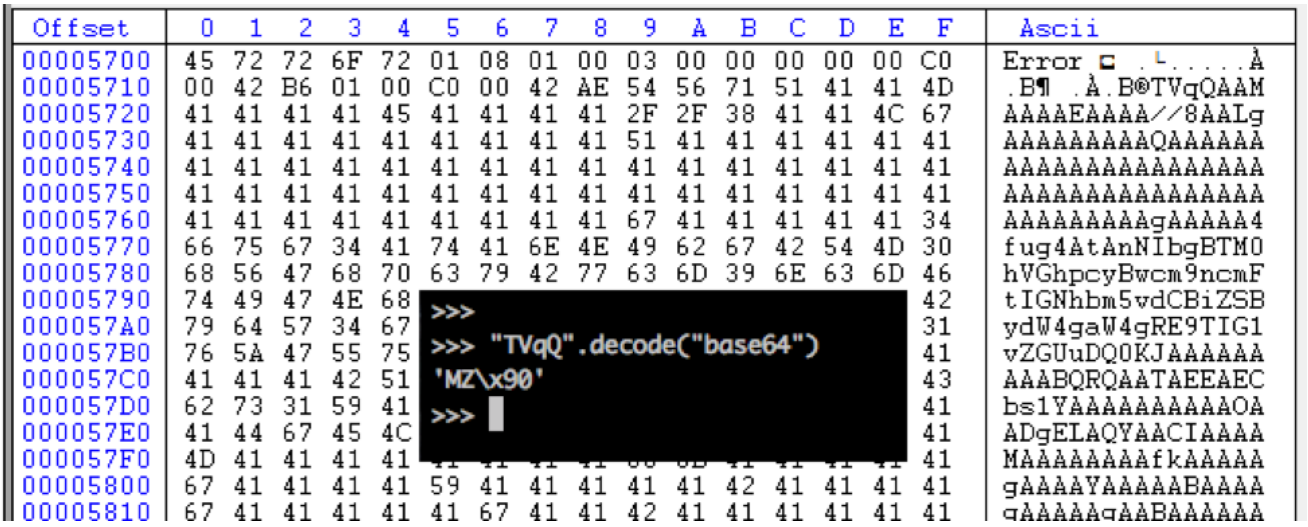
```
// Benchmark.EntryPoint
internal static byte[] ImagesToData(Image[] images)
{
    MemoryStream memoryStream = new MemoryStream();
    for (int i = 0; i < images.Length; i++)
    {
        Image image = images[i];
        Rectangle rect = new Rectangle(Point.Empty, image.Size);
        MemoryStream memoryStream2 = new MemoryStream();
        image.Save(memoryStream2, ImageFormat.Bmp);
        Bitmap expr_40 = new Bitmap(memoryStream2);
        BitmapData bitmapData = expr_40.LockBits(rect, ImageLockMode.ReadOnly, PixelFormat.Format24bppRgb);
        byte[] array = new byte[Marshal.ReadInt32(bitmapData.Scan0)];
        Marshal.Copy(new IntPtr(bitmapData.Scan0.ToInt32() + 4), array, 0, array.Length);
        expr_40.UnlockBits(bitmapData);
        memoryStream2.Close();
        memoryStream.Write(array, 0, array.Length);
    }
    memoryStream.Close();
    return memoryStream.ToArray();
}
```

Once everything is decrypted, the set options are executed, and the NanoCore RAT payload is injected into a new child process. The method of injection is discussed later.

Unpacking PerfWatson.exe

Now that "Benchmark" is loaded into memory, it is tasked with copying the main executable and extracting PerfWatson.exe to %APPDATA%\\Microsoft\Blend\14.0\FeedCache\.

PerfWatson.exe is stored inside Benchmark as a base64 encoded string. There is no encryption or obfuscation outside of the base64 encoding.



Inside the .NET assembly, the string is stored as a DefaultSettingValue string. The developers might have used this as a way to conceal the meaning of this long string.

```
[DefaultSettingValue("TVqQAAMAAAAEAAAA//8AALgAAAAAAAAAQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAgAAAAA4fug4AtAnN
clIAbDhoV0ZKUThGAG9qZVpUWlNuNQBPRTdWTkRGTUoARmxmSDBFbGp2AE5KaTNxcFdhaABVNjM2SUlzQmMAbmpXQ0FqVjNiAG80c1k1UDdMVgBFdmVudF
```

Once the string is decoded, it is written to disk and executed.

```
Address  | Stack    | Procedure / arguments
0021E758 | 76E35186 | KERNEL32.CreateProcessW
0021E75C | 0428453C |   ModuleFileName = "C:\Users\user\AppData\Roaming\Microsoft\Blend\14.0\FeedCache\PerfWatson.exe"
0021E760 | 04D06990 |   CommandLine = ""C:\Users\user\AppData\Roaming\Microsoft\Blend\14.0\FeedCache\PerfWatson.exe" "
0021E764 | 00000000 |   pProcessSecurity = NULL
0021E768 | 00000000 |   pThreadSecurity = NULL
0021E76C | 00000000 |   InheritHandles = FALSE
0021E770 | 04080410 |   CreationFlags = CREATE_NEW_CONSOLE|CREATE_UNICODE_ENVIRONMENT|CREATE_DEFAULT_ERROR_MODE|80000
0021E774 | 00000000 |   pEnvironment = NULL
0021E778 | 04281188 |   CurrentDir = "C:\Users\user\Desktop"
0021E77C | 0021E7C0 |   pStartupInfo = 0021E7C0
0021E780 | 04282888 |   pProcessInfo = 04282888
0021E850 | 76E42C34 | ? shell32.76E34FBC
0021E8A4 | 76E34F65 | shell32.76E42B75
```

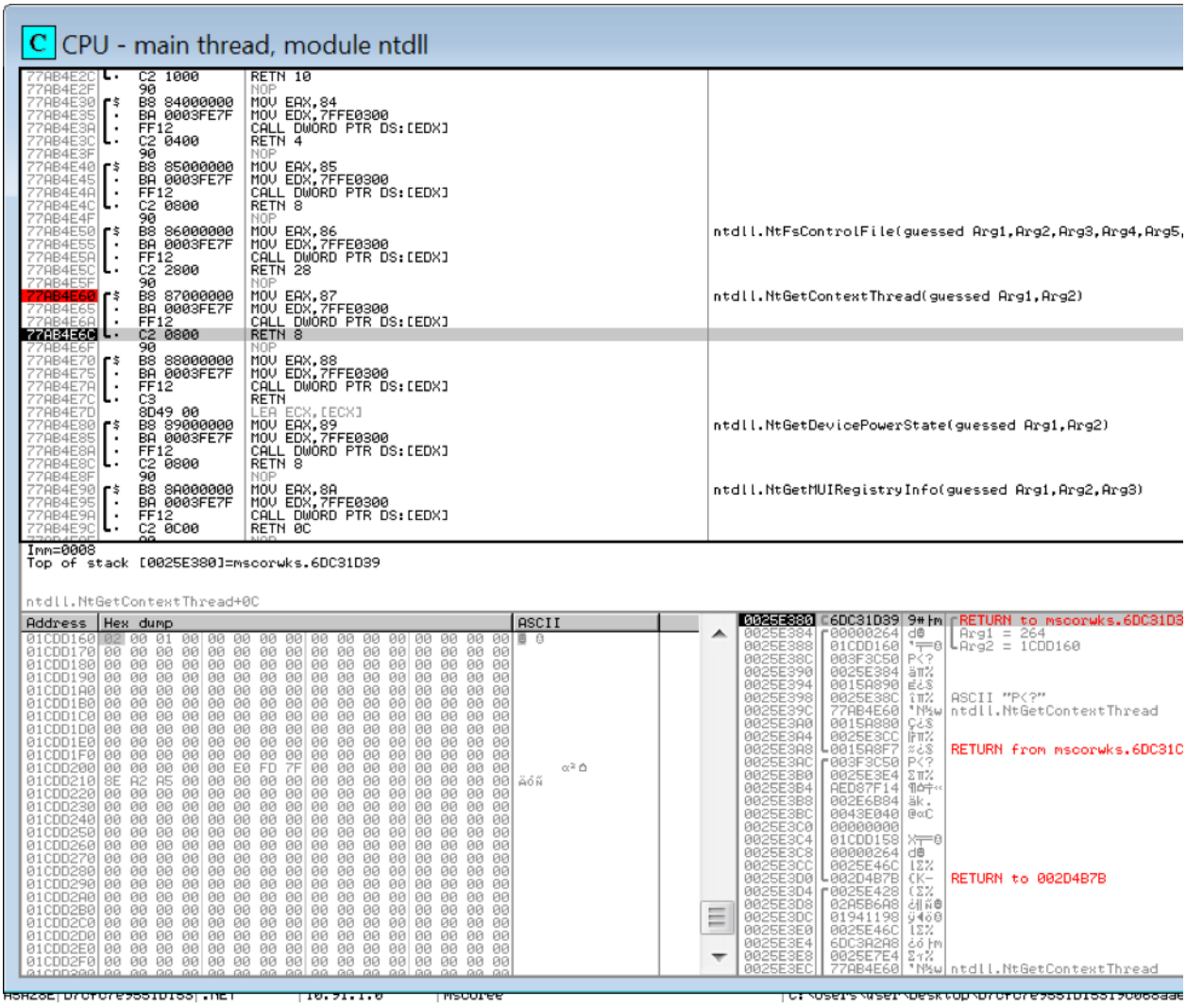
Injecting the Payload

The NanoCore RAT payload is never written to disk to avoid detection. Instead, it is injected into a new process. The injection routine can be summarized by these Win32 API and system calls:

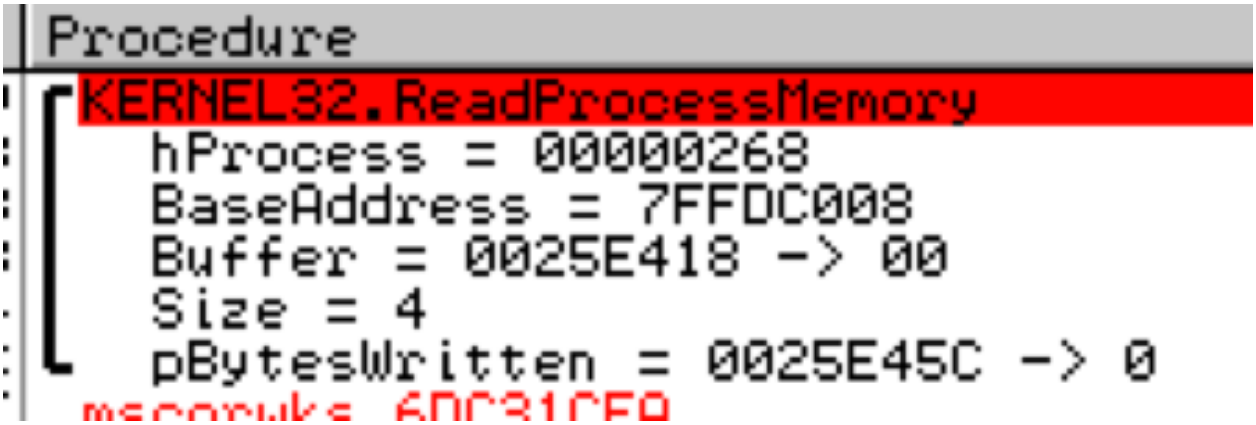
- **CreateProcessW(CREATE_SUSPEND): create the child process in suspend mode.**
- **NtGetContextThread(): Used to find the PEB and to update the EIP register.**
- **ReadProcessMemory(): Reads the PEB.ImageBaseAddress field.**
- **NtUnmapViewOfSection(): This runs only when there is an image already mapped to 0x400000.**
- **VirtualAllocEx(): Used to allocate the pages for injection.**
- **NtWriteVirtualMemory():**
- **0x00400000: MZ/PE Header**
- **0x00402000: .text**
- **0x00436000: .rsrc**
- **0x0043a000: .reloc**
- **PEB.ImageBaseAddress: Updates the base address to 0x400000.**
- **NtSetContextThread(): Updates the EIP register in the thread context.**
- **NtAlertResumeThread(): Causes the child process to leave suspend mode and become runnable.The process begins in suspend mode:**

```
Procedure
KERNEL32.CreateProcessW
  ApplicationName = "C:\Users\user\Desktop\b7cfc7e9551b15319c068aae966f8a9ff563b522ed9b1b42d19c122778e018c8.exe"
  CommandLine = ""C:\Users\user\Desktop\b7cfc7e9551b15319c068aae966f8a9ff563b522ed9b1b42d19c122778e018c8.exe""
  pProcessSecurity = NULL
  pThreadSecurity = NULL
  InheritHandles = FALSE
  CreationFlags = CREATE_SUSPENDED
  pEnvironment = NULL
  CurrentDirectory = NULL
  pStartupInfo = 0043E058 -> STARTUPINFOW {Size=68., Reserved1=NULL, Desktop=NULL, Title=NULL, X=0, Y=0, Width=0,
  pProcessInformation = 0025E428 -> PROCESS_INFORMATION {hProcess=NULL, hThread=NULL, ProcessID=0 (0.), ThreadID=
msconrwks.60C31CEA
```

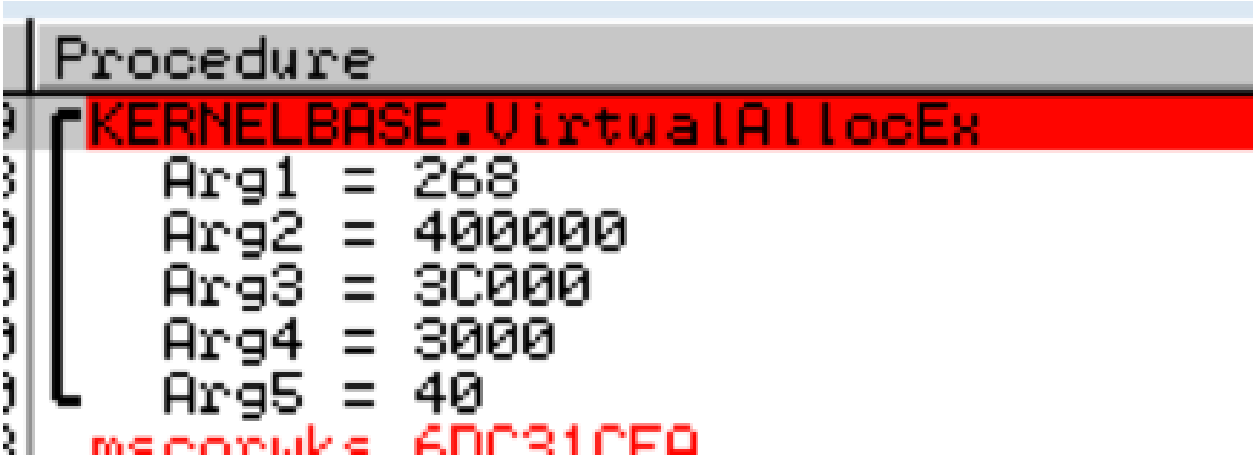
Next, the thread context is read from the child process:



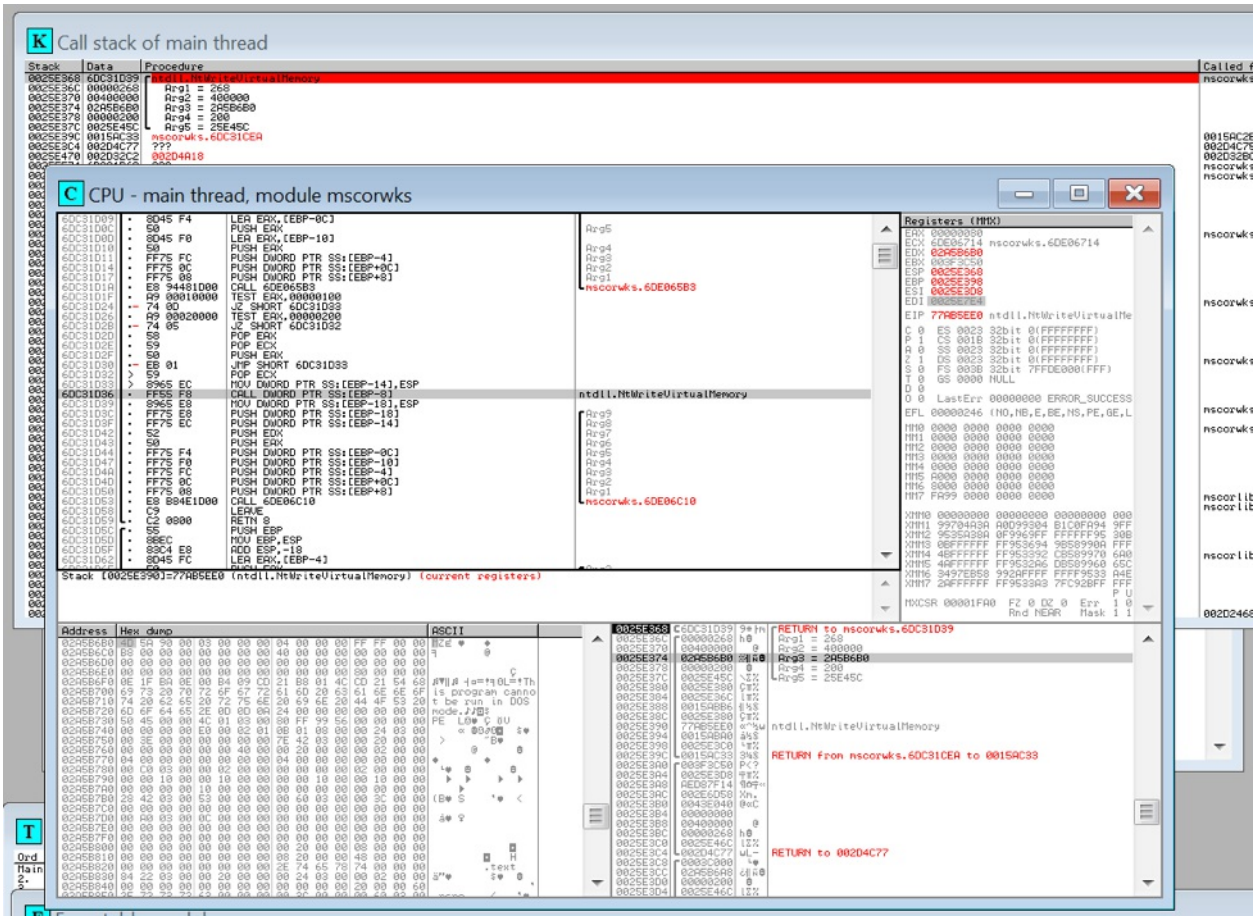
From the thread context, the address of the PEB is now known and is can be read:



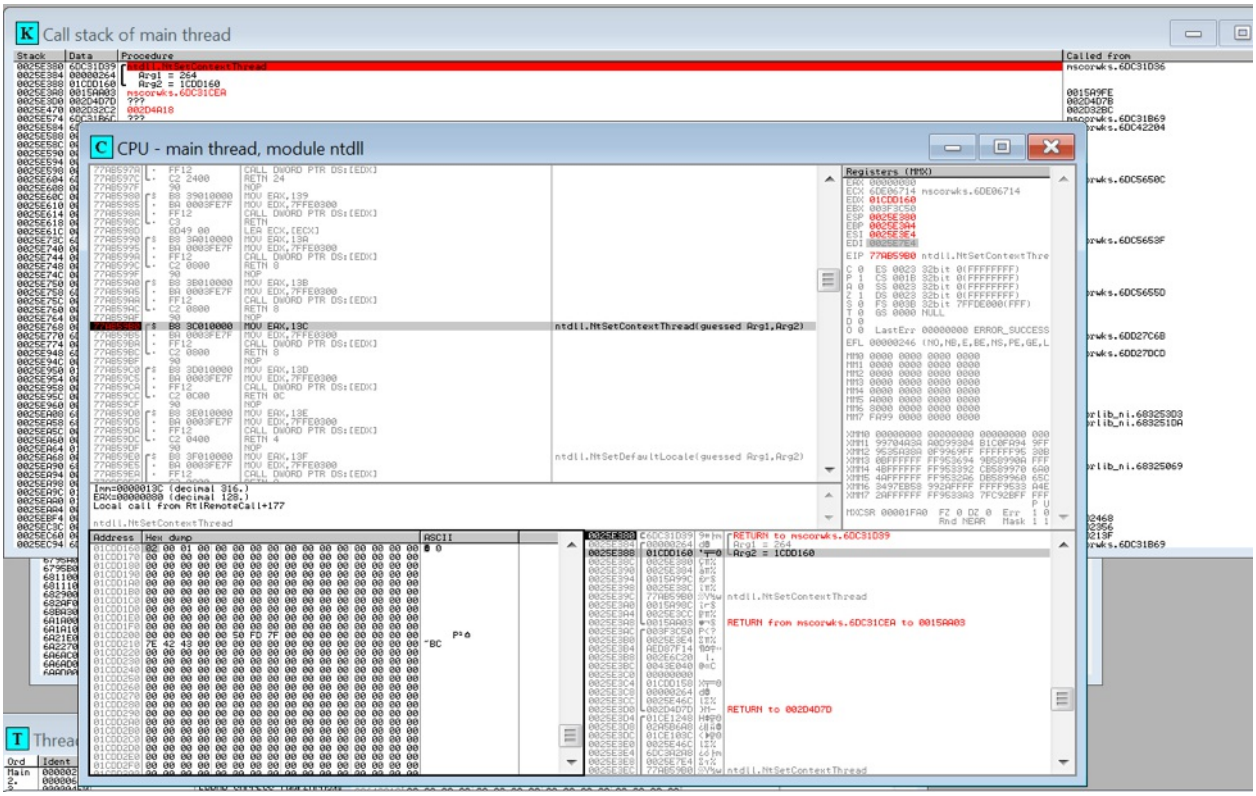
The address range for the injected image is now allocated:



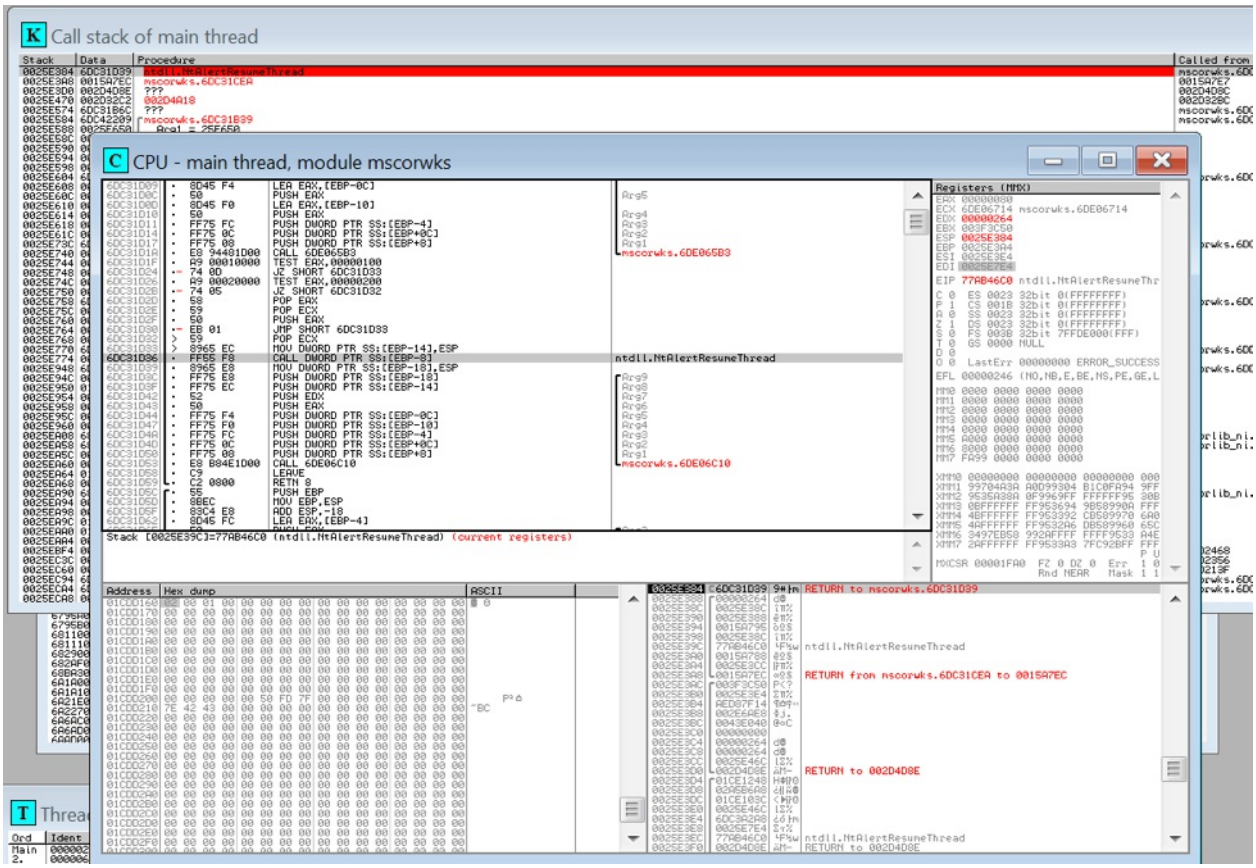
And now a series of NtWriteVirtualMemory() to inject the RAT image and update PEB.ImageBaseAddress.



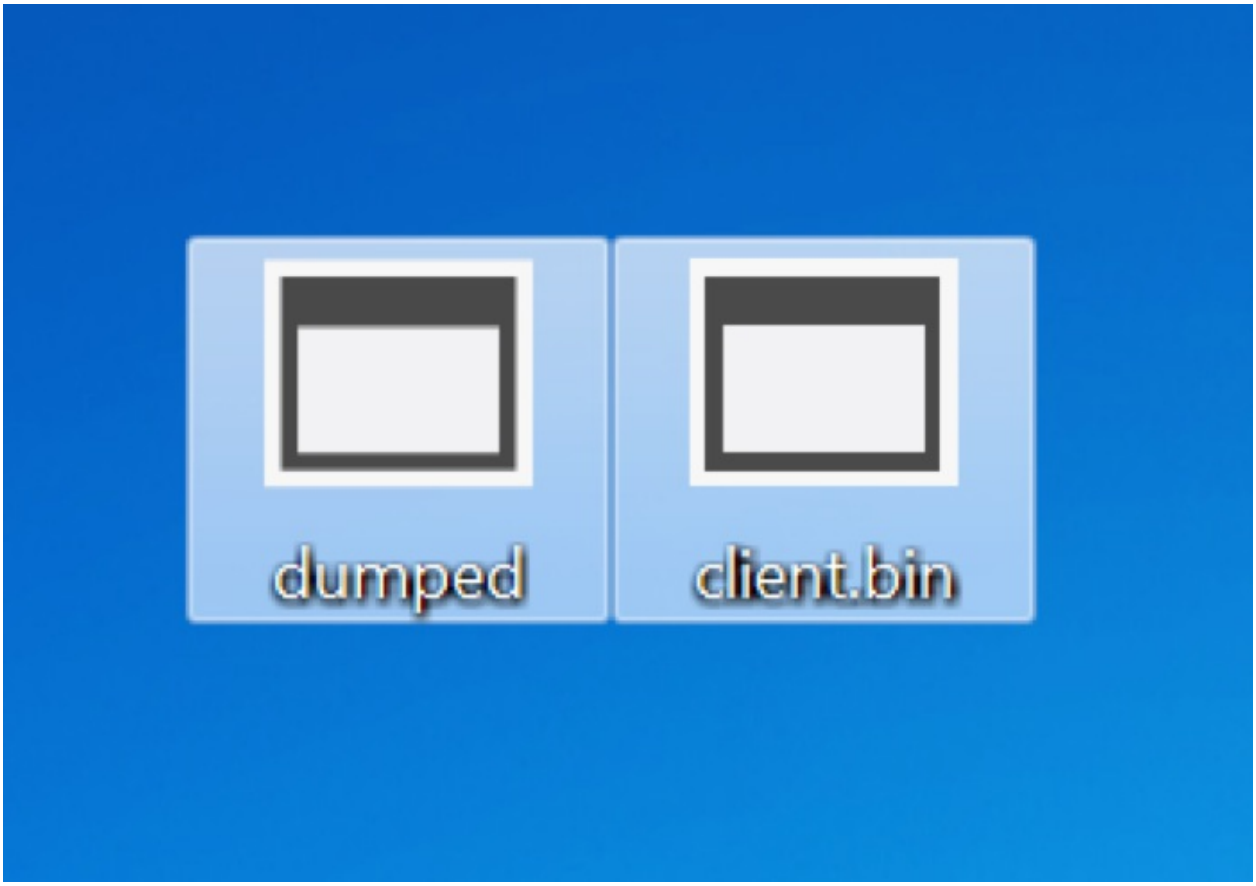
NtSetContextThread is invoked to update the EIP register's value:



Finally, execution is started with NtAlertResumeThread:



By dumping the process to disk, we can see that the injected process is just the NanoCore client.



For more information on how SentinelOne protects against attacks such as these, visit our resources page at sentinelone.com/resources