Teaching an old RAT new tricks

By Joseph Landry

Published: 2016-04-21 · Archived: 2026-04-05 22:54:06 UTC

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%MicrosoftBlend14.0FeedCachenvSCPAPISrv.exe and extracts a second binary named PerfWatson.exe

It then executes both binaries.

For persistence, a registry key is created at HKEY_CURRENT_USERSoftwareMicrosoftWindows NTCurrentVersionWindowsLoad 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.

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.

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.

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.

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.

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%\MicrosoftBlend14.0FeedCache.

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.

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

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:

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.

Worried about DDoS attacks? Check out our thorough guide about the attack vectors of this malicious virus and how to protect your data from the likes of BlackEnergy 3.
