Bird's nest

virusbulletin.com/virusbulletin/2014/08/bird-s-nest

2014-07-18

Raul Alvarez

Fortinet, Canada **Editor:** Martijn Grooten

Abstract

Usually, prepending viruses are relatively easy to clean and remove – just cut off the prepended virus and, in theory, the host file should be restored. However, in the case of prepending file infector Neshta, simply cutting the virus off will not do the job. Raul Alvarez takes a close look at Neshta, and at why it can't be removed as simply as other prepending infectors.

One of the ways in which file infectors are categorized is by how they attach themselves to the host file. The categories are: cavity, appending and prepending. A cavity virus infects a file by attaching itself to the available spaces in the host file, while an appending virus attaches its code at the end of the file, and of course, a prepending virus can be seen at the beginning of the victim file.

Usually, prepending viruses are easier to clean and remove from the infected file than the other types – just cut off the prepended virus and, in theory, the host file should be restored.

However, in the case of prepending file infector Neshta, simply cutting the virus off will not do the job. In this article, we will take a closer look at Neshta, and at why it can’t be removed as simply as other prepending infectors. Neshta is not new, yet we are still finding samples of it in the wild.

Simple decryption

Before we go into the details of how Neshta prepends its code to a host file, let’s look into some of its initialization routines.

Neshta decrypts some of its encrypted data using a simple decryption algorithm. The algorithm has a loop that decrypts each byte using the following routine:

- A variable is multiplied by 0x8088405 using signed integer multiplication. (The initial value of the variable is the counter, which is the number of bytes to decrypt.)

- The product of the multiplication, plus 1, is saved as the next value of the variable for the next itera-tion.
- The same product is multiplied by 0xFF to produce the decrypted byte, which is in the DL register.
- The byte in the DL register is copied into the AL register, then it is XORed to the encrypted byte to get the equivalent decrypted version (see Figure 1).

The decryption algorithm is used several times to obtain the following relevant strings: '3582-490', 'exe' and '*.*'.

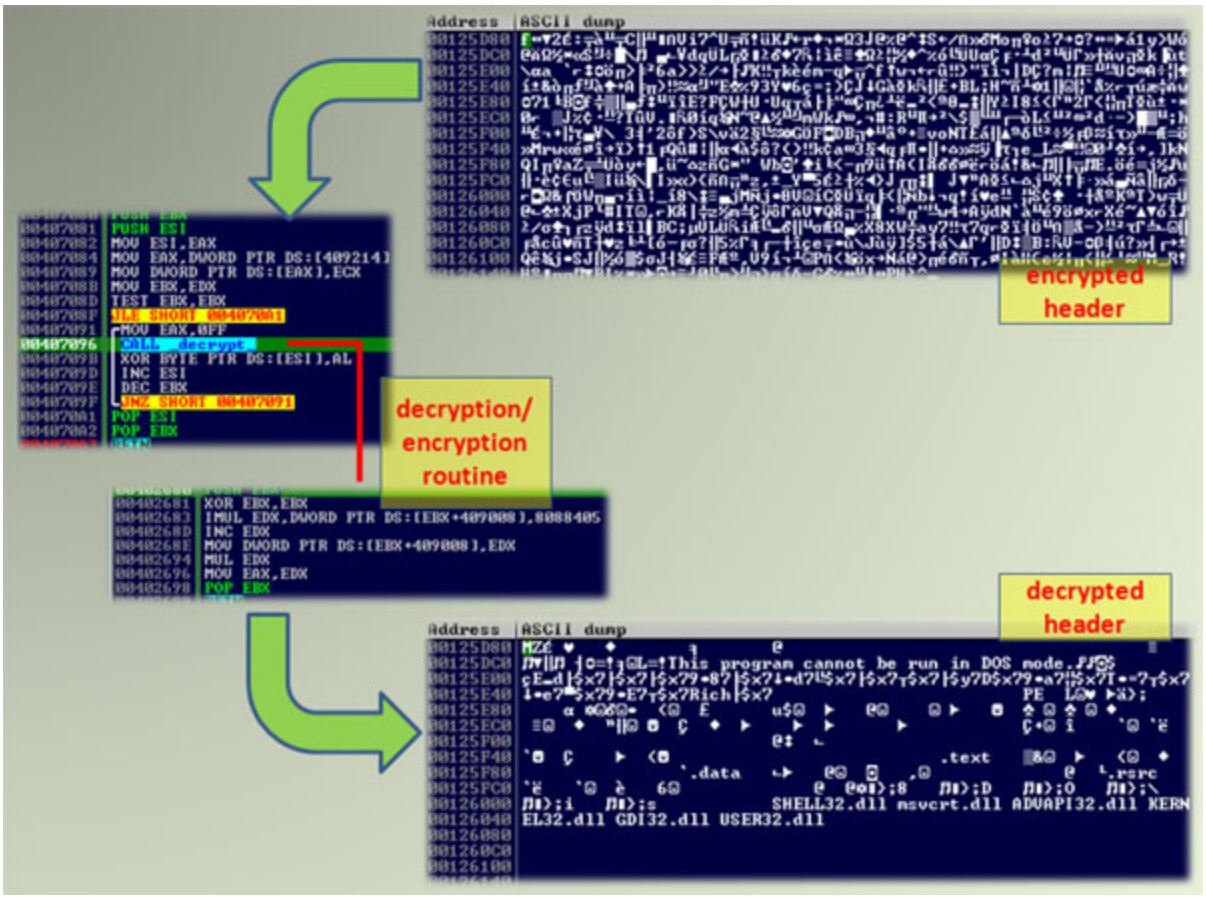

Figure 1. Decryption.

(Click here to view a larger version of Figure 1.)

Preparing the %temp% folder

The malware opens a copy of itself using a combination of the GetModuleFileNameA and CreateFileA APIs. Note that the file consists of the virus component and the host file. Then, Neshta extracts 41,472 (0xA200) bytes of the virus component by reading it into memory using the ReadFile API.

This is followed by getting the size of the infected file using the FindFirstFileA API. This is an unusual method, since the malware could just use a simple call to the GetFileSize API. Although the method works, the resulting WIN32_FIND_DATA structure contains the size of the file as well as other information.

Next, Neshta gets the %temp% folder name using a call to the GetTempPathA API. This is followed by concatenating one of the decrypted strings, '3582-490', to the temporary folder, producing the pathname '%temp%\3582-490'. The malware checks if the pathname exists by calling the GetFileAttributesA API. If it doesn't, it creates the pathname using the CreateDirectoryA API.

This is followed by concatenating another decrypted string, '*.*', producing '%temp%\3582-490*.*'. Using a combination of the FindFirstFileA and FindNextFileA APIs, the malware attempts to list all the files found in the '%temp%\3582-490\' folder, and tries to delete them one by one, using the DeleteFileA API.

Restoring the host file

After deleting all possible files that can be found in the '%temp%\3582-490\' folder, the malware extracts the filename of the current module from the result of the call to the GetModuleFileNameA API. The extracted filename, such as 'calc.exe', is concatenated to the '%temp%\3582-490\' folder, producing '%temp%\3582-490\calc.exe'. (Assuming that the current module is the infected version of the file, calc.exe.)

This is followed by creating '%temp%\3582-490\calc.exe' using the CreateFileA API with GENERIC_WRITE access in preparation for restoring the clean version of the host file.

Focusing back on the infected module, Neshta gets the file size of the infected calc.exe using the GetFileSize API, and subtracts 41,472 (0xA200) bytes from it. The difference is used to set the file pointer to the last 41,472 bytes of the infected file. This is followed by reading 41,472 (0xA200) bytes of data from the file into the stack memory.

Using the same decryption algorithm as discussed earlier, the malware decrypts the first 1,000 (0x3E8) bytes of the recently read data. Instead of using the counter, the malware uses the value 0x5A4D77D7 as a seed (see Figure 1).

It is worth mentioning that the first 1,000 (0x3E8) bytes are the only encrypted data in the host file. These bytes comprise the MZ/PE header of the original host file.

After the decryption, Neshta writes the 41,472 (0xA200) bytes to the '%temp%\3582-490\calc.exe' file, including the decrypted header.

To read the second block, the malware sets the file pointer 41,472 bytes from the beginning of the infected file using the SetFilePointer API (basically skipping over the virus component). Then it calculates the size of the remaining content of the host file (calc.exe), and reads it

into an allocated section of virtual memory using the ReadFile API. Finally, it writes the second block to the ‘%temp%\3582-490\calc.exe’ file. Then the handles for both files – infected and restored – are closed using the CloseHandle API.

The restored ‘%temp%\3582-490\calc.exe’ file is an exact copy of the original clean file (calc.exe). Note that calc.exe is just one example of a file infected by Neshta. If another infected file, such as notepad.exe, is executed, it will also be restored in the ‘%temp%\3582-490\’ folder for this variant of the malware.

Neshta executes the host file from the ‘%temp%\3582-490\’ folder to avoid raising suspicion that the file is infected, using a call to the ShellExecuteA API.

Hosting svchost

After restoring and executing the host file, Neshta gets the ‘%windows%’ folder name by calling the GetWindowsDirectoryA API. Then, using the same decryption algorithm, the malware generates the string ‘svchost.com’ and concatenates it to the *Windows* folder name, producing ‘%windows%\svchost.com’. Then, it checks whether ‘%windows%\svchost.com’ exists using the GetFileAttributesA API. If it does, the malware deletes the file using the DeleteFileA API.

After making sure that the ‘%windows%\svchost.com’ file does not exist, the malware creates it using a call to the CreateFileA API with GENERIC_WRITE access.

The malware reads the virus component of the infected file into memory, as discussed earlier. These bytes are written to the ‘%windows%\svchost.com’ file using the WriteFile API. Then, to finalize the routine, the malware closes the handle of the file using the CloseHandle API.

The file ‘%windows%\svchost.com’ is now an exact copy of the virus.

Modifying the shell key

After dropping the virus component as svchost.com, the malware generates the string ‘exefile\shell\open\command’ using the same decryption algorithm. Then it opens the registry key ‘HKEY_CLASSES_ROOT\exefile\shell\open\command’ using the RegOpenKeyExA API. Afterwards, another string, ‘"%1" %*’, is generated using the same decryption algorithm.

This is followed by setting the registry key ‘HKEY_CLASSES_ROOT\exefile\shell\open\command’ with ‘C:\WINDOWS\svchost.com "%1" %*’ as the data value using the RegSetValueExA API.

In a normal system installation, if the key ‘HKEY_CLASSES_ROOT\exefile\shell\open\command’ is missing or corrupted, any attempt to execute an application will not work, and an error message will be displayed, as shown in

Figure 2.

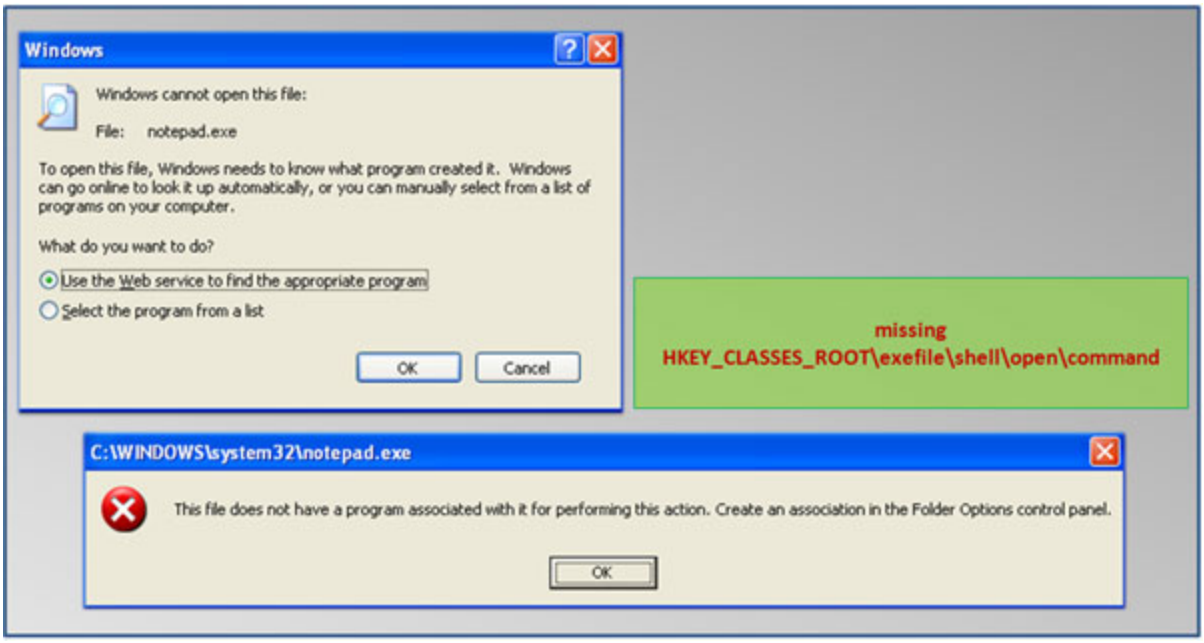

Figure 2. In a normal system installation, an error message will be displayed when attempting to execute an application if the 'HKEY_CLASSES_ROOT\exefile\shell\open\command' key is corrupted or missing.

However, a system infected with Neshta will have the modified registry key 'HKEY_CLASSES_ROOT\exefile\shell\open\command' with 'C:\WINDOWS\svchost.com "%1" %*'. Since C:\WINDOWS\svchost.com is the virus itself, an application will run, with the virus becoming the parent process and the .exe file the child process. This is a unique method to make sure that the malware will run even after restarting the system.

Creating a mutex

Another pass to the decryption algorithm reveals the string 'MutexPolesskayaGlush*.*', which is used as the name of a mutex created using a call to the CreateMutexA API. This is used to avoid running multiple instances of the malware.

Infecting removable drives

After creating the mutex, Neshta searches for available drives for infection. It lists the available logical drives in the system using the GetLogicalDriveStringsA API. Using the GetDriveTypeA API, it skips the infection routine for the CD-ROM drive, drive 'A' and drive 'B'.

With the exception of the list of drives to avoid, the malware tries to infect the executable files found on any attached removable drives (such as USB flash drives), all connected hard disks, and mapped network shared folders.

The malware traverses each folder in each drive searching for executable files for possible infection.

Skipping unwanted folders

If an executable file with the '.exe' extension name is found, Neshta gets its equivalent short path name using the GetShortPathNameA API. The short path name is the MS DOS-style naming convention. It has 8:3 form, where eight is the number of characters in the filename and three is the number of characters in the extension name.

This is followed by getting the '%windows%' folder name using the GetWindowsDirectoryA API. The malware skips the infection routine if the current short path name of the victim file contains the '%windows%' folder name. It also skips the infection routine if the victim file is inside the '%temp%' folder and if the path name of the victim file contains 'PROGRA~1' (the short name for 'Program Files').

When the path name has passed the filtering, it gets the size of the victim file using the FindFirstFileA API. The file size is taken from the resulting WIN32_FIND_DATA structure. Considering the file size, Neshta also skips the infection routine if the size is less than or equal to 41,472 (0xA200) bytes, or greater than 10,000,000 (0x989680) bytes.

The following section discusses how the malware determines whether the victim is already infected or not.

Avoiding reinfection

Initially, Neshta opens the victim file using the CreateFileA API. Then it sets the file pointer 1,000 (0x3E8) bytes from the beginning of the file. Afterwards, it reads 256 (0x100) bytes into memory using the ReadFile API, and closes the file using the CloseHandle API.

To avoid reinfection, the malware compares the 256 (0x100) bytes of the data read from the victim file against the virus component's data from the memory. If these bytes match, the malware will skip the infection routine.

Infection routine

Once the victim file has passed through all the necessary filtering, it is ready for infection. The first thing the malware does is to get the attributes of the victim file using the GetFileAttributesA API. If the file has an attribute of FILE_ATTRIBUTE_READONLY, the malware sets it back to 0, using the SetFileAttributesA API.

Using a series of calls to the ExtractIconA, GetIconInfo, GetObjectA, and DeleteObject APIs, the malware copies the icon used by the victim file into memory as part of the virus component.

Since Neshta is a prepending file infector, the first 41,472 (0xA200) bytes of each infected file belongs to the virus. A different block of data near the end of the virus component is a copy of the icon of the infected host file.

This is followed by opening the victim file with GENERIC_READ | GENERIC_WRITE access using the CreateFileA API. The malware reads the first two bytes and checks whether the file is a valid executable file by checking for the string 'MZ'. Then, it sets the file pointer to the beginning of the file using the SetFilePointer API.

Then, Neshta reads the first 41,472 (0xA200) bytes of the victim file using the ReadFile API. Using the encryption/decryption algorithm discussed earlier, the malware encrypts the first 1,000 (0x3E8) bytes of the recently read data, with the constant value 0x5A4D77D7 as a seed. Note that this is the same seed as was used to restore the original host file from the previous infection.

A regular prepending virus pushes the content of the victim file down to the end of the file, and places the virus component at the very beginning. However, instead of pushing the original bytes down, Neshta overwrites the first 41,472 (0xA200) bytes of the victim file with the virus component, using a combination of the SetFilePointer (to move the pointer to the beginning of the victim file) and WriteFile APIs.

Afterwards, the malware sets the file pointer to the end of the victim file and writes another 41,472 (0xA200) bytes of data, once again using a combination of the SetFilePointer (to move the pointer to the end of the victim file) and WriteFile APIs.

The data written at the end of the host file includes the newly encrypted 1,000 header bytes and the rest of the first 41,472 (0xA200) bytes of the victim file. These bytes are the original content taken from the beginning of the file which were overwritten with the virus component.

To finalize the infection routine, the malware closes the handle of the now infected file.

Finally, the malware traverses each folder in every selected drive to look for executable files to infect. After infecting all possible files, the malware terminates the current process. Note that the restored host executable file is still running, thus the malware only terminates the virus component's execution.

Wrap up

At first glance, Neshta looks like a simple prepending file infector. Detection of this malware is fairly easy since it is not polymorphic or metamorphic. In terms of cleaning, the encrypted header should be restored first and copied to the beginning of the infected file.

However, closer inspection of its malicious code reveals that Neshta is tricky in nature. It has embedded garbage code that works as a normal collection of instructions that are used throughout the virus body. It also uses several linked function calls that only perform simple

tasks.

As simple as it looks, one of the goals of the malware is to deter researchers from performing code analysis. In this regard, even a piece of malware with a strong encryption algorithm may seem easier to analyse than Neshta, as once the algorithm is broken down, analysis is straightforward.

In the end, as the old adage says, ‘Patience is a virtue’.

Latest articles:

Cryptojacking on the fly: TeamTNT using NVIDIA drivers to mine cryptocurrency

TeamTNT is known for attacking insecure and vulnerable Kubernetes deployments in order to infiltrate organizations’ dedicated environments and transform them into attack launchpads. In this article Aditya Sood presents a new module introduced by…

Collector-stealer: a Russian origin credential and information extractor

Collector-stealer, a piece of malware of Russian origin, is heavily used on the Internet to exfiltrate sensitive data from end-user systems and store it in its C&C panels. In this article, researchers Aditya K Sood and Rohit Chaturvedi present a 360…

Fighting Fire with Fire

In 1989, Joe Wells encountered his first virus: Jerusalem. He disassembled the virus, and from that moment onward, was intrigued by the properties of these small pieces of self-replicating code. Joe Wells was an expert on computer viruses, was partly…

Run your malicious VBA macros anywhere!

Kurt Natvig wanted to understand whether it’s possible to recompile VBA macros to another language, which could then easily be ‘run’ on any gateway, thus revealing a sample’s true nature in a safe manner. In this article he explains how he recompiled…

Dissecting the design and vulnerabilities in AZORult C&C panels

Aditya K Sood looks at the command-and-control (C&C) design of the AZORult malware, discussing his team's findings related to the C&C design and some security issues they identified during the research.

Bulletin Archive