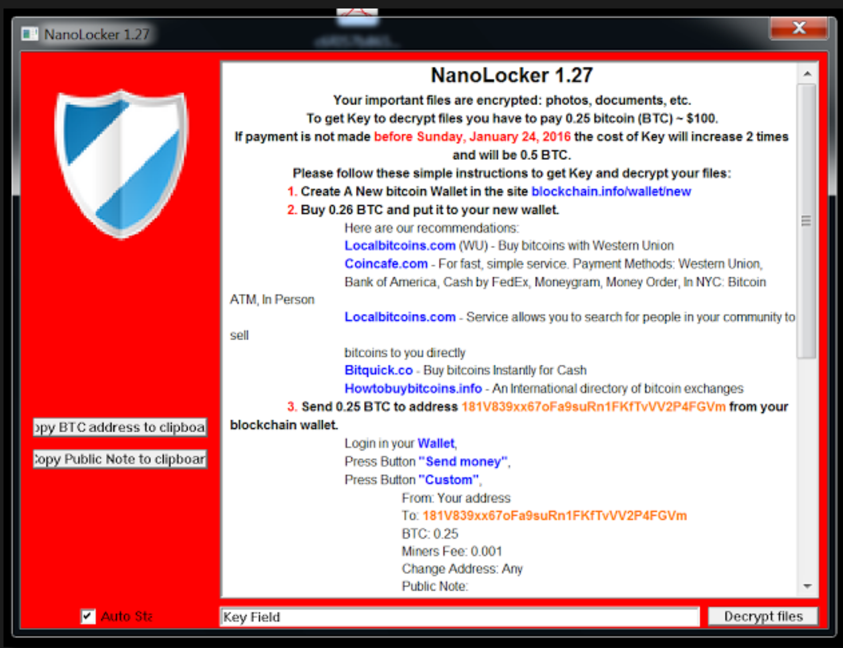
malware clipboard

Sunday, 24 January 2016

NanoLocker - Ransomware analysis



I had occasion recently to come across a previously unseen (by me) ransomware variant which goes by the name 'NanoLocker'. Other than some very general posts such as this one by Symantec, there doesn't seem to be much published analysis of this ransomware. In light of this fact I decided to reverse-engineer a few samples in order to understand how it worked.

The detailed analysis is presented below, but I will mention upfront that I was able to build a decryption tool for files encrypted with this ransomware which will work ***in certain specific circumstances.*** The details of these required preconditions are outlined below.

Note on Samples

The samples I examined for this post were of two different (yet functionally similar) versions of NanoLocker:

- Version 1.27 - MD5 : c1cf7ce9cfa337b22ccc4061383a70f6
- Version 1.29 - MD5 : fce023be1fb28b656e419c5c817deb73

Overview

NanoLocker takes a similar functional approach to that of other common ransomware variants. This malware was written in C++ and leverages the Windows CryptAPI for its crypto routines. The cryptographic design is mostly solid - the files are encrypted using a run-time generated AES-256 key, however the initialization of this AES key is done with a default (null) IV. After use, this AES key is obfuscated slightly before being encrypted using an RSA public key which is imported from a base64 string hardcoded into the binary. This newly encrypted key is then finally base64 encoded and displayed to the user in order for it to be transmitted to the malware author via public note during the Bitcoin payment. We will refer to this final base64 encoded key material as `MK`.

The main design flaw in this ransomware is similar to that of early TeslaCrypt/AlphaCrypt in that the AES key is written to a file on disk, and this key is left untouched until the encryption process is complete. The implications of this are that any interruption of this encryption process (eg. by entering hibernation, power-down, etc) can leave the symmetric encryption key available on disk.

The decryption tool I have written capitalizes on this design choice and thus offers a method of decryption for victims who have either captured a copy of the key file while NanoLocker was in the midst of encrypting their data, or if they were able to interrupt the process via shutdown/hibernate and have managed to acquire this key file from disk.

About Me

Adam (@cyberclues)
View my complete profile

Blog Archive

Execution Details of Interest

State Tracking

NanoLocker creates a file on disk which is used as a tracking mechanism for its state, key information, and file target list. In the analyzed samples the filename used for this is `%LOCALAPPDATA%\lansrv.ini` - and this file is created with the hidden attribute set.

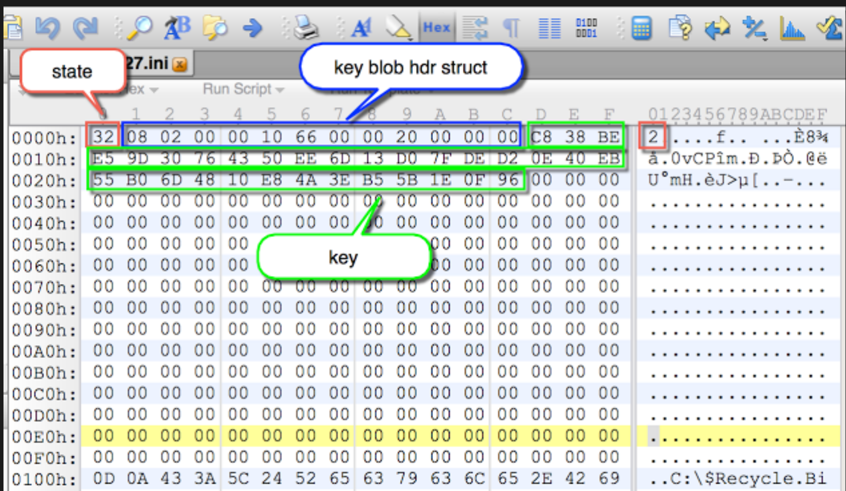
The first byte of this file is used to hold an integer which represents the current execution state. This state tracking value is increased as the malware progresses with its various functions. There are three possible states for the tracking byte:

1. *Initialization & Enumeration* - symmetric key has been created and is stored in plaintext format [44 bytes] in the tracking file starting at byte offset 0x02. All locally mapped drives are enumerated for filenames matching the targeted extensions, and these filenames are written to the tracking file.
2. *Encryption* - Encryption of the files listed in the tracking file begins once this state is reached.
3. *Encryption Complete* - all targeted files have been encrypted. Once in state 3, the raw symmetric key is replaced with the bitcoin payment address followed by **MK**. At this point the NanoLocker splash screen is displayed to the user.

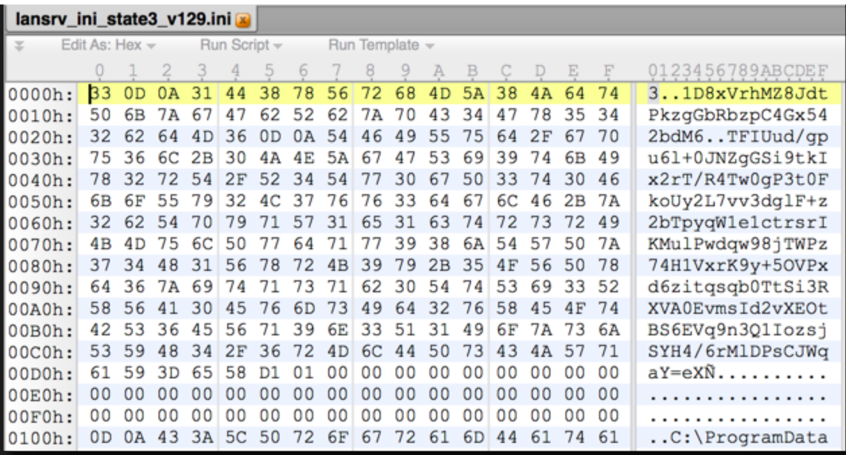
During initialization the AES key is generated and written to the file along with the state 1 marker as shown in the following pseudocode from the unpacked binary:

```
UpdateWindow(hWnd);
if ( !CurrentState )
{
  CryptAcquireContextA(&hProv, 0, 0, 0x18u, 0xF0000000);
  CryptGenKey(hProv, 0x6610u, 1u, (HCRYPTKEY *)&hKey);
  CryptExportKey((HCRYPTKEY)hKey, 0, 8u, 0, 0, &nNumberOfBytesToRead);
  CryptExportKey((HCRYPTKEY)hKey, 0, 8u, 0, &Buffer, &nNumberOfBytesToRead);
  CryptDestroyKey((HCRYPTKEY)hKey);
  CryptReleaseContext(hProv, 0);
  hFile = CreateFileA(Lansrv_ini, 0xC0000000, 3u, 0, 4u, 2u, 0);
  SetFilePointer(hFile, 1, 0, 0);
  v9 = WriteFile(hFile, &Buffer, nNumberOfBytesToRead, &NumberOfBytesWritten, 0);
  if ( NumberOfBytesWritten != 44 )
    ExitProcess(v9);
  SetFilePointer(hFile, 0, 0, 0);
  CurrentState = 0x31;
  WriteFile(hFile, &CurrentState, 1u, &NumberOfBytesWritten, 0);
  SetFilePointer(hFile, 256, 0, 0);
  WriteFile(hFile, asc_4068F3, 2u, &NumberOfBytesWritten, 0);
  CloseHandle(hFile);
}
```

Once in state 2, and until all discovered files are encrypted, the tracking file (`lansrv.ini`) holds the exported key data which was written in the above call to WriteFile. Capturing the tracking file at this point will reveal the state flag, key data, and a list of targeted files:



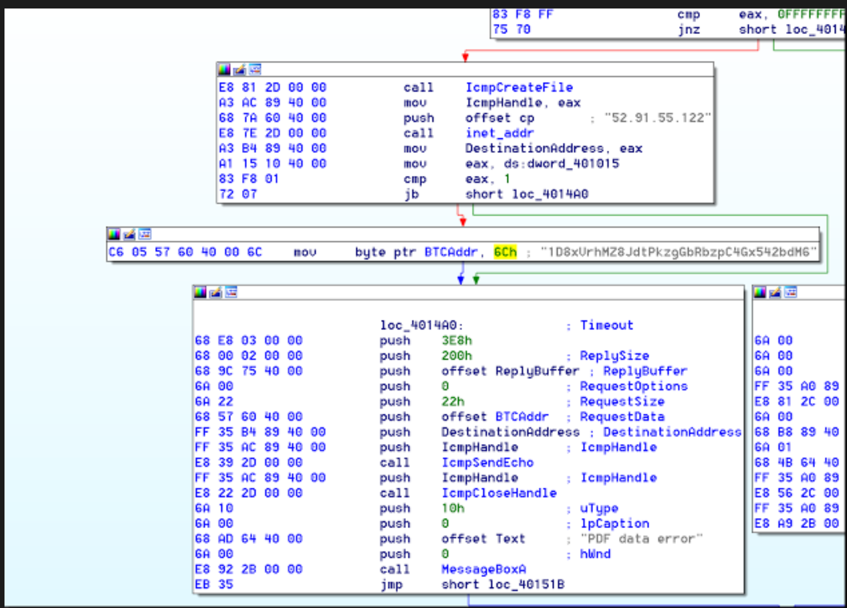
Finally, once the targeted files have been encrypted the state 3 flag is written to the tracking file. At this point the key data is replaced with the bitcoin address followed by the base64 encoded, public key encrypted symmetric key (**MK**):



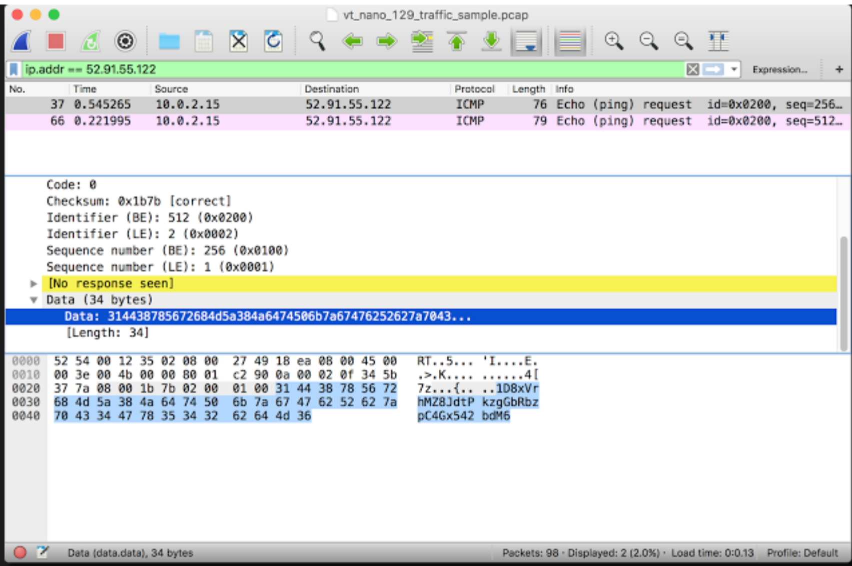
Communication

A common action for ransomware threats is to transmit the key material over the network to an attacker controlled command server. NanoLocker on the other hand takes a minimalistic approach to network communication - it transmits only two ICMP packets out to the C2 server.

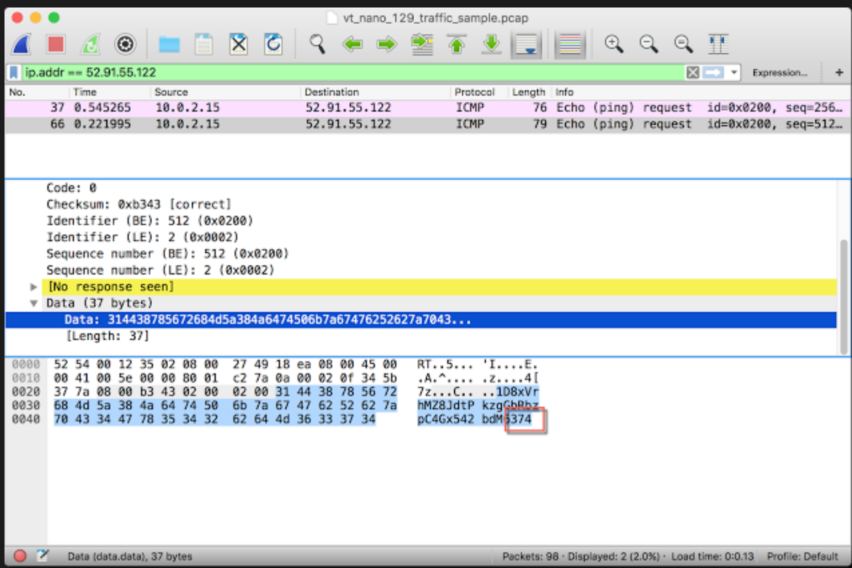
Once the malware unpacks itself in memory it carries out some initialization steps (dropping a copy of itself to disk, setting a persistence mechanism, etc). The next step is to submit a ping to the command and control server. These ping packets might appear at first glance to be typical echo request packets, but as the code below reveals, there is something else being sent:



As we can see above, the call to `IcmpSendEcho` uses the ransomware bitcoin address as the data to submit with the echo request. We can see this in action if we capture the packets going out from the infected system:



The second ICMP packet sent by NanoLocker occurs once the encryption process has completed, immediately after state 3 is reached. This packet, similar to the first, uses the data bytes of the ICMP message to send the bitcoin address, and it also appends to this the total number of files that were encrypted during state 2:



Decryption Tool

As described above, if the tracking file can be captured during either state 1 or 2 (through interruption of the encryption process or otherwise), the symmetric key can be extracted and used to decrypt any files that may have already been encrypted.

Admittedly, for most stand alone Windows systems, capturing this tracking file in either of its first two states may be operationally infeasible. This is simply a function of the relatively small number of files present and the speed of encryption on such a system. However in larger environments with huge distributed file systems such as those found in modern enterprise networks, it may be possible to detect NanoLocker-encrypted files prior to the completion of the encryption phase (stage 2). Such a scenario is more likely due to the increased time required to encrypt tens or hundreds of thousands of files. Encryption at these scales can take several hours or possibly even days for larger file systems.

The usage of the tool is as follows:

```
NanoLocker_Decryptor.exe <encrypted_file> <output_file> <tracking_file>
```

This decryption tool will examine the encrypted source file and the provided tracking file to validate two required conditions:

1. that the tracking file is in either state 1 or state 2, and
2. that the encrypted source file was encrypted using the key found in the provided tracking file

The second step is possible thanks to another design choice made by the creator of NanoLocker. In each encrypted file there is a header prefixed to the actual encrypted bytes of the original file. This header contains a checksum value which can be used to validate that the key we are trying to decrypt with was the same one used to encrypt the file initially.

The source code is available on github here. A precompiled version is available here. For the precompiled version, you will need the 32bit Visual C++ 2013 redistributable if you don't already have it installed. It is available from Microsoft here.

Final Comments

I avoided describing the details of unpacking and disassembling this ransomware. In a follow-up post I will document the steps required to carry out this unpack in a tutorial fashion. However it is worth pointing out that an unpacked copy of this particular malware can be easily dumped from memory using Volatility.

First determine the process id using pslist :

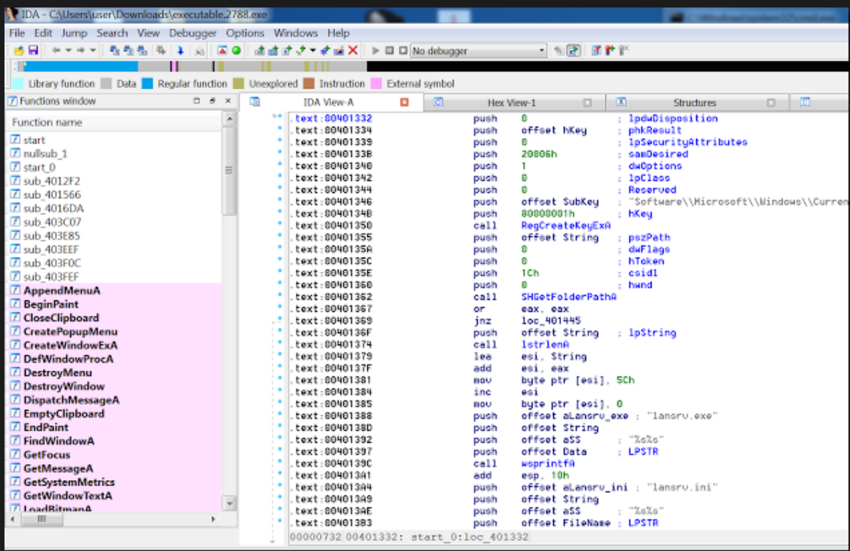
```
malwareclipboard:volatility cyberclues$ python vol.py -f /Volumes/mcb_tb/Virtual\ Machines/Linked\ Clone\ of
evm/Linked\ Clone\ of\ Windows\ 7-32bit-70a42bb5.vmem --profile=Win7SP1x86 pslist -n 'nano'
Volatility Foundation Volatility Framework 2.5
Offset(V)  Name                    PID   PPID   Thds     Hnds   Sess  Wow64 Start                          E

---------- -------------------- ------ ------ ------ -------- ------ ------ ------------------------------ -
-----
0x8515f030 nanov129.exe           2788    992      3      140      1      0 2016-01-24 05:47:44 UTC+0000
```

Then dump the process to disk, using the fix-up option (-x):

```
malwareclipboard:volatility cyberclues$ python vol.py -f /Volumes/mcb_tb/Virtual\ Machines/Linked\ Clone\ of\ Windows\ 7-32bit.vmwar
evm/Linked\ Clone\ of\ Windows\ 7-32bit-70a42bb5.vmem --profile=Win7SP1x86 procdump -p 2788 -x -D ~/Documents/Research/NanoLocker/vo
ldumps/
Volatility Foundation Volatility Framework 2.5
Process(V) ImageBase  Name                 Result
---------- ---------- -------------------- ------
0x8515f030 0x00400000 nanov129.exe         OK: executable.2788.exe
```

Then load into your favourite disassembler:



Posted by Adam (@cyberclues) at 17:12

Recommend this on Google

No comments:

Post a Comment

Enter your comment...

Comment as: ggyy (Google)

Sign out

Publish Preview Notify me

Home Older Post

Subscribe to: Post Comments (Atom)