

# Diving into the Sun — SunCrypt: A new neighbour in the ransomware mafia

---

 [medium.com/@sapphirex00/diving-into-the-sun-suncrypt-a-new-neighbour-in-the-ransomware-mafia-d89010c9df83](https://medium.com/@sapphirex00/diving-into-the-sun-suncrypt-a-new-neighbour-in-the-ransomware-mafia-d89010c9df83)

Sapphire

February 1, 2021



Sapphire

Nov 12, 2020

.


16 min read



The first time I heard about **SunCrypt** I was just enjoying my time off preparing some stuff to get dinner ready. My colleagues sent me over some weird samples that were flagged as **SunCrypt** that were also beaconing to **Maze** C2 infrastructure, however they weren't **Maze** but shared some similarities and that's basically why they did it as they know my interest in ransomware and my personal fascination with **Maze**.

As I was checking a couple of hashes, I saw very interesting things that made me go hands-on to investigate further and I ended up spending a lot of time not only analysing but also hunting and creating some detections for this malware.

Initial Open Source Intelligence shows that **SunCrypt** made its first appearance around October 2019 and it has been active since then. The **SunCrypt** team is active on underground forums where they look for affiliates for their program just like the rest of ransomware operating under the **RaaS** model.

 SunCrypt forum advertisement

In their advertisement in a popular forum, shared in Twitter by ShadowIntelligence, can be observed some of the features they offer for their product such as crypto independency from the Windows CryptoAPI, “bypass” of 70% of the AV engines, which is not a lot..but I guess at least they didn’t lie selling it as FUD as other threat actors, asynchronous search and encryption, speed etc.

Additionally to the lock service, the **SunCrypt** team provides **DDoS** if they consider it as part of the extortion to get the ransom paid.

 SunCrypt shaming website

SunCrypt shaming site where they expose victims, status and data dumps

For some journalists and websites related to **infosec**, they are part of what they call “**The Maze Cartel**” and they also drew some shy lines associating this team with **Maze**.

My **personal** opinion: I don't believe such thing as a Cartel exists but sounds fancy. This would imply a high degree of collaboration, potential relationships and involvement in the money-laundry circuit, extortion and affiliation methodology... this is very different from a good or

neutral relationship with potential collaborations to share resources and infrastructure. I wanted to see with my own eyes the degree of relationship and sophistication of this ransomware family.

That said, to start the analysis I grabbed two samples that are available not only in **VT** but also in **AnyRun** and from there I started the analysis to figure out how it works and hunting more samples for code comparison.

This was fundamental to me to construct a solid opinion and conclusions that I may share publicly with the community.

## **SunCrypt PowerShell loader analysis**

---

**SunCrypt** ransomware has been spotted in many cases using **PowerShell** loaders for delivery and deploy following the tendency marked by other groups offering ransomware service.

After the analysis of some of these loaders, I could observe that share some similarities or reminds to me the structure and functionality of other loaders such a **Netwalker PowerShell Loader** scripts. The **SunCrypt** loaders contains an embedded resource in plain text with two export functions that are called by the script after the compilation of the **C#** code, heavy obfuscation and a lot of junk code and useless data to harden the analysis and detection.

The obfuscation of the script includes **arithmetical operations**, **encoding** and **string manipulation** not only for anti-analysis but probably to avoid detection by segmenting the base64 strings.

SunCrypt ransomware powershell

PowerShell Loader script obfuscation

Jumping to the analysis, the sample md5 **c171bcd34151cbcd48edbce13796e0ed** is a **PowerShell** loader containing the heavy obfuscation mentioned above and contains an embedded small PE file md5 **479712042d7ad6600cbe2d1e5bc2fa88** coded in **C#** that is compiled in runtime and dropped to disk.

This DLL is used to assist with **Process Injection** of the payload by adding a class with obfuscated names for the exported functions to call the functions **VirtualAlloc** and **EnumDesktopW**. These are used to allocate memory and enumerate desktops associated with the process.

The **PowerShell** script spawns an additional process instance of **Powershell** which is directly invoked with the following arguments after the deobfuscation on the fly of the parameters:

```
C:\Windows\SysWOW64\WindowsPowerShell\v1.0\powershell.exe -ep bypass -file  
<filepath of the script>
```

The main structure of script contains two heavily obfuscated payloads base64 encoded in multiple strings that are subsequently deobfuscated by multiple operations and then decoded.

**PowerShell** uses the dropped **.NET DLL** to allocate memory space and copies the resultant buffers to the newly created **PowerShell** process. The injected bytes are consistent with **shellcode** and a **PE32 file** that are injected into PowerShell's memory address space.

To analyze the payload the buffer containing the deobfuscated payload was dumped to disk to carve the bytes.

A basic triage of the **shellcode** showed that it contains functions related to process injection such as **VirtualAlloc** and **VirtualProtect**, likely indicating that the **shellcode** is assisting the **PowerShell** script to leverage the injection of the payload.

## SunCrypt payload Analysis

---

Most analyzed samples share the same code structure and characteristics but one of them had a simpler structure that facilitated the analysis as the only major modification that most samples contain was the addition of more obfuscation layers.

In this case I will jump directly to the analysis of the code shared between multiple samples followed by a summarisation and conclusion.

From the PowerShell loader md5 **c171bcd34151cbcd48edbce13796e0ed**, the payload md5 **0a0882b8da225406cc838991b5f67d11** was dumped and the bytes carved for analysis. This PE file is consistent with an executable file likely to be coded in **C** and contains capabilities to **encrypt the filesystem, delete backups** and to **gather and exfiltrate user and host information**.

One of the first noticeable difference of the sample md5 **0a0882b8da225406cc838991b5f67d11** is that it does not contain commandline arguments unlike other analyzed samples like md5 **3d756f9715a65def4a302f5008b03809** (payload carved from the PowerShell loader with MD5 ) which contains multiple arguments .

Most samples contain these command line arguments just like the one I just mentioned above that are intended to modify the behaviour of the ransomware. One of the key characteristics of **SunCrypt** is that contain an embedded configuration that is likely to be inserted and then encoded by the builder.

These command line arguments seen for example in md5 **0a0882b8da225406cc838991b5f67d11** that are expected share some similarities with the ones seen in other malware like **Maze** ransomware.

**SunCrypt** commandline arguments are:

- 
- 
- 
- 
- 

The triaged samples that expect command line arguments contain obfuscated parameters and **SunCrypt** uses **FNV hashing** to obfuscate these parameters.



SunCrypt arguments flow

IDA flow with each hash value and parameter

**FNV ()** is a **non-cryptographic hash function** created by **Glenn Fowler, Landon Curt Noll, and Kiem-Phong Vo**. FNV functions contains two primary operations: **XOR** and **multiplication**. These can be spotted also by identifying the hash values for **FNV Prime** (0x01000193) and the **FNV offset basis** (0x811c9dc5).

In this case the identified version that **SunCrypt** is using is **FNV-1a hashing**.

## SunCrypt arguments flow

Above the FNV hashing can be observed to obfuscate the commandline arguments, however the value for “-noreport” is not hashed.

The next step the ransomware takes is to decode the configuration.

This configuration added and encoded by the builder contains the following parameters:


- 
- 
- 
-

The whole section containing the configuration is decoded in multiple steps to save some of its values for its use in different routines, however the whole configuration is decoded by multiple loops that decode the data with a single-byte xor key 0x11.

The image shows a placeholder for a document titled "SunCrypt ransom note decoded". The document content is not visible, only the title and a small icon of a document with a red 'x' in the top-left corner are present.

Decoded data containing the HTML code for the ransom note.

One of the most noticeable things on the ransom note is that is written in multiple languages including **Spanish**(Latin American Spanish), **German**, **French** and **English**. This note contains an hardcoded identifier in hex format that is intended to be used to start negotiating the payment of the ransom along with the **TOR Hidden Service** URL where **SunCrypt** hosts its webpage and Shaming list containing the victim names, proofs and data dumps.

 SunCrypt ransom note

SunCrypt ransom note

After the initial analysis of the configuration its data showed that both the offset of the ransom note containing the **individual key** in hex format and other values that are used later are **not generated on the fly** indicating that they are likely to be added by the builder during the construction process.

One of the decoded details is a string in hex format that is re-encoded at a later stage and sent within the **POST** data that is likely to be an implant identifier for the attackers.

The fact that all the data is precomputed or hardcoded is not always typical in ransomware and popped up some questions in my head about the crypto used that I had to figure out later.

Once the C2 are decoded the malware looks for multiple addresses in order to save them for its use later or just continues if only one is used.

The image below shows that after the configuration decoding and extraction, the next step taken by **SunCrypt** is a quick system check to delete the **Shadow Copy Volumes**. Microsoft's shadow copy technology allows Windows systems to create snapshots-backups of files and volumes. This step ensures no possible recovery.

The image area is mostly blank, with a small icon and text at the top left. The text reads "SunCrypt ransomware configuration".

 SunCrypt ransomware configuration

After decoding the embedded configuration, it saves the C2 addresses for its use  
In most samples all the data such a **function names, DLL names and other data** are dynamically loaded and obfuscated using **xor, sub** or **add** operations with random values likely to be inserted by the builder combined with the use of **StackStrings** for **anti-analysis** and

## **evasion.**

The function in charge of deleting the **Shadow Copy Volumes** works in the same way and all its steps are obfuscated. **SunCrypt** retrieves the system version to look for its architecture to handle this via **WMI query** executing the query “**select \* from Win32\_ShadowCopy**” to then delete all the available volumes.

A screenshot of a process window titled "SunCrypt shadowcopy deletion". The window is mostly empty, suggesting the process is either just starting or has completed its task. The title bar and a small icon are visible at the top left.

SunCrypt shadowcopy deletion

obfuscated flow to decode strings used to retrieve system architecture and to handle the task via WMI

One of the most interesting things I noticed is the fact that so far most samples create a mutex but not all of them. In this case, the below screenshot of the sample

**3d756f9715a65def4a302f5008b03809** creates it using this hex string that was mentioned

above and that is decoded during the configuration extraction.

A screenshot of a process window titled "SunCrypt pulls system version and create mutex". The window content is mostly blank, with some faint, illegible text visible in the background. The title bar and a small icon are visible at the top left of the window.

SunCrypt pulls system version and create mutex

After the config decoding, gets the System Version and creates the mutex

Once **SunCrypt** deletes the **Shadow Volumes**, it scans for available drives following the keyboard scheme from **Q:** to **M:**. In the case of the sample md5

**0a0882b8da225406cc838991b5f67d11** it directly contains the hardcoded volume letters instead of using other techniques to harden detection such a incrementing loops. However

most samples like md5 **3d756f9715a65def4a302f5008b03809** contains a different technique using **FNV-hashing** again to obfuscate this phase, indicating that the developer team modified this to make its detection more complicated.

A screenshot of a process window titled "SunCrypt drive scanning". The window is mostly empty, with the title bar and a small icon in the top-left corner. The rest of the window is blank white space.

In most samples, drive scanning phase is obfuscated using FNV hashing, funnily enough, contains the strings.

## File encryption and Crypto walkthrough

---

**SunCrypt** uses a particular way to encrypt the filesystem and this is somehow reflected in their advertisement where they specify that **SunCrypt uses a completely independent cryptography from the system API**. Well, this is true and stepping through the crypto made it complicated as there's only a few visible calls and it does completely rely on itself instead of the **Windows CryptoAPI** to generate the keys because the implementation is almost completely manual.

Another interesting feature observed during the crypto walkthrough is that skips files that are **empty** or less the **512 bytes size** to skip potential garbage files. The Crypto is basically identical in all the samples and it consists on a first loop to scan all the directories ensuring the drive **C:** is present and is encrypted. However **SunCrypt** scans all the available network drives such as **Windows SMB File Shares** in order to encrypted them if they are connected and available.



The malware in most cases contains **two lists**:

- of directories: contains a list of directories that skips from the encryption process to ensure its recovery. These directories are: “”, “ ”, “”, “”, “”.
- of file extensions: A huge list of file extensions are used to ensure only vital files are maintained intact.

SunCrypt file extension check via FNV

note again the use of FNV hashing and the FNV prime hash above. The number 13:19:00 corresponds to the end of the array containing the file extensions.

These two arrays of lists are used with the **FNV hashing** technique again to ensure obfuscation, however md5 **0a0882b8da225406cc838991b5f67d11** again saves the day containing all this information in plain text. The directories and file extensions don't change

between samples and versions.



Another file extension that is skipped is if the filename contains “**YOUR\_FILES\_ARE\_ENCRYPTED.HTML**” which is used for the ransom note to ensure its delivery to the victim, however there is a problem that is not resolved.

If the ransomware is executed twice, it would re-encrypt the filesystem as the **file extension of the encrypted files is generated on the fly**. This embarrassing scenario indicates that if a previous instance of **SunCrypt** encrypts the filesystem, another execution would re-encrypt all the encrypted files, making this very hard to solve.

Once the directory is picked for encryption **SunCrypt** decodes the DLL name “**advapi32.dll**” and a function to be loaded and used. The malware executes a **GetProcAddress** to use the function **SystemFunction036(RtlGenRandom)**, this will be our **RNG** to generate **32 random bytes without using CryptGenRandom** function call or an insecure generator.

The image shows a small icon of a document with a green checkmark, followed by the text "SunCrypt RNG". The rest of the image area is empty.

After decoding the DLL and the function that will be called, SunCrypt uses SystemFunction036 to create 32 byte keys.

This **32-byte buffer** is validated after it's generation to create a secure **private key** for the **Elliptic Curve** algorithm **Curve**. **Curve25519** is a **Diffie-Hellman** function suitable for a wide variety of applications and is used by many software applications.

**Curve25519** works as follows according to the official documentation: “*given a user’s , computes the Given the user’s and another , computes a . This secret can then be used to authenticate and ”.*

After the generation of this private session key, **SunCrypt** jumps to the **curve function** to compute its **session public key** where this **session private key** is passed to the function along with a basepoint constant of 9 followed by all zeros. This **session public key will be converted to hex format and used as file extension** for the encrypted file **to allow the recovery** as each time a file is encrypted the **session private key** is destroyed. Next step of **SunCrypt** is to load its embedded **Curve25519** public key to compute a **shared secret**.

The analyzed sample md5 **0a0882b8da225406cc838991b5f67d11** contains the following public key in hex format:

```
c75d83161c3768477c859b15cfe3f6c7bf707976bfed511af7015d04f7066558
```

The other analyzed sample used in the blog to compare **SunCrypt** versions (MD5:**3d756f9715a65def4a302f5008b03809**) containing the improvement and obfuscation shared with most of the observed **SunCrypt** samples contains the public key:  
**695c567285a5b331dcf1d61bb291ce850e92c57111678fe79a2e5c2e399c9310**

The implementation of **Curve25519** looks manual and it’s likely to be using code from any of the multiple open source implementation of **Curve25519**,

The **shared secret** is computed by calling the **curve function** and passing the **attacker’s public key**, the generated **private key** and the **basepoint** as parameters. This **shared secret is used by another algorithm to encrypt the file**, allowing its recovery with the attacker’s **Curve25519** private key hosted in the **SunCrypt** C2 servers and the **session public key** that is the file extension of each file.

Summarising, each time a file is selected for encryption after the aforementioned checks, **SunCrypt** creates **Curve25519** session keys, stores each public key as **file extension** and **destroys the session private key**.

## SunCrypt key generation

session public key is formatted and used as extension, then the shared key is computed with the session private key and the attacker's public key

Once the **session keys** are created and the **shared key** is computed for a given file, **the required data is sent via completion I/O port to the encryption thread** which takes a pointer to the file and its new generated extension. To handle encryption threading, **SunCrypt** gets the **number of processors** and the maximum number of available **threads** that the operating system can allow to process I/O completion packets for I/O completion port.

This provide an efficient threading for multiple **asynchronous** I/O requests for encryption. Again, this was referenced in their forum ad.

## SunCrypt asynchronous multithreading

CreateIOCompletionPort initialisation

For the file encryption **SunCrypt** uses a very fast and secure algorithm which in this case is **ChaCha**. The implementation of **ChaCha20** stream cipher is also **manual** and makes sense its use, because it does not rely on **statically linked libraries** or the **CryptoAPI** avoiding suspicious calls and it's very fast and secure. This cipher is used by other popular ransomware like **Maze**. Again another similarity.

The actual file encryption happens after the above steps. The malware calls **ReadFile** function to get the content of the file and pushes a pointer to the buffer with the file content, the **file extension** and the computed **shared key** for that file to **ChaCha20** function that encrypts the file partially. This limitation is intended to speed up a bit more the encryption as is enough to make the file useless.



After reading the file, the flow redirects the data containing the computed shared key and the file to the ChaCha20 encryption routine

## C2 communications

---

**SunCrypt** contains a network module unlike many other ransomware and in my opinion this is also influenced by **Maze**. The ransomware uses the IP addresses that are embedded in the configuration and attempts to push different information from the compromised host.

It's worth noting that the observed IP addresses in **SunCrypt** are in many cases consistent with previous associated **Maze infrastructure's subnet**, reinforcing the idea of some sort of collaboration between ransomware dev groups.

**SunCrypt** gathers information from the **Windows** system gathering the **Minor Version**, **Major Version** and **Buildnumber** of the compromised system. The system survey also consists of gathering **username** and **hostname** information by calling **GetUserNameA** and **GetComputerA** functions.



SunCrypt information gathering

The network module gathers the information and creates a buffer that is later encoded with a hardcoded key

In the above image can be observed the information mentioned plus the fact of the use of this offset containing the bytes used for the mutex or implant id.

One of the things that raised my attention is that this step is identical between versions, meaning that these bytes are always used and intended to be sent in the network traffic, however, they are also used as mutex if the sample generates one, which is not always the



case. That's why I'm naming it implant-id and mutex at the same time. Is some sort of identifier that they may find useful.



Additionally, **SunCrypt** adds to this information (Versions and implant/mutex) the victim's information as mentioned above.

The **network buffer** contains the following example structure:

- 
- 
- 
- 
-

- 
- 
- 
- 
- 
- 

The above mentioned bytes are likely to be related to a manually implemented library used by the network routine to create the network buffer, however the library was not identified. Below an example of the network buffer prior to it's encoding using an hardcoded single-byte xor key 0x11.

Afterwards the C2 address that was stored during the configuration decoding is loaded and pushed into the stack to send all this information to the routine handling the HTTP request.

 SunCrypt network buffer

SunCrypt network buffer entering the encoding loop stage

Below an example of what **SunCrypt** HTTP traffic looks like in WireShark, containing the encoded data in the POST data.

 SunCrypt network traffic

SunCrypt network traffic

Another interesting detail that was mentioned before was the use of C2 infrastructure associated to the **Maze** team, reinforcing the idea of collaboration between ransomware teams which may indicate not only an stable relationship between threat actors but also a potential cooperation to share infrastructure, resources and techniques.

## Summary

---

After the analysis of the samples, can be appreciated some but not a lot of evolution and changes. So far only one sample was found using clear-text strings (md5:**0a0882b8da225406cc838991b5f67d11**) which contains vital strings such as the user-

agent and the drive letter whereas the rest of the identified samples contains some of the following modifications:

- Use of and hiding all the relevant data
- implementation
- 

From all the observed samples, most of them has been spotted using **PowerShell** as initial vector to **obfuscate the payload** and to **load the binary into memory** and “honours” all the functionalities that the developer team announce in their forum post like the **non-use of the CryptoAPI for file encryption** and **thread** features. By looking at the **PowerShell loaders** I personally found it similar to **Netwalker** PowerShell loaders, which makes me think that the **SunCrypt team** got some inspiration from them.

**SunCrypt** contains multiple manual implementations on the source code to avoid detections but despite the notable differences, it can be noticed that the team got a lot of inspiration from **Maze**. **SunCrypt** copied part of its cryptoscheme like the use of **ChaCha20** stream-cipher and **modified its session key implementation** by replacing **RSA** for the **Curve25519** algorithm probably for **two main reasons**:

- Key generation is way and uses than , which allows a very fast key generation and operation.
- . This allows to and . could be implemented manually, but I guess they considered that it wasn't probably worth.

Additionally, to **increase the encryption speed** the **SunCrypt** team sends file data via **completion I/O port** to the encryption thread.

Overall I have to say **SunCrypt** is not a very sophisticated ransomware unlike some of its competitors like **Maze**, **Egregor**, **Ragnar** or **Wastedlocker** and the main motivation I have to say this, is due to the **evolution, techniques, language(C over OOP like C++)** and **obfuscation** techniques point to this direction but they got a lot of inspiration to make something similar to **Maze**. In fact when I saw **SunCrypt** for the first time I thought: “*someone from the team just left and started its own project(?)*” but after diving into it and checking more samples I could notice that the level of sophistication and experience, tooling and techniques used are not the same, but I feel confident saying that **they were a source of inspiration for sure**.

However, what has in common with **Maze**? It shares some of the **command line arguments** , the use of a **network module for data exfiltration, algorithm for file encryption** and **other references**, however the **main differences with Maze are**:

- 
- 
- 
-

- 
- 
- 
- 

It will be interesting to keep an eye on this malware to see its evolution and modifications to survive as the **RaaS model service is a fast paced ecosystem** full of very capable competitors.

## MITRE ATT&CK

---

- 
- 
- 
- 
- 
- 

## Samples

---

- 
- 
- 
- 
- 

## C2

---

- 
-