

## Petya and Mischa: ransomware duet (part 2)

By hasherezade

Published: 2016-06-09 · Archived: 2026-04-06 00:57:43 UTC

After being [defeated](#) in April, [Petya](#) comes back with new tricks. Now, not as a single ransomware, but in a bundle with another malicious payload – Mischa. Both are named after the satellites from the [GoldenEye](#) movie.

They deploy attacks on different layers of the system and are used as alternatives. That's why, we decided to dedicate more than one post to this phenomenon. Welcome to part two! **The main focus of this analysis is Mischa and Setup.dll** (the malicious installer that chooses which payload to deploy).

The first part (about Green Petya) you can read about it [here](#).

**UPDATE: Improved version of Green Petya is out. [More details given in the new article](#).**

### Analyzed samples

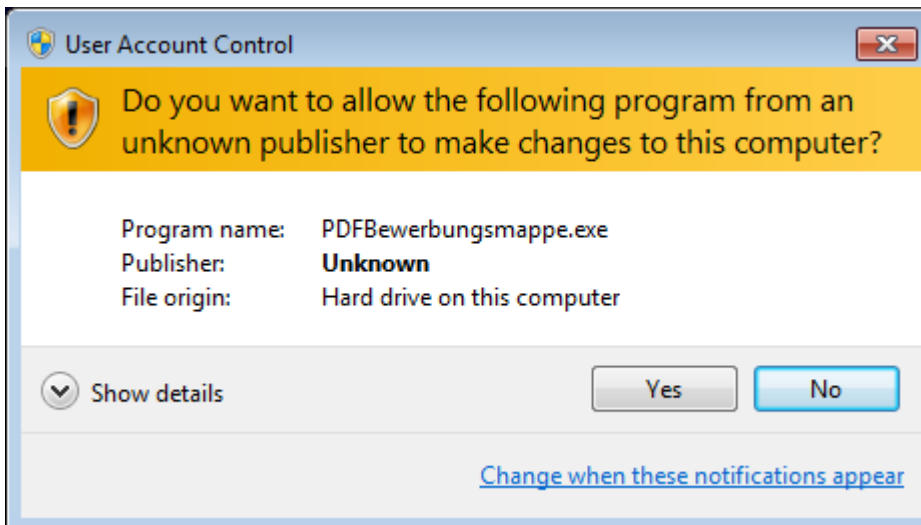
- [8a241cfcc23dc740e1fad7f2df3965e](#) – main executable
  - [c8e4829dcba8b288bd0ed75717214db6](#) – Setup.dll
    - [10b2d20a3c36fe6a5bf6f3b15149c3d1](#) – Mischa.dll (raw dump from the memory)
    - [34da44570eb8c7a5038370f553eb3899](#) – Mischa.dll (with filled section xxxx)

### Execution flow

- The [main executable](#) – a dropper [protected by a crypter/FUD](#):
  - unpack and deploy: [Setup.dll](#)
    - install: Petya
    - alternatively – deploy: Mischa.dll

### Behavioral analysis

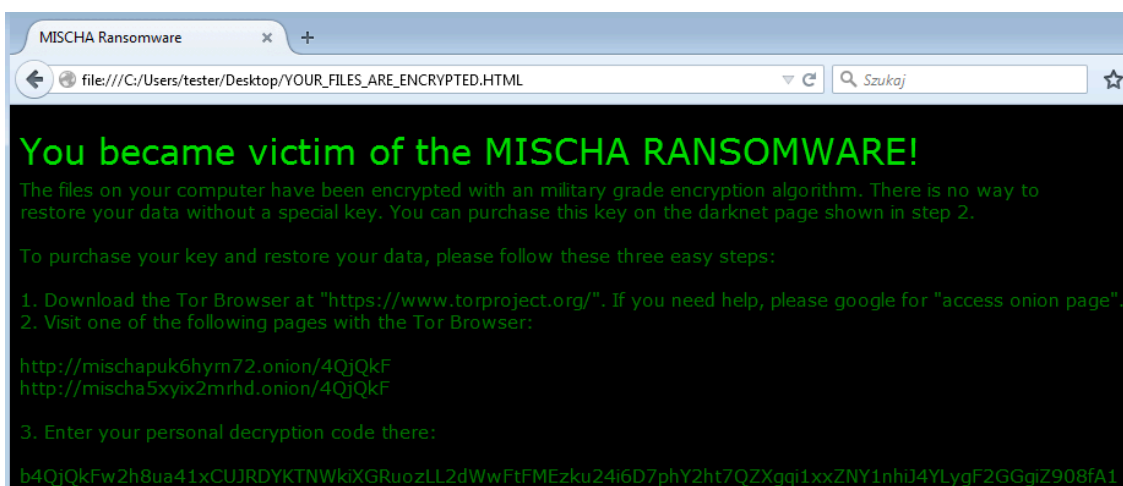
As mentioned in the [previous part of the article](#), both malicious payloads are dropped by the same dropper. The choice of which one will be used for the attack is made based on the privileges with which the sample is deployed. First, there is a request asking a user to elevate the application's privileges:



In case the user answered “Yes” to the question – his/her machine was getting infected by the Petya ransomware (described in details [here](#)).

But even in case the user was more cautious and didn’t allow to deploy payload with administrator privileges, it didn’t help much. Authors of the [malware](#) still found a way to attack the system. Just by launching another payload – Mischa, that does not require elevated privileges in order to work.

This payload works just like any other ransomware – encrypting files one by one and dropping a ransom note: *YOUR\_FILES\_ARE\_ENCRYPTED.HTML* (identical name was used before by another ransomware: [Chimera](#)). The layout is analogical to the one used by Petya.



The same text we can find in a dropped TXT file.

## Encryption process

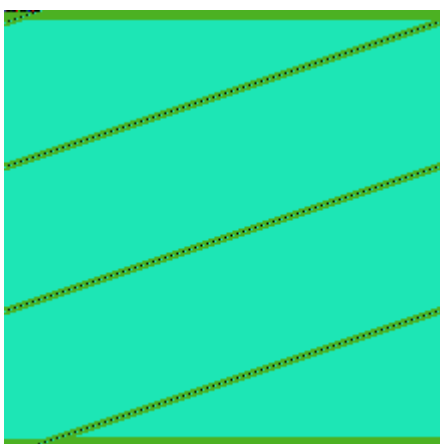
Mischa does not need to download a key from the CnC server – data can be encrypted offline as well. Extensions given to the encrypted files are random, generated at runtime (they are same like a part of the tor address):

Name	Date modified	Type	Size
square1 - Copy - Copy.bmp.7QzX	2016-05-12 18:47	7QZX File	141 KB
square1 - Copy.bmp.7QzX	2016-05-12 18:47	7QZX File	141 KB
square1.bmp.7QzX	2016-05-12 18:47	7QZX File	141 KB
YOUR_FILES_ARE_ENCRYPTED.HTML	2016-05-12 18:47	Firefox HTML Doc...	2 KB
YOUR_FILES_ARE_ENCRYPTED.TXT	2016-05-12 18:47	Text Document	1 KB

The atypical feature of Mischa is that it encrypts not only documents, but executables also (only few ransomware has been observed to do it).

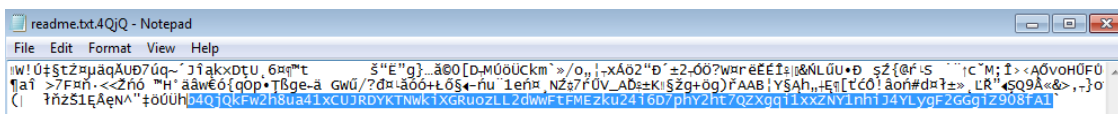
Entropy of encrypted samples is high and no patterns are visible. See below a visualization of bytes.

*square.bmp* : left – original, right encrypted with *Mischa*

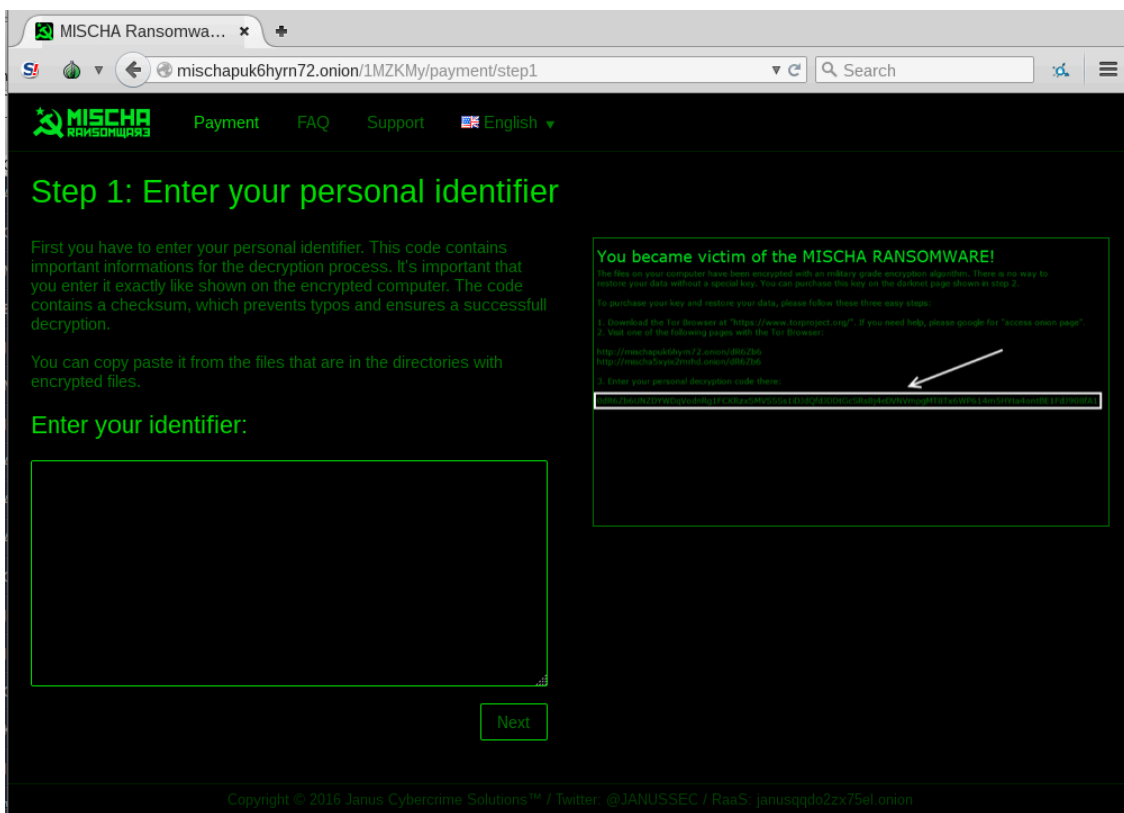


The same input does not produce the same output – that suggest that every file is encrypted with an individual key (or initialization vector).

At the end of every encrypted file, the unique ID is appended (like the one displayed in the ransom note):



Page for the victim:



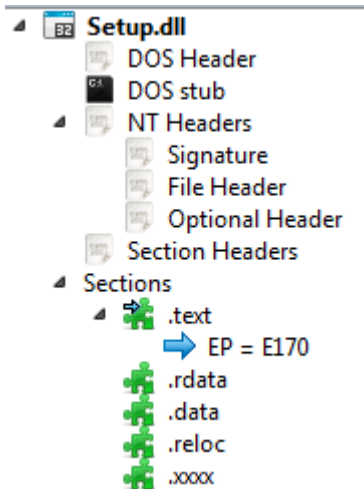
## Inside

The main executable (with an icon pretending a PDF document) is packed in an underground cryptor and its only role is to deliver and deploy the malicious core – Setup.dll. This exe's code doesn't make much sense for the functionality of the malware – it is only a deception layer added to create a noise and cover a real mission of the sample. Description of the packing will be omitted this time (it's very similar to the packing of the previous Petya).

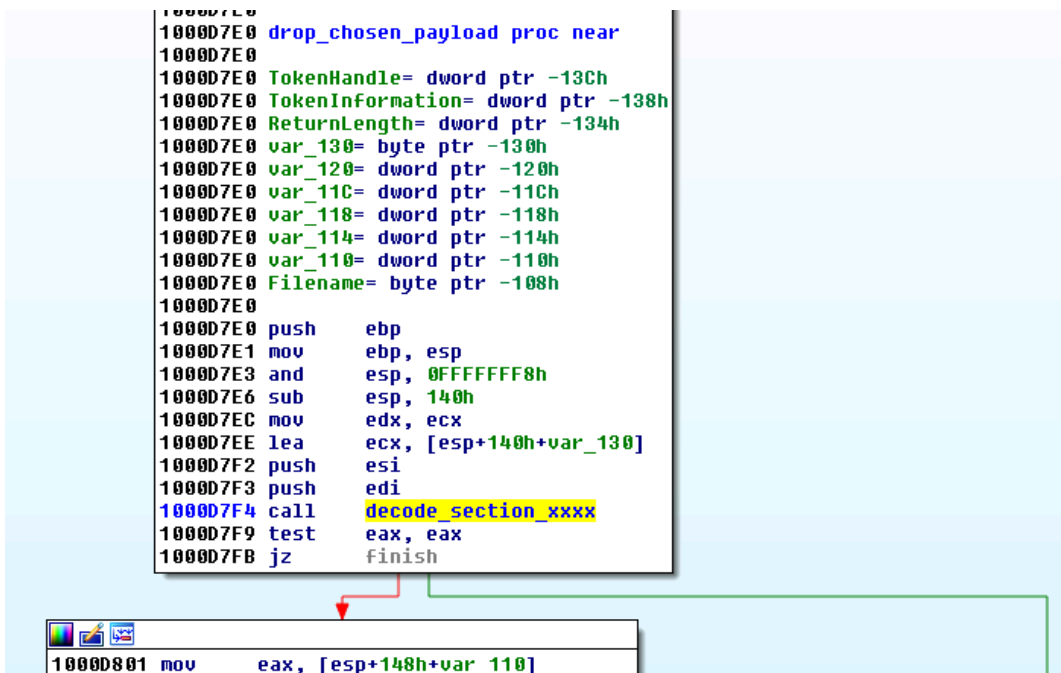
## Setup.dll

Setup.dll carry inside Petya and Mischa and decides which one of them will be dropped. This is the part of the malware is responsible for triggering the UAC popup.

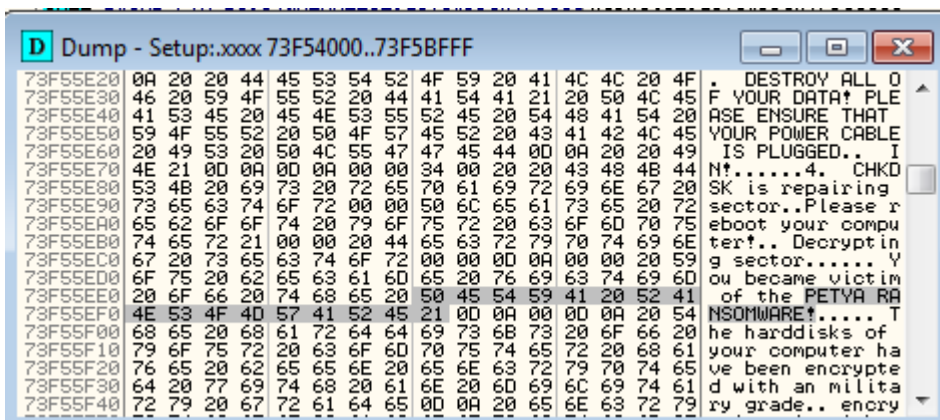
Similarly to the dropper of the previous Petya, it comes with a section .xxxx:



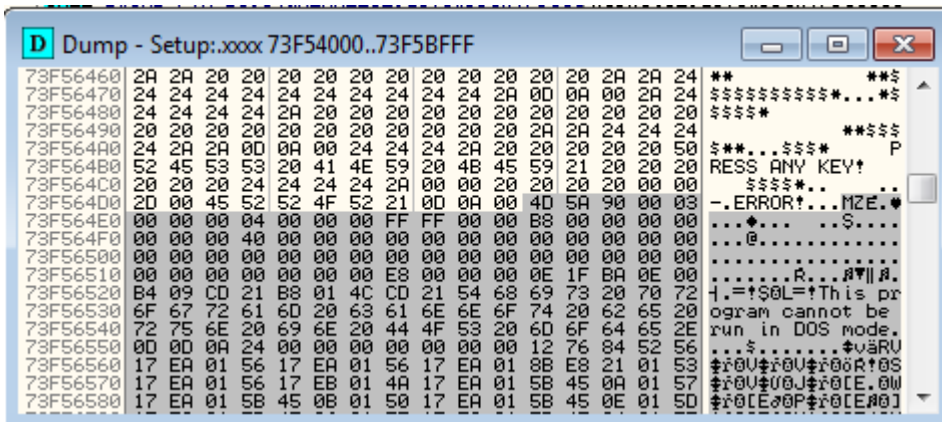
This section is very important, because it contains both payloads – Petya and Mischa (encrypted by simple XOR based algorithm). At the beginning of the execution they are being decrypted:



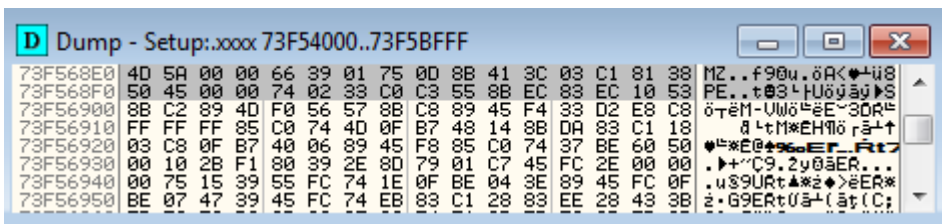
We can see a stub similar to the previous Petya:



In the same section a new PE file is revealed, that turns out to be a DLL of Mischa.

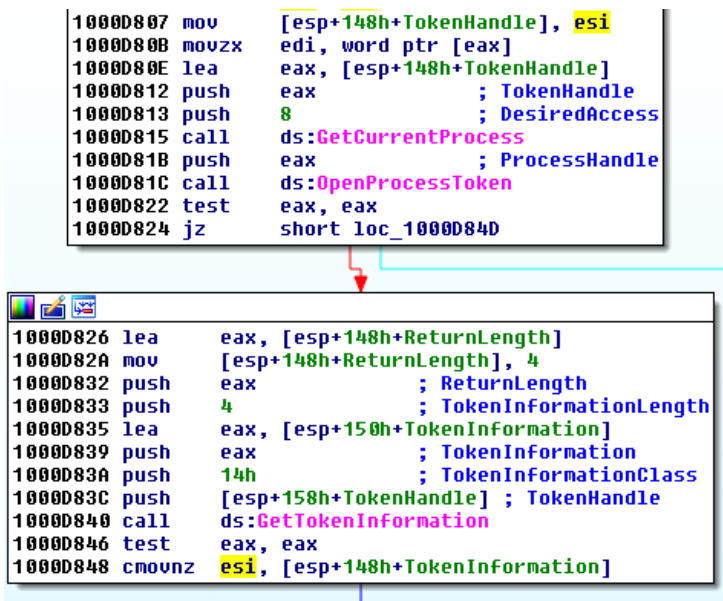


Authors tried to deceive tools for automated dumping of PE files from the memory, and provided fake “MZ”...”PE” patterns:



After decrypting the payloads, an environment check is performed in order to choose which one of them will be installed. The process token (resembling the privileges with which the sample was run) is used for choosing which installation path to follow next.

Reading the token of current process:



Choosing between **Petya** and **Mischa** is done in few steps. First, the token check is used to get information if the application is deployed with administrative rights. If it is not, then the it tries to run it’s new copy with higher

privileges (using *runas* command). If this attempt failed, Mischa is dropped (otherwise – Petya).

Dropper comes with a list of Anti-Malware products, which presence is checked before the payload is deployed:

Address	Hex dump	ASCII
73F4F7CF	59 5A 61 62 63 64 65 66 67 68 69 6A 6B 6C 6E 6F	VZabodefghijklmno
73F4F7DF	70 71 72 73 74 75 76 77 78 79 7A 00 00 53 48 41	pqrstuvwxyz..SHA
73F4F7EF	32 32 34 00 00 53 48 41 32 35 36 00 00 53 48 41	224..SHA256..SHA
73F4F7FF	33 38 34 00 00 53 48 41 35 31 32 00 00 41 68 6E	384..SHA512..Ahn
73F4F80F	4C 61 62 00 00 41 56 41 53 54 20 53 6F 66 74 77	Lab..AVAST Softw
73F4F81F	61 72 65 00 00 41 56 47 00 41 76 69 72 61 00 00	are..AVG.Avira..
73F4F82F	00 42 69 74 64 65 66 65 6E 64 65 72 00 42 75 6C	.Bitdefender.Bul
73F4F83F	6C 47 75 61 72 64 20 4C 74 64 00 00 00 43 68 65	lGuard Ltd...Che
73F4F84F	63 68 50 6F 69 6E 74 00 00 43 4F 40 4F 44 4F 00	ckPoint..COMODO.
73F4F85F	00 45 53 45 54 00 00 00 00 46 2D 53 65 63 75 72	.ESET...F-Secur
73F4F86F	65 00 00 00 00 47 20 44 41 54 41 00 00 4B 37 20	e...G DATA..K7
73F4F87F	43 6F 6D 70 75 74 69 6E 67 00 00 00 00 4B 61 73	Computing...Kas
73F4F88F	70 65 72 73 6B 79 20 4C 61 62 00 00 00 4D 61 6C	persky Lab...Mal
73F4F89F	77 61 72 65 62 79 74 65 73 20 41 6E 74 69 2D 4D	warebytes Anti-M
73F4F8AF	61 6C 77 61 72 65 00 00 00 4D 63 41 66 65 65 00	alware...McAfee.
73F4F8BF	00 4D 63 41 66 65 65 2E 63 6F 6D 00 00 4D 69 63	.McAfee.com..Mic
73F4F8CF	72 6F 73 6F 66 74 20 53 65 63 75 72 69 74 79 20	rosoft Security
73F4F8DF	43 6C 69 65 6E 74 00 00 00 4E 6F 72 6D 61 6E 00	Client...Norman.
73F4F8EF	00 50 61 6E 64 61 20 53 65 63 75 72 69 74 79 00	.Panda Security.
73F4F8FF	00 51 75 69 63 6B 20 48 65 61 6C 00 00 53 70 79	.Quick Heal..Spy
73F4F90F	62 6F 74 20 2D 20 53 65 61 72 63 68 20 26 20 44	bot - Search & D
73F4F91F	65 73 74 72 6F 79 20 32 00 53 70 79 62 6F 74 20	estroy 2.Spybot
73F4F92F	2D 20 53 65 61 72 63 68 20 26 20 44 65 73 74 72	- Search & Destr
73F4F93F	6F 79 00 00 00 4E 6F 72 74 6F 6E 20 53 65 63 75	oy...Norton Secu
73F4F94F	72 69 74 79 20 77 69 74 68 20 42 61 63 6B 75 70	ry with Backup
73F4F95F	00 4E 6F 72 74 6F 6E 20 53 65 63 75 72 69 74 79	.Norton Security
73F4F96F	00 4E 6F 72 74 6F 6E 49 6E 73 74 61 6C 6C 65 72	.NortonInstaller
73F4F97F	00 56 49 50 52 45 00 00 00 54 72 65 6E 64 20 4D	.VIPRE...Trend M
73F4F98F	69 63 72 6F 00 00 00 00 00 00 00 00 00 00 00	icro.....

Among strings we can see URLs for Petya as well as for Mischa. The below part of code is responsible for generating individual URLs for the particular victim and writing them into the payload:

```

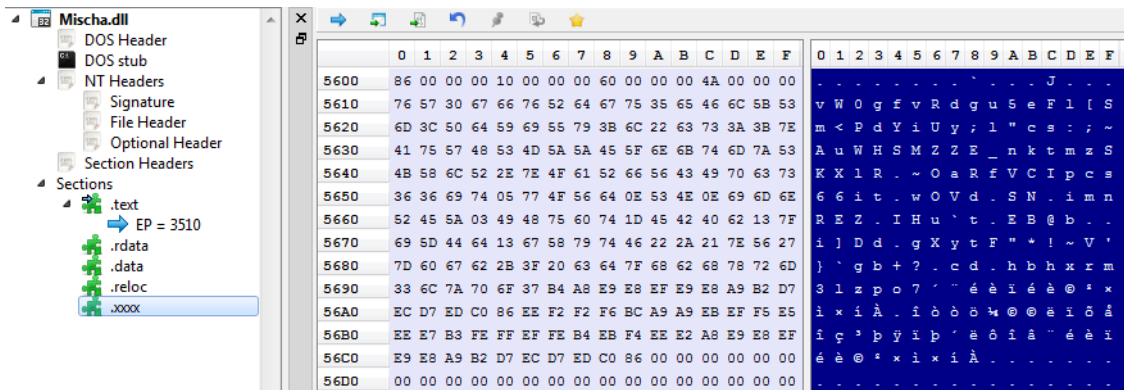
7116C9D0 . PUSH EDI
7116C9D1 . CALL Setup.7116ECB0
7116C9D6 . ADD ESP,4
7116C9D9 . LEA EDX,DWORD PTR DS:[ESI+1]
7116C9DC . MOV ECX,Setup.7116F9D8
7116C9E1 . CALL Setup.7116CE20
7116C9E6 . MOV DWORD PTR DS:[EBX+14],EAX
7116C9E9 . MOV ECX,Setup.7116F9F8
7116C9EE . MOV EDX,DWORD PTR DS:[EBX+10]
7116C9F1 . INC EDX
7116C9F2 . CALL Setup.7116CE20
7116C9F7 . MOV DWORD PTR DS:[EBX+18],EAX
7116C9FA . MOV ECX,Setup.7116FA18
7116C9FF . MOV EDX,DWORD PTR DS:[EBX+10]
7116CA02 . INC EDX
7116CA03 . CALL Setup.7116CE20
7116CA08 . MOV DWORD PTR DS:[EBX+1C],EAX
7116CA0B . MOV ECX,Setup.7116FA38
7116CA10 . MOV EDX,DWORD PTR DS:[EBX+10]
7116CA13 . INC EDX
7116CA14 . CALL Setup.7116CE20
7116CA18 . DBB EBX
    
```

```

[Arg1 = 0006FD24
Setup.7116ECB0
ASCII "http://petya3jxfp2f7g3i.onion/"
ASCII "http://petya3sen7dyko2n.onion/"
ASCII "http://mischapuk6hyrn72.onion/"
ASCII "http://mischa5xyix2mrhd.onion/"
Setup.7116F9D8
    
```

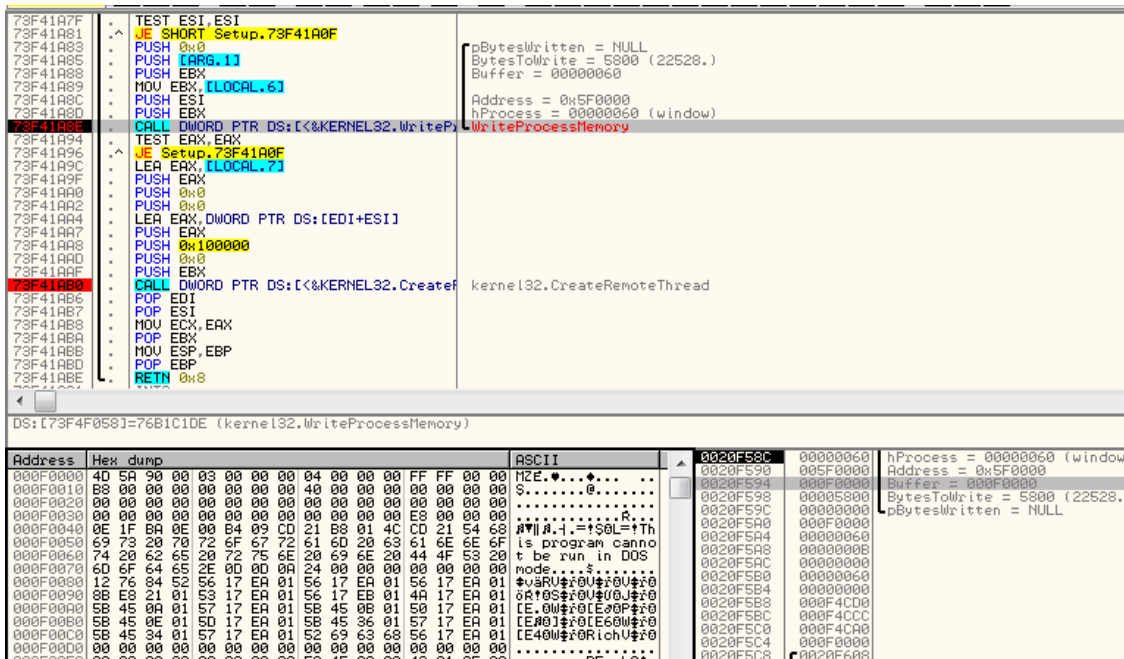
Inside the dropper, Mischa’s DLL (similarly to Petya’s stub) is being filled with additional, unique data. Similarly to Petya, Mischa gets a random key that will be used in further encryption process. This key is encrypted using ECC and transformed into a victim ID. Then, part of this victim ID becomes a part of the individual web address.

This unique data is generated by the dropper and (encrypted by a simple XOR based algorithm) stored in a new section – *.xxxx* – dynamically appended to the payload in the preparation phase. (If we dump Mischa too early, without this section, we will get incomplete data and the DLL will not run properly). See below – example of *Mischa.dll* with the added section:



At this stage, the victim ID that later is being displayed in the ransom note, as well as the onion addresses are ready.

After such preparation, Mischa.dll is injected to *conhost.exe* and deployed as a remote thread. Below, we can see the buffer containing the prepared Mischa.dll being written to the memory allocated in the remote process:



Execution (including encryption) continues in the remote thread.

### Mischa.dll

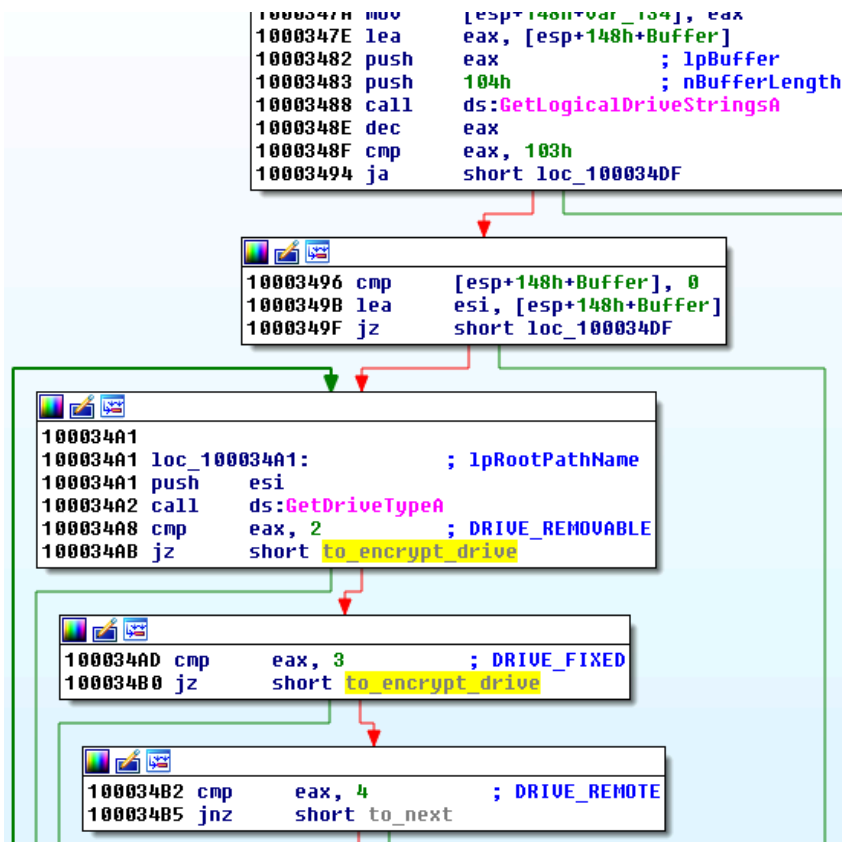
Again we can see a DLL using *ReflectiveLoader\** – just like in the case of [Chimera](#) and [Rokku](#) (along with other similarities in the code, it may confirm the theory, that authors behind those projects are the same):

Offset	Name	Value	Meaning	
4CA0	Characteristics	0		
4CA4	TimeDateStamp	5730A00E		
4CA8	MajorVersion	0		
4CAA	MinorVersion	0		
4CAC	Name	64D2	Mischa.dll	
4CB0	Base	1		
4CB4	NumberOfFunctions	1		
4CB8	NumberOfNames	1		
4CBC	AddressOfFunctions	64C8		
4CC0	AddressOfNames	64CC		
4CC4	AddressOfNameOrdinals	64D0		
Details				
Offset	Ordinal	Function RVA	Name RVA	Name
4CC8	1	1112	64DD	_ReflectiveLoader@4

**\*ReflectiveLoader** is a special stub belonging to the technique of [Reflective DLL Injection](#). This technique allows to produce a DLL that can be easily injected into another process. Similarly to a shellcode, such DLL is self-contained and automatically loads all its dependencies.

### What is attacked?

Mischa fetches the list of mapped drives ([GetLogicalDriveStringsA](#)) and identifies the drive type by a Windows API function: [GetDriveType](#). It attacks removable, fixed and remote drives.



Blacklisted paths:

Windows \$Recycle.Bin Microsoft Mozilla Firefox Opera Internet Explorer Temp Local LocalLow Chrome

Attacked extensions:

txt doc docx docm odt ods odp odf odc odm odb rtf xlsx xlsb xlk xls xlsx pps ppt pptm pptx pub epub

### How does the encryption work?

Every file is encrypted with a random key. First, using WindowsCryptoAPI function [CryptGenRandom](#) 128 random bits are fetched. Then, they are hashed and used to generate the initialization vector.

0FB62466	MOV DWORD PTR DS:[EDI],EBX	
0FB62468	CALL DWORD PTR DS:[&ADVAPI32.CryptAcquireContextA]	advapi32.CryptAcquireContextA
0FB6246E	TEST EAX,EAX	
0FB62470	JNZ SHORT Mischa.0FB62477	
0FB62472	PUSH -0x3C	
0FB62474	POP EAX	003B89E0
0FB62475	JMP SHORT Mischa.0FB624A0	
0FB62477	PUSH ESI	
0FB62478	PUSH [ARG_2]	
0FB6247B	MOV ESI,[ARG_3]	
0FB6247E	PUSH ESI	0x80 = 128
0FB6247F	PUSH [ARG_4]	
0FB62482	CALL DWORD PTR DS:[&ADVAPI32.CryptGenRandom]	advapi32.CryptGenRandom
0FB62488	TEST EAX,EAX	
0FB6248A	JNZ SHORT Mischa.0FB62491	
0FB6248C	PUSH -0x3C	
0FB6248E	POP EAX	003B89E0
0FB6248F	JMP SHORT Mischa.0FB6249F	
0FB62491	PUSH EBX	
0FB62492	PUSH [ARG_4]	
0FB62495	CALL DWORD PTR DS:[&ADVAPI32.CryptReleaseContext]	advapi32.CryptReleaseContext
0FB6249B	MOV DWORD PTR DS:[EDI],ESI	

Apart from the above few calls, Windows Crypto API is not used for the cryptography. Instead, all is implemented locally (just like in case of Chimera and Rokku). Below – fragment of the local implementation of function [SHA-256](#), containing typical constants:

```
100024A4 sub_100024A4 proc near
100024A4 xor     eax, eax
100024A6 mov     dword ptr [ecx+8], 6A09E667h
100024AD mov     [ecx], eax
100024AF mov     [ecx+4], eax
100024B2 mov     dword ptr [ecx+0Ch], 0BB67AE85h
100024B9 mov     dword ptr [ecx+10h], 3C6EF372h
100024C0 mov     dword ptr [ecx+14h], 0A54FF53Ah
100024C7 mov     dword ptr [ecx+18h], 510E527Fh
100024CE mov     dword ptr [ecx+1Ch], 9B05688Ch
100024D5 mov     dword ptr [ecx+20h], 1F83D9ABh
100024DC mov     dword ptr [ecx+24h], 5BE0CD19h
100024E3 mov     [ecx+68h], eax
100024E6 retn
100024E6 sub_100024A4 endp
```

File content is read in portions – 1024 bytes at once:

and then, encrypted by the locally implemented algorithm:



Encryption process is divided in 2 phases.

Phase 1:

Each 16 bytes of the read chunk is preprocessed by XOR with a 16 byte long buffer:

```

0F6636FE . MOV [LOCAL.2],ESI
0F663701 > LEA EDI,[LOCAL.438] content_buffer
0F663707 . XOR EDX,EDX
0F663709 . ADD EDI,EBX
0F66370B > LEA EAX,[LOCAL.20]
0F66370E . ADD EAX,EDX
0F663710 . MOV CL,BYTE PTR DS:[EDI+EAX] content[i]
0F663713 . XOR CL,BYTE PTR DS:[EAX] random_buf[i]
0F663715 . ADD EAX,EBX
0F663717 . INC EDX
0F663718 . MOV BYTE PTR SS:[EBP+EAX-0x0E0],CL output[i] = content_buffer[i] ^ random_buf[i]
0F66371F . CMP EDX,0x10 i < 16?
0F663722 . JLT SHORT Mischa.0F66370B
    
```

Address	Hex dump	ASCII
0018F630	B3 F2 BF 8A 31 CA C2 EA 4E 5E 22 87 F6 DA DC 68	1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
0018F640	C8 88 3D 00 28 05 00 00 28 05 00 00 00 00 00 00	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

At first, as the XOR key a random buffer is used. For next portions of data, the output of the second phase becomes the XOR key (it is a characteristics of [Cipher Block Chaining – CBC](#))

*Phase 2:*

The output of *phase 1* is passed to another encrypting function:

```

0FB636F7 . NEG EBX
0FB636F9 . LEA EAX,[LOCAL.20]
0FB636FC . ADD ESI,EAX
0FB636FE . MOV [LOCAL.2],ESI
0FB63701 > LEA EDI,[LOCAL.438] content_buffer
0FB63707 . XOR EDX,EDX
0FB63709 . ADD EDI,EBX
0FB6370B > LEA EAX,[LOCAL.20]
0FB6370E . ADD EAX,EDX
0FB63710 . MOV CL,BYTE PTR DS:[EDI+EAX] next_character
0FB63713 . XOR CL,BYTE PTR DS:[EAX]
0FB63715 . ADD EAX,EBX
0FB63717 . INC EDX
0FB63718 . MOV BYTE PTR SS:[EBP+EAX-0x0E0],CL out_buffer
0FB6371F . CMP EDX,0x10
0FB63722 . JLT SHORT Mischa.0FB6370B
0FB63724 . ADD ESI,EBX
0FB63726 . LEA ECX,[LOCAL.182]
0FB6372C . PUSH ESI
0FB6372D . MOV EDX,ESI
0FB6372E . CALL Mischa.0FB61956 crypt_block
0FB63734 . LEA EDI,[LOCAL.20]
0FB63737 . ADD EBX,0x10
    
```

Address	Hex dump	ASCII
001BE990	6A 00 C1 CE 4F FB 4C 88 71 25 AC F6 2D C4 B7 1D	j . 4 P O L r q % C + - E #
001BE9A0	0F D7 6A E9 E7 AB 89 0F A7 C0 51 14 9E EE B5 B9	* i j o s z e e z ' o q i x t A j
001BE9B0	4F FF 40 24 60 D4 2C 32 35 8E CD 01 2B 94 26 F5	o - o s ' o . s z i o
001BE9C0	DE F5 A9 0B AA 80 25 A8 C5 EA 69 20 C9 54 F6 98	0 s e d C x E t f i . f i t s e

This block cipher processes 16 bytes of the input and gives as a result 16 bytes of encrypted output. Encryption involves a 16 byte long key (that was hardcoded in the appended section) – in a given example it is **vW2ebtSboq7gBdUU**.

Notice the same key saved inside the .xxxx section (client ID – stored just after that – represents the encrypted form of this key, that only the attackers can decode):

```

Dump - Mischa:xxxx0FB6B000..0FB6BFFF
0FB6B000 86 00 00 00 10 00 00 00 60 00 00 00 4A 00 00 00 c . . . . .
0FB6B010 76 57 32 65 62 74 53 62 6F 71 37 67 42 64 55 55 vW2ebtSboq7gBdUU
0FB6B020 62 34 51 6A 51 6B 46 77 32 68 38 75 61 34 31 78 b40jQkFw2h8u a41x
0FB6B030 43 55 4A 52 44 59 4B 54 4E 57 68 69 58 47 52 75 CUJRDYKTHWk iXGRu
0FB6B040 6F 7A 4C 4C 32 64 57 77 46 74 46 4D 45 7A 68 75 ozLL2dWwFtFMEzku
0FB6B050 32 34 69 36 44 37 70 68 59 32 68 74 37 51 5A 58 24i6D7phV2ht7Q2X
0FB6B060 67 71 69 31 78 78 5A 4E 59 31 6E 68 69 4A 34 59 gqiixx2NY1nh iJ4Y
0FB6B070 4C 79 67 46 32 47 47 67 69 5A 39 30 38 66 41 31 LygF2G6giZ908fA1
0FB6B080 68 74 74 70 3A 2F 2F 6D 69 73 63 68 61 70 75 68 http://mischapuk
0FB6B090 36 68 79 72 6E 37 32 2E 6F 6E 69 6F 6E 2F 34 51 6hyrn72.onion/4Q
0FB6B0A0 6A 51 6B 46 00 68 74 74 70 3A 2F 2F 6D 69 73 63 jQkF.http://misc
0FB6B0B0 68 61 35 78 79 69 78 32 6D 72 68 64 2E 6F 6E 69 ha5xy ix2mrhd.oni
0FB6B0C0 6F 6E 2F 34 51 6A 51 6B 46 00 00 00 00 00 00 00 on/4QjQkF.....
    
```

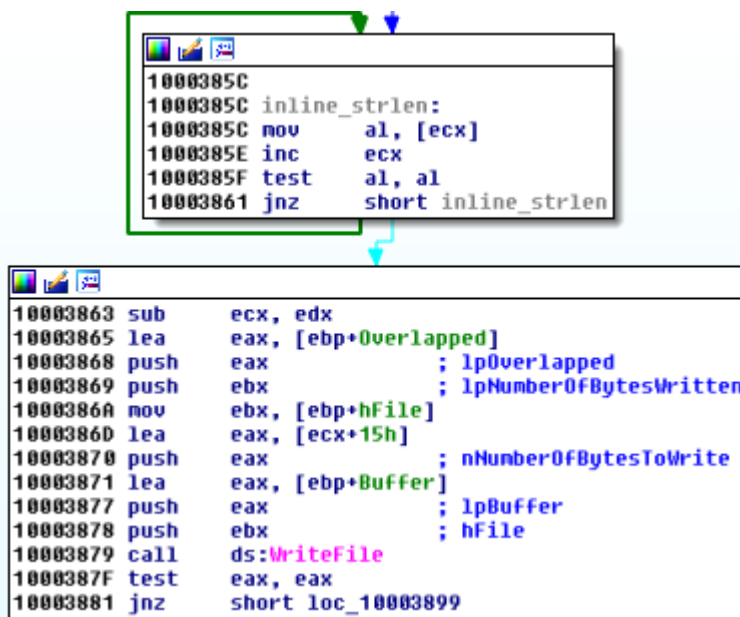
As long as Mischa is running, this key is in memory in open text. But once it finishes, this data is being destroyed and only the encrypted form of the key is left – user receives it in the ransom note. (It is somehow similar to the

logic of Petya).

Encrypted chunks are being written into the file one by one:

```
10003765 mov     esi, eax
10003767 mov     edi, 1024
1000376C lea     eax, [ebp+Overlapped]
1000376F mov     [ebp+Overlapped.hEvent], esi
10003772 push   eax           ; lpOverlapped
10003773 push   0             ; lpNumberOfBytesWritten
10003775 push   edi           ; nNumberOfBytesToWrite
10003776 lea     eax, [ebp+encBuffer]
1000377C push   eax           ; lpBuffer
1000377D push   [ebp+hFile]   ; hFile
10003780 call   ds:WriteFile
```

After the full file is encrypted and the content stored, additional data is appended at the end.



```
1000385C
1000385C inline_strlen:
1000385C mov     al, [ecx]
1000385E inc     ecx
1000385F test    al, al
10003861 jnz    short inline_strlen

10003863 sub     ecx, edx
10003865 lea     eax, [ebp+Overlapped]
10003868 push   eax           ; lpOverlapped
10003869 push   ebx           ; lpNumberOfBytesWritten
1000386A mov     ebx, [ebp+hFile]
1000386D lea     eax, [ecx+15h]
10003870 push   eax           ; nNumberOfBytesToWrite
10003871 lea     eax, [ebp+Buffer]
10003877 push   eax           ; lpBuffer
10003878 push   ebx           ; hFile
10003879 call   ds:WriteFile
1000387F test    eax, eax
10003881 jnz    short loc_10003899
```

Then, file is moved under the new name.

```
0F6638C0 | . LEA EAX, [LOCAL_504]
0F6638C3 | . PUSH EAX
0F6638D4 | . CALL DWORD PTR DS:[&KERNEL32.MoveFileA]
0F6638D5 | . MOV EDX, EDI
0F6638D8 | . MOV ECX, ESI
```

[NewName = "C:\\pin\\extras\\pinadx-vsplugin\\readme.txt.40j0"  
ExistingName = "C:\\pin\\extras\\pinadx-vsplugin\\readme.txt"  
MoveFileA

Let's have a look at the appended data and it's role in decoding the file. At the end of the encrypted file we can find:

1. Length of the original file (0x528 -> 1320)

```

readme.txt.4QjQ
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
000007D0 9B 9A F6 15 91 B5 04 1F 5E AB 24 DF 37 BC 29 31 >šö. 'µ..^«$B7L)1
000007E0 AB AE E6 B3 F7 33 60 7F 6A 97 FE 04 BB FE 31 DF «@ćł÷3`.j-ť.»ť1š
000007F0 62 8D CF C1 5F 77 9A 13 E4 2F 45 A8 0B 56 21 00 bTĎÁ wš.ä/E".V!..
00000800 28 05 00 00 B3 F2 BF 8A 31 CA C2 EA 4E 5E 22 87 (...iňžš1EĀēN^"+
00000810 F6 DA DC 68 62 34 51 6A 51 6B 46 77 32 68 38 75 ôŮŮhb4QjQkFw2h8u
00000820 61 34 31 78 43 55 4A 52 44 59 4B 54 4E 57 6B 69 a41xCUJRDKTNWki
00000830 58 47 52 75 6F 7A 4C 4C 32 64 57 77 46 74 46 4D XGRuozLL2dWwFtFM
00000840 45 7A 6B 75 32 34 69 36 44 37 70 68 59 32 68 74 Ezku24i6D7phY2ht
00000850 37 51 5A 58 67 71 69 31 78 78 5A 4E 59 31 6E 68 7QZXgqi1xxZNY1nh
00000860 69 4A 34 59 4C 79 67 46 32 47 47 67 69 5A 39 30 iJ4YLygF2GGgiZ90
00000870 38 66 41 31 60 8fA1`

```

2. Initialization vector – the random buffer of 16 bytes, that was used to initialize the XOR cycle:

```

readme.txt.4QjQ
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
000007C0 3D F2 56 3B 79 87 6D 7D AD C3 FA 9B D4 66 59 89 =ñV;y+m).Ăú>ÔfY%
000007D0 9B 9A F6 15 91 B5 04 1F 5E AB 24 DF 37 BC 29 31 >šö. 'µ..^«$B7L)1
000007E0 AB AE E6 B3 F7 33 60 7F 6A 97 FE 04 BB FE 31 DF «@ćł÷3`.j-ť.»ť1š
000007F0 62 8D CF C1 5F 77 9A 13 E4 2F 45 A8 0B 56 21 00 bTĎÁ wš.ä/E".V!..
00000800 28 05 00 00 B3 F2 BF 8A 31 CA C2 EA 4E 5E 22 87 (...iňžš1EĀēN^"+
00000810 F6 DA DC 68 62 34 51 6A 51 6B 46 77 32 68 38 75 ôŮŮhb4QjQkFw2h8u
00000820 61 34 31 78 43 55 4A 52 44 59 4B 54 4E 57 6B 69 a41xCUJRDKTNWki
00000830 58 47 52 75 6F 7A 4C 4C 32 64 57 77 46 74 46 4D XGRuozLL2dWwFtFM
00000840 45 7A 6B 75 32 34 69 36 44 37 70 68 59 32 68 74 Ezku24i6D7phY2ht
00000850 37 51 5A 58 67 71 69 31 78 78 5A 4E 59 31 6E 68 7QZXgqi1xxZNY1nh
00000860 69 4A 34 59 4C 79 67 46 32 47 47 67 69 5A 39 30 iJ4YLygF2GGgiZ90
00000870 38 66 41 31 60 8fA1`

```

3. Client ID (as mentioned before) – that is encrypted key which was used for the second encryption operation. In the above example, this key was: **vW2ebtSboq7gBdUU**

```

readme.txt.4QjQ
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
000007C0 3D F2 56 3B 79 87 6D 7D AD C3 FA 9B D4 66 59 89 =ñV;y+m).Ăú>ÔfY%
000007D0 9B 9A F6 15 91 B5 04 1F 5E AB 24 DF 37 BC 29 31 >šö. 'µ..^«$B7L)1
000007E0 AB AE E6 B3 F7 33 60 7F 6A 97 FE 04 BB FE 31 DF «@ćł÷3`.j-ť.»ť1š
000007F0 62 8D CF C1 5F 77 9A 13 E4 2F 45 A8 0B 56 21 00 bTĎÁ wš.ä/E".V!..
00000800 28 05 00 00 B3 F2 BF 8A 31 CA C2 EA 4E 5E 22 87 (...iňžš1EĀēN^"+
00000810 F6 DA DC 68 62 34 51 6A 51 6B 46 77 32 68 38 75 ôŮŮhb4QjQkFw2h8u
00000820 61 34 31 78 43 55 4A 52 44 59 4B 54 4E 57 6B 69 a41xCUJRDKTNWki
00000830 58 47 52 75 6F 7A 4C 4C 32 64 57 77 46 74 46 4D XGRuozLL2dWwFtFM
00000840 45 7A 6B 75 32 34 69 36 44 37 70 68 59 32 68 74 Ezku24i6D7phY2ht
00000850 37 51 5A 58 67 71 69 31 78 78 5A 4E 59 31 6E 68 7QZXgqi1xxZNY1nh
00000860 69 4A 34 59 4C 79 67 46 32 47 47 67 69 5A 39 30 iJ4YLygF2GGgiZ90
00000870 38 66 41 31 60 8fA1`

```

Having the important pieces of data – initial XOR buffer and the decrypted key – full process of encryption can be reversed by the attackers.

## Conclusion

Mischa, in contrast to Petya, is yet another typical ransomware. It is well packed and written cleanly, but the core looks simple. We didn't find any novel or unexpected features inside. It seems like the main focus of the authors was Petya, and Mischa was added just as a failsafe. However, even if it is simple, it plays the planned role pretty well. When the user rejected the request of elevating application privileges, he/she will probably not expect the application to be running at all. But this is the event that makes Mischa deploy its sneaky attack. In fact it may have more painful consequences than the attack of Petya. In case of Petya, some part of the disk content can be recovered using forensics tools – but with Mischa it is not possible.

## Appendix

<http://www.bleepingcomputer.com/news/security/petya-is-back-and-with-a-friend-named-mischa-ransomware/> – Bleeping Computer about Mischa

</blog/threat-analysis/2016/04/petya-ransomware/> – about the previous version of Petya

### **Petya and Mischa – Ransomware Duet (Part 1):**

</blog/threat-analysis/2016/05/petya-and-mischa-ransomware-duet-p1/>

## About the author

Unpacks malware with as much joy as a kid unpacking candies.

---

Source: <https://www.malwarebytes.com/blog/news/2016/06/petya-and-mischa-ransomware-duet-p2>