

# Petya and Mischa: ransomware duet (part 2)

---

 malwarebytes.com/blog/news/2016/06/petya-and-mischa-ransomware-duet-p2

hasherezade

June 9, 2016

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

---

## Execution flow

---

The main executable – a dropper protected by a crypter/FUD:

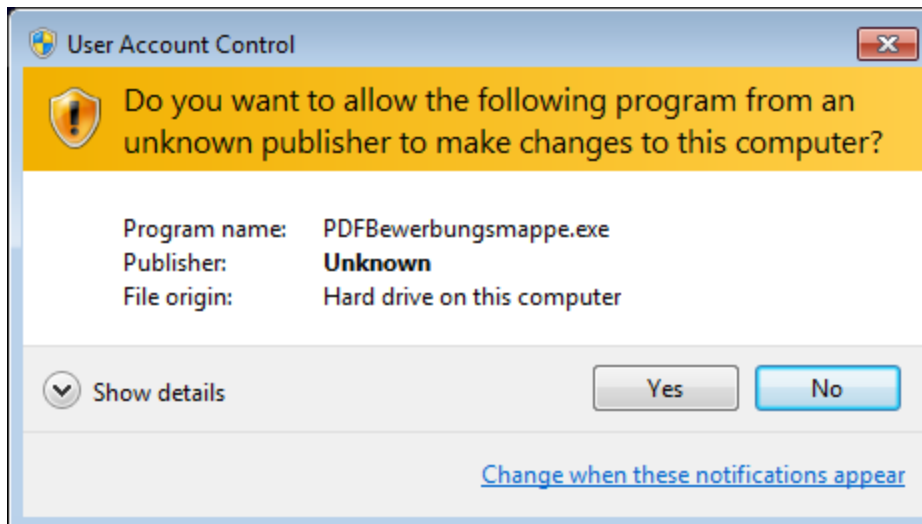
unpack and deploy: Setup.dll

- install: Petya
- alternatively – deploy: Mischa.dll

## Behavioral analysis

---

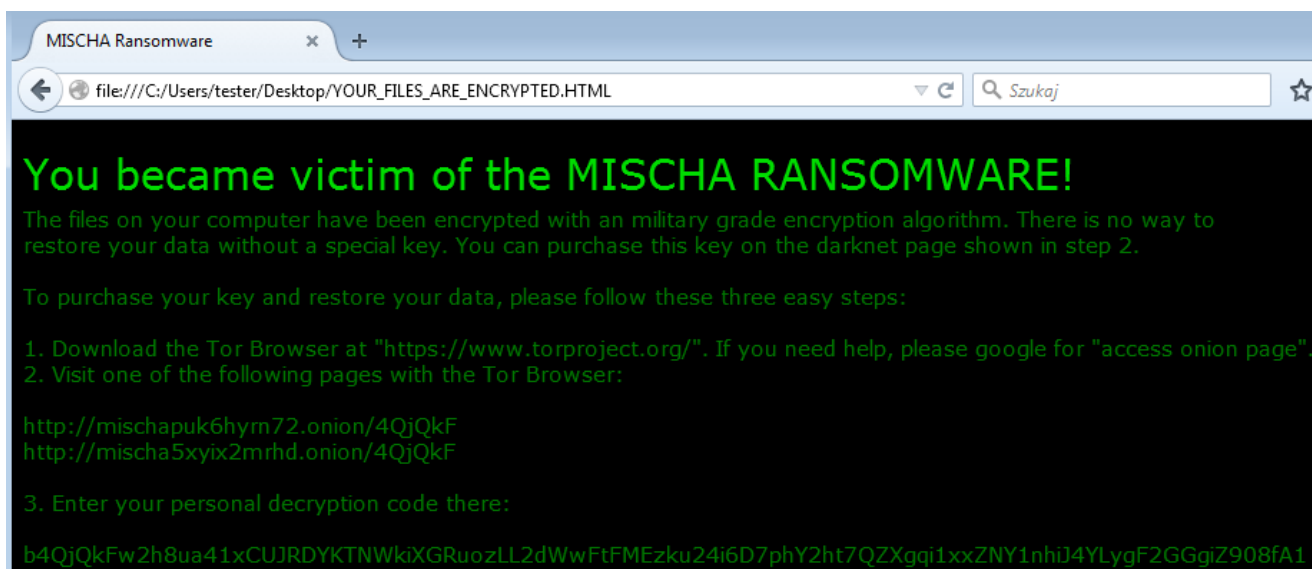
As mentioned in the previous part of the article, both malicious payloads are dropped by the same dropper. The choice of which one will be used for the attack is made based on the privileges with which the sample is deployed. First, there is a request asking a user to elevate the application's privileges:



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

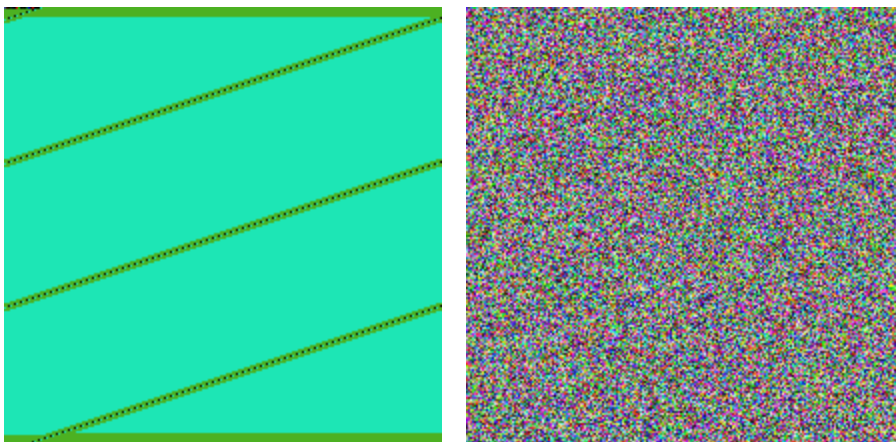
Mischa does not need to download a key from the CnC server – data can be encrypted offline as well. Extensions given to the encrypted files are random, generated at runtime (they are same like a part of the tor address):

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).

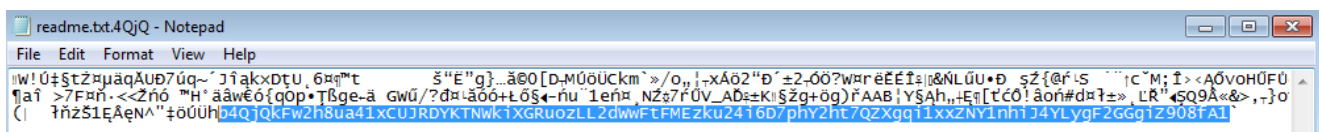
Entropy of encrypted samples is high and no patterns are visible. See below a visualization of bytes.

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

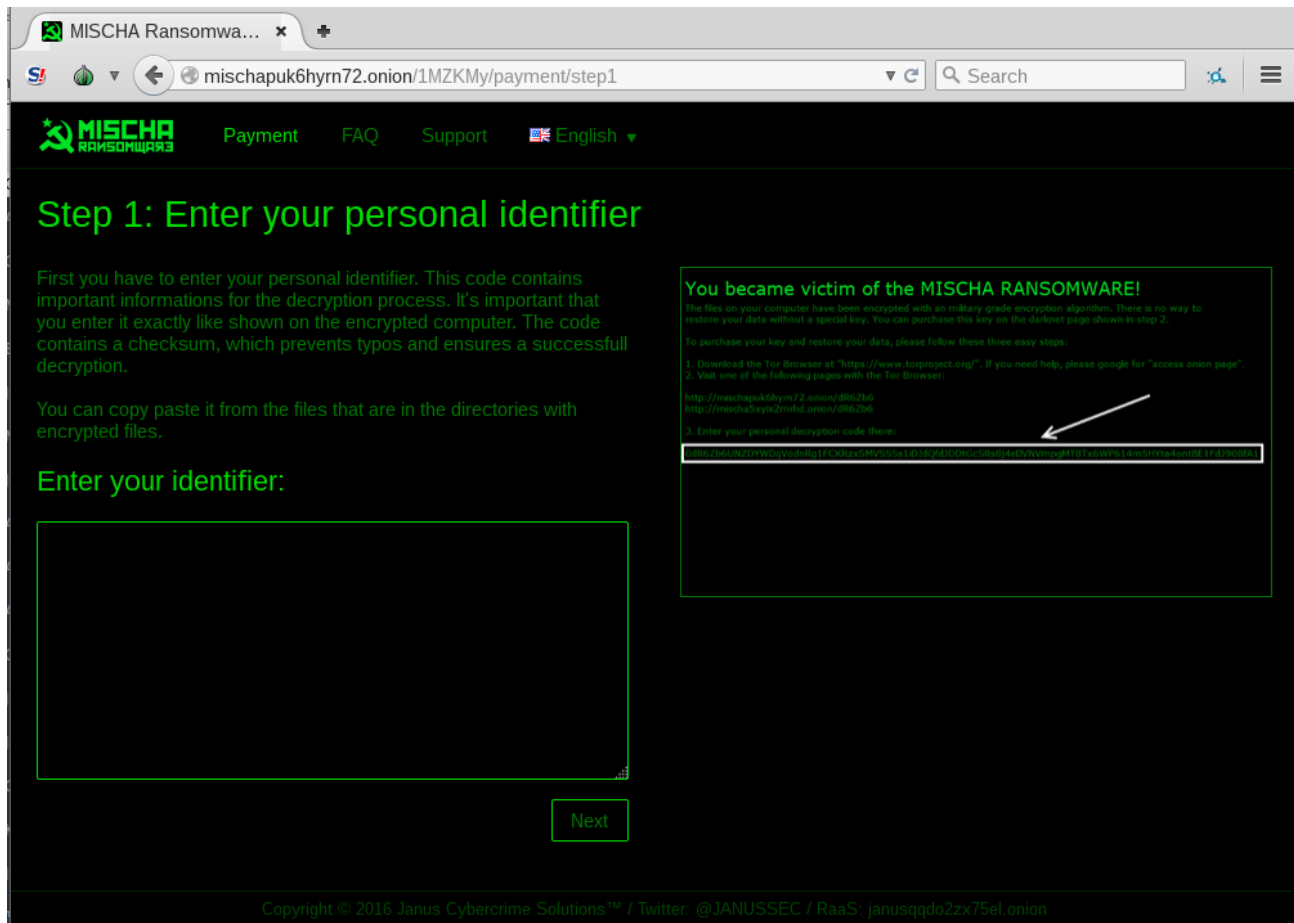


The same input does not produce the same output – that suggest that every file is encrypted with an individual key (or initialization vector).

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



Page for the victim:



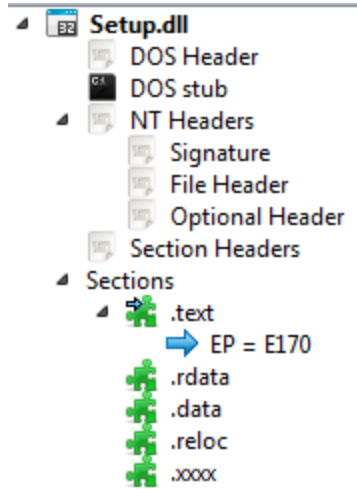
## Inside

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:

```

100007E0
100007E0 drop_chosen_payload proc near
100007E0
100007E0 TokenHandle= dword ptr -13Ch
100007E0 TokenInformation= dword ptr -138h
100007E0 ReturnLength= dword ptr -134h
100007E0 var_130= byte ptr -130h
100007E0 var_120= dword ptr -120h
100007E0 var_11C= dword ptr -11Ch
100007E0 var_118= dword ptr -118h
100007E0 var_114= dword ptr -114h
100007E0 var_110= dword ptr -110h
100007E0 Filename= byte ptr -108h
100007E0
100007E0 push    ebp
100007E1 mov     ebp, esp
100007E3 and     esp, 0FFFFFFF8h
100007E6 sub     esp, 140h
100007EC mov     edx, ecx
100007EE lea    ecx, [esp+140h+var_130]
100007F2 push   esi
100007F3 push   edi
100007F4 call   decode_section_xxxx
100007F9 test   eax, eax
100007FB jz     finish

```

```

10000801 mov     eax, [esp+148h+var_110]

```

We can see a stub similar to the previous Petya:

```

D Dump - Setup:..xxxx 73F54000..73F5BFFF
73F55E20 0A 20 20 44 45 53 54 52 4F 59 20 41 4C 4C 20 4F . DESTROY ALL O
73F55E30 46 20 59 4F 55 52 20 44 41 54 41 21 20 50 4C 45 F YOUR DATA! PLE
73F55E40 41 53 45 20 45 4E 53 55 52 45 20 54 48 41 54 20 ASE ENSURE THAT
73F55E50 59 4F 55 52 20 50 4F 57 45 52 20 43 41 42 4C 45 YOUR POWER CABLE
73F55E60 20 49 53 20 50 4C 55 47 47 45 44 00 0A 20 20 49 IS PLUGGED.. I
73F55E70 4E 21 00 0A 00 0A 00 00 34 00 20 20 43 48 4B 44 N!.....4. CHKD
73F55E80 53 4B 20 69 73 20 72 65 70 61 69 72 69 6E 67 20 SK is repairing
73F55E90 73 65 63 74 6F 72 00 00 50 6C 65 61 73 65 20 72 sector..Please r
73F55EA0 65 62 6F 6F 74 20 79 6F 75 72 20 63 6F 6D 70 75 eboot your compu
73F55EB0 74 65 72 21 00 00 20 44 65 63 72 79 70 74 69 6E ter!.. Decryptin
73F55EC0 67 20 73 65 63 74 6F 72 00 00 00 0A 00 00 20 59 g sector..... Y
73F55ED0 6F 75 20 62 65 63 61 60 65 20 76 69 63 74 69 6D ou became victim
73F55EE0 20 6F 66 20 74 68 65 20 50 45 54 59 41 20 52 41 of the PETVA RA
73F55EF0 4E 53 4F 40 57 41 52 45 21 00 0A 00 00 20 54 NSOMWARE!..... T
73F55F00 68 65 20 68 61 72 64 64 69 73 68 73 20 6F 66 20 he harddisks of
73F55F10 79 6F 75 72 20 63 6F 6D 70 75 74 65 72 20 68 61 your computer ha
73F55F20 76 65 20 62 65 65 6E 20 65 6E 63 72 79 70 74 65 ve been encrypte
73F55F30 64 20 77 69 74 68 20 61 6E 20 6D 69 6C 69 74 61 d with an milita
73F55F40 72 79 20 67 72 61 64 65 00 0A 20 65 6E 63 72 79

```

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

```

D Dump - Setup:..xxxx 73F54000..73F5BFFF
73F56460 2A 2A 20 20 20 20 20 20 20 20 20 2A 2A 24 **
73F56470 24 24 24 24 24 24 24 24 24 2A 00 0A 00 2A 24 ****
73F56480 24 24 24 2A 2A 20 20 20 20 20 20 20 20 20 20 ****
73F56490 20 20 20 20 20 20 20 20 20 20 20 2A 2A 24 24 ****
73F564A0 24 2A 2A 00 0A 00 24 24 24 2A 20 20 20 20 20 50 ***...$$$* P
73F564B0 52 45 53 53 20 41 4E 59 20 4E 45 59 21 20 20 20 RES$ ANY KEY!
73F564C0 20 20 20 24 24 24 2A 00 00 20 20 20 20 20 00 00 $$$*..
73F564D0 2D 00 45 52 52 4F 52 21 00 0A 00 4D 5A 90 00 03 -.ERROR?...MZE.
73F564E0 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 ..+.
73F564F0 00 00 00 40 00 00 00 00 00 00 00 00 00 00 00 ..@.
73F56500 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..
73F56510 00 00 00 00 00 00 00 E8 00 00 00 0E 1F BA 0E 00 .....R...A||#
73F56520 B4 09 CD 21 B8 01 4C CD 21 54 68 69 73 20 70 72 +.=!$@L=!This pr
73F56530 6F 67 72 61 6D 20 63 61 6E 6E 6F 74 20 62 65 20 ogram cannot be
73F56540 72 75 6E 20 69 6E 20 44 4F 53 20 6D 6F 64 65 2E run in DOS mode.
73F56550 00 0D 0A 24 00 00 00 00 00 00 00 12 76 84 52 56 ...$.
73F56560 17 EA 01 56 17 EA 01 56 17 EA 01 88 E8 21 01 53 $t0U$t0U$t00Rt0S
73F56570 17 EA 01 56 17 EB 01 4A 17 EA 01 58 45 0A 01 57 $t0U$t00J$t0LE.0U
73F56580 17 EA 01 5B 45 0B 01 50 17 EA 01 5B 45 0E 01 5D $t0LE00P$t0LE001

```

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

```

D Dump - Setup:..xxxx 73F54000..73F5BFFF
73F568E0 4D 5A 00 00 66 39 01 75 0D 8B 41 3C 03 C1 81 38 MZ..f90u.0A<+u8
73F568F0 50 45 00 00 74 02 33 C0 C3 55 8B EC 83 EC 10 53 PE..t03+U0y0y0S
73F56900 8B C2 89 4D F0 56 57 8B C8 89 45 F4 33 D2 E8 C8 0t0M-U000E-30Rt
73F56910 FF FF FF 85 C0 74 4D 0F B7 48 14 8B DA 83 C1 18 0tM*EHt0 r0t+
73F56920 03 C8 0F B7 40 06 89 45 F8 85 C0 74 37 BE 60 50 *E0+*0ER.Ft
73F56930 00 10 2B F1 80 39 2E 8D 79 01 C7 45 FC 2E 00 00 .+<C9.2y00ER...
73F56940 00 75 15 39 55 FC 74 1E 0F BE 04 3E 89 45 FC 0F .u09URt0*00EER*
73F56950 BE 07 47 39 45 FC 74 EB 83 C1 28 83 EE 28 43 38 0t09ERt00t0t(C;

```

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:

```

1000D807 mov     [esp+148h+TokenHandle], esi
1000D80B movzx  edi, word ptr [eax]
1000D80E lea   eax, [esp+148h+TokenHandle]
1000D812 push  eax                ; TokenHandle
1000D813 push  8                  ; DesiredAccess
1000D815 call  ds:GetCurrentProcess
1000D81B push  eax                ; ProcessHandle
1000D81C call  ds:OpenProcessToken
1000D822 test  eax, eax
1000D824 jz    short loc_1000D84D

```

```

1000D826 lea   eax, [esp+148h+ReturnLength]
1000D82A mov   [esp+148h+ReturnLength], 4
1000D832 push  eax                ; ReturnLength
1000D833 push  4                  ; TokenInformationLength
1000D835 lea   eax, [esp+150h+TokenInformation]
1000D839 push  eax                ; TokenInformation
1000D83A push  14h                ; TokenInformationClass
1000D83C push  [esp+158h+TokenHandle] ; TokenHandle
1000D840 call  ds:GetTokenInformation
1000D846 test  eax, eax
1000D848 cmovnz esi, [esp+148h+TokenInformation]

```

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 6D 6E 6F	VZabcdefghijklmnopqrstuvwxyz..SHA
73F4F7DF	70 71 72 73 74 75 76 77 78 79 7A 00 00 53 48 41	pqrstuvwxyz..SHA
73F4F7EF	32 33 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 4D 4F 44 4F 00	ckPoint..COMODO.
73F4F85F	00 45 53 45 54 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 4B 61 73	Computing...Kas
73F4F88F	70 65 72 73 6B 79 20 4C 61 62 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	rity 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 | . POP EDI

```

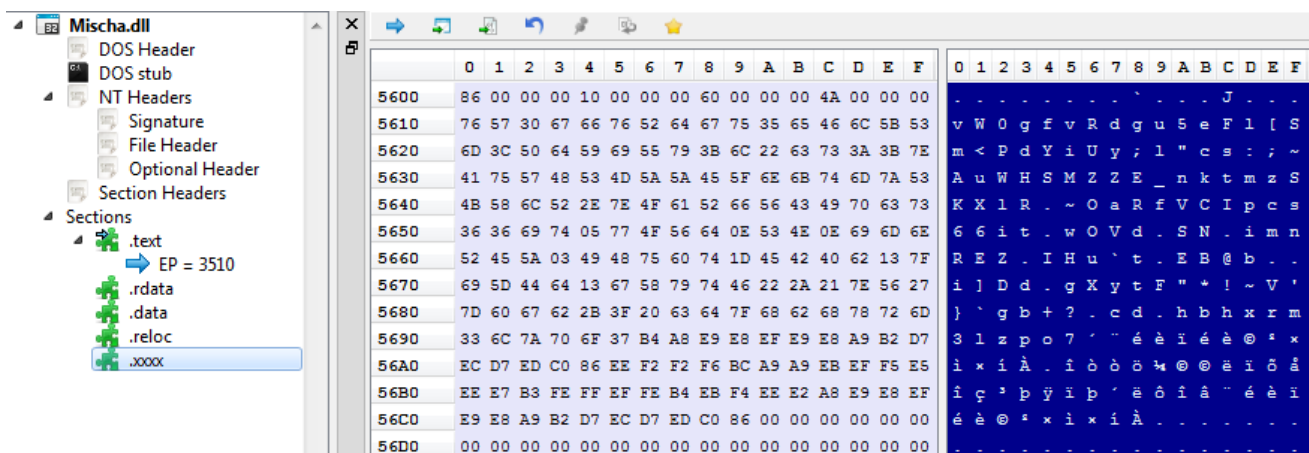
```

[Arg1 = 0006FD24
Setup.7116ECB0
ASCII "http://petya3jxfp2f7g3i.onion/"
ASCII "http://petya3sen7dyko2n.onion/"
ASCII "http://mischapuk6hyrn72.onion/"
ASCII "http://misha5xyix2mrhd.onion/"
Setup.7116F9D8
Setup.7116F9F8
Setup.7116FA18
Setup.7116FA38
Setup.7116F9D8
Setup.7116F9F8
Setup.7116FA18
Setup.7116FA38

```

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:





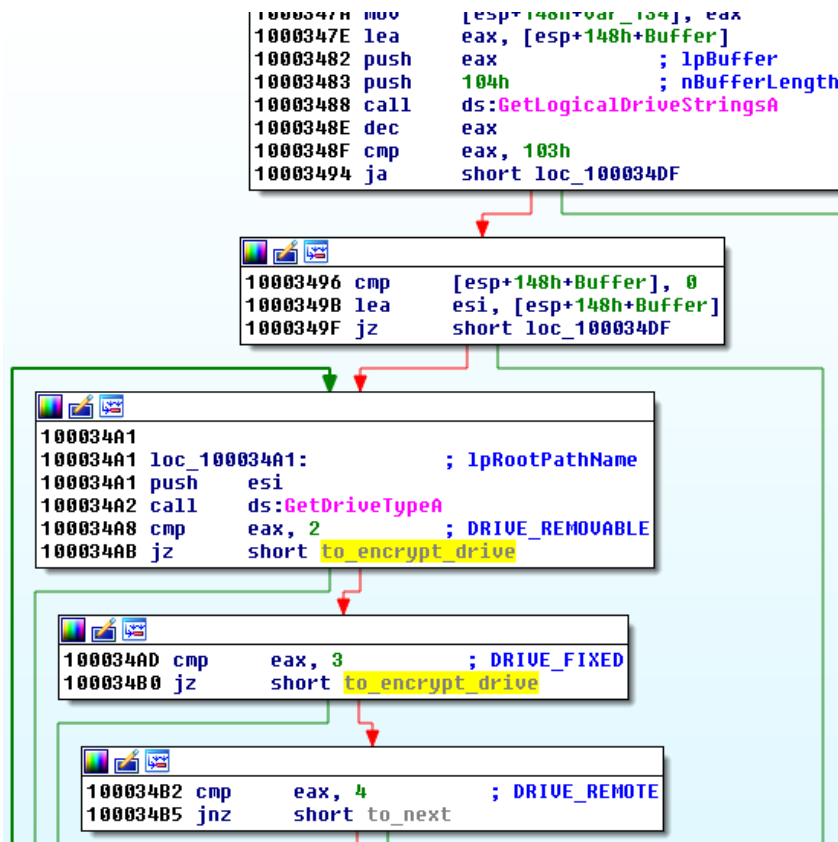
Offset	Name	Value	Meaning	
4CA0	Characteristics	0		
4CA4	TimeStamp	5730A00E		
4CA8	MajorVersion	0		
4CAA	MinorVersion	0		
4CAC	Name	64D2	Mischa.dll	
4CB0	Base	1		
4CB4	NumberOfFunctions	1		
4CB8	NumberOfNames	1		
4CBC	AddressOfFunctions	64C8		
4CC0	AddressOfNames	64CC		
4CC4	AddressOfNameOrdinals	64D0		
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 pdf jpg jpeg frm wdb ldf myi vmx xml xsl wps cmf vbs accdb ini cdr svg conf cfg config wb2 msg azw azw1 azw3 azw4 lit apnx mobi p12 p7b p7c pfx pem cer key der mdb htm html class java asp aspx cgi cpp php jsp bak dat pst eml xps sql lite sql jar wpd crt csv prf cnf indd number pages lnk dcu pas dfm directory pbk yml dtd rll lib cert cat inf mui props idl result localstorage ost default json sqlite log bat ico dll exe x3f srw pef raf orf nrw nef mrw mef kdc dcr crw eip fff iiq k25 crwl bay sr2 ari srf arw cr2 raw rwl rw2 r3d 3fr eps pdd dng dxf dwg psd png jpe bmp gif tiff gfx jge tga jfif emf 3dm 3ds max obj a2c dds pspimage yuv 3g2 3gp asf asx mpg mpeg avi mov flv wma wmv ogg swf ptx ape aif wav ram m3u movie mp1 mp2 mp3 mp4 mp4v mpa mpe mpv2 rpf vlc m4a aac aa3 amr mkv dvd mts vob 3ga m4v srt aepx camproj dash zip rar gzip vmdk mdf iso bin cue dbf erf dmg toast vcd ccd disc nrg nri cdi

## 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
0FB62498	·	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]
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]
0F663713  XOR  CL,BYTE PTR DS:[EAX]
0F663715  ADD  EAX,EBX
0F663717  INC  EDX
0F663718  MOV  BYTE PTR SS:[EBP+EAX-0xDE0],CL
0F66371F  CMP  EDX,0x10
0F663722  JLE  SHORT Misha.0F66370B

```

content\_buffer  
content[i]  
random\_buf[i]  
output[i] = content\_buffer[i] ^ random\_buf[i]  
i < 16?

CL=C4 ('-')  
SS:[0031DED0]=20 (' ')

Address	Hex dump	ASCII
0018F630	B3 F2 BF 8A 31 CA C2 EA 4E 5E 22 87 F6 DA DC 68	110122N^c^nmh
0018F640	C8 88 3D 00 28 05 00 00 28 05 00 00 00 00 00 00	8=.(#..(#.....

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 . . . . . >
0FB6370E . . . . . LEA EAX, [LOCAL.20]
0FB63710 . . . . . ADD EAX, EDX
0FB63713 . . . . . MOV CL, BYTE PTR DS:[EDI+EAX] next_character
0FB63715 . . . . . XOR CL, BYTE PTR DS:[EAX]
0FB63717 . . . . . ADD EAX, EBX
0FB63718 . . . . . INC EDX
0FB6371F . . . . . MOV BYTE PTR SS:[EBP+EAX-0x0E0], CL out_buffer
0FB63722 . . . . . CMP EDX, 0x10
0FB63724 . . . . . JLT SHORT Mischa.0FB6370B
0FB63726 . . . . . ADD ESI, EBX
0FB63728 . . . . . LEA ECX, [LOCAL.182]
0FB6372C . . . . . PUSH ESI
0FB6372E . . . . . MOV EDX, ESI
0FB63730 . . . . . 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..+p0GLtq%C+--E#
001BE9A0	0F D7 6A E0 E7 AB 89 0F A7 C0 51 14 9E EE B5 B9	*iJ0&2&*2'Q1x%AJ
001BE9B0	4F FF 40 24 60 D4 2C 92 35 8E CD 01 2B 94 26 F5	o'060.'520
001BE9C0	DE F5 A9 0B AA 80 25 A8 C5 EA 69 20 C3 54 F6 28	03e2 C%E+L..FI+s

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):

```

D Dump - Mischa:..xxxx 0FB6B000..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 68 46 77 32 68 38 75 61 34 31 78 b4Qj0kFw2h8ua41x
0FB6B030 43 55 4A 52 44 59 4B 54 4E 57 68 69 58 47 52 75 CUJRDYKTNWk iXGRu
0FB6B040 6F 7A 4C 4C 32 64 57 77 46 74 46 40 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 qqi1xx2NY1nhj4Y
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 j0kF.http://misc
0FB6B0B0 68 61 35 78 79 69 78 32 6D 72 68 64 2E 6F 6E 69 ha5xy ix2mrd.oni
0FB6B0C0 6F 6E 2F 34 51 6A 51 6B 46 00 00 00 00 00 00 00 on/4Qj0kF.....

```

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+0verlapped]
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.

```

0F6638C7 | .: LEA EAX, [LOCAL_504]
0F6638C8 | .: PUSH EAX
0F6638C9 | .: PUSH ESI
0F6638CA | .: CALL DWORD PTR DS:[&KERNEL32.MoveFileA]
0F6638CB | .: MOV EDX, EDI
0F6638CC | .: MOV ECX, ESI

```

```

NewName = "C:\pin\extras\pinadx-vsplugin\readme.txt.4QjQ"
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-ç.»ı1B
000007F0 62 8D CF C1 5F 77 9A 13 E4 2F 45 A8 0B 56 21 00 bTĐÁ wš.ä/E".V! .
00000800 28 05 00 00 B3 F2 BF 8A 31 CA C2 EA 4E 5E 22 87 (...)_ňžŠ1EĀēN^"+
00000810 F6 DA DC 68 62 34 51 6A 51 6B 46 77 32 68 38 75 öÜŪhb4QjQkFw2h8u
00000820 61 34 31 78 43 55 4A 52 44 59 4B 54 4E 57 6B 69 a41xCUJRdYKTNWki
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 7QZXgqilxxZNY1nh
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 «@ć1÷3`.j-ť.»ť1B
000007F0 62 8D CF C1 5F 77 9A 13 E4 2F 45 A8 0B 56 21 00 bŤĎÁ_wš.ă/E".V!
00000800 28 05 00 00 B3 F2 BF 8A 31 CA C2 EA 4E 5E 22 87 (..._ňžŠ1EĀēN^"+
00000810 F6 DA DC 68 62 34 51 6A 51 6B 46 77 32 68 38 75 öÜŮh_b4QjQkFw2h8u
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 «@ć1÷3`.j-ť.»ť1B
000007F0 62 8D CF C1 5F 77 9A 13 E4 2F 45 A8 0B 56 21 00 bŤĎÁ_wš.ă/E".V!
00000800 28 05 00 00 B3 F2 BF 8A 31 CA C2 EA 4E 5E 22 87 (..._ňžŠ1EĀēN^"+
00000810 F6 DA DC 68 62 34 51 6A 51 6B 46 77 32 68 38 75 öÜŮh_b4QjQkFw2h8u
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 it's 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/>