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.

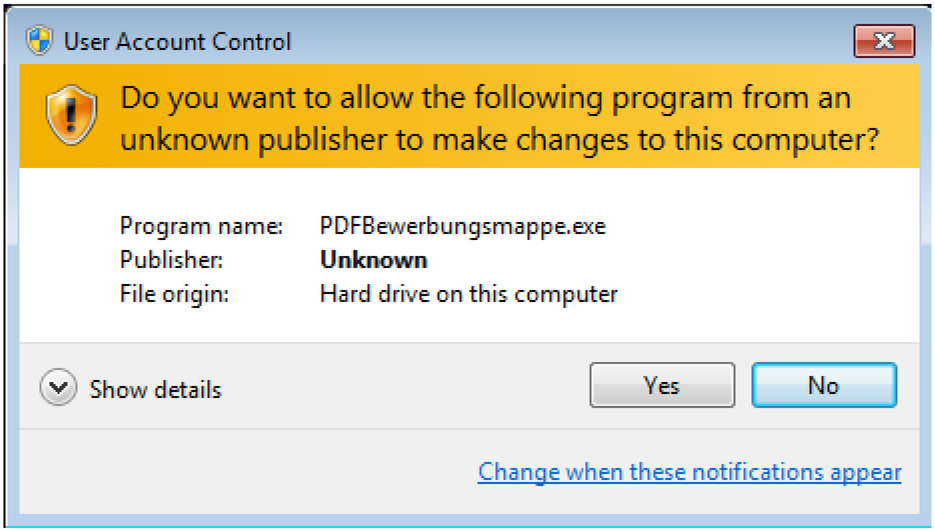
Analyzed samples

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

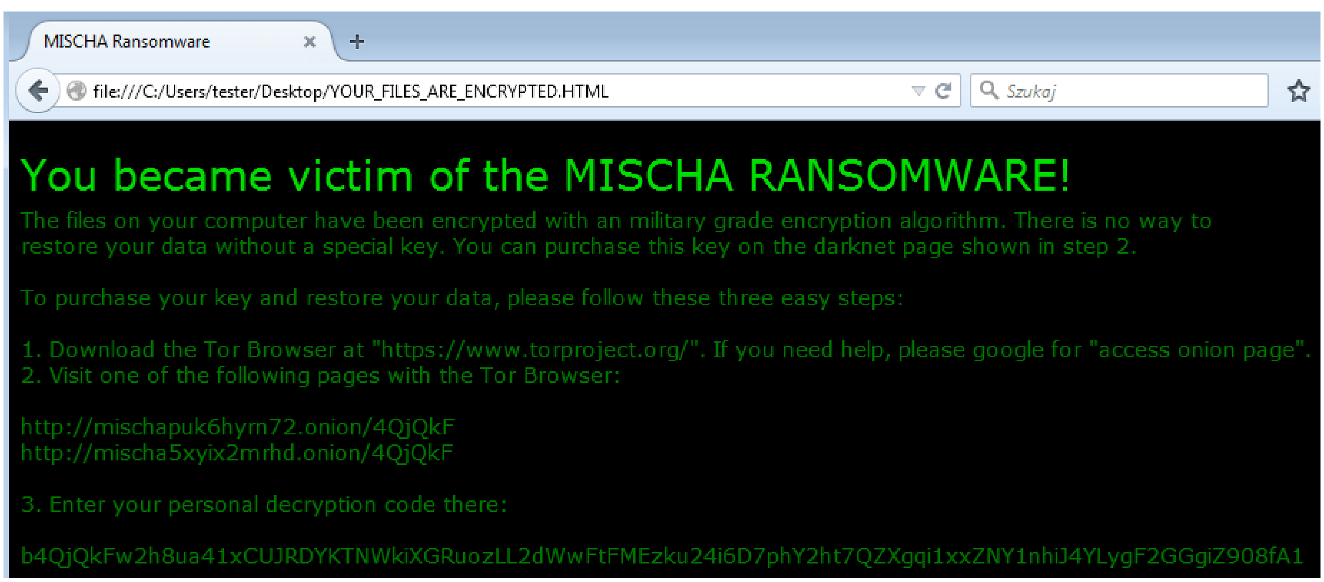
As mentioned in the previous part of the article, both malicious payloads are dropped by the same dropper. The choice of which one will be used for the attack is made based on the privileges with which the sample is deployed. First, there is a request asking a user to elevate the application's privileges:



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.

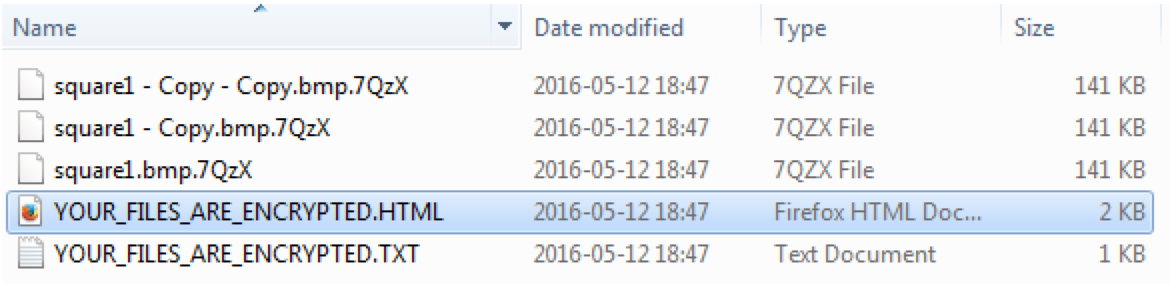
This payload works just like any other ransomware – encrypting files one by one and dropping a ransom note: *YOUR_FILES_ARE_ENCRYPTED.HTML* (identical name was used before by another ransomware: Chimera). The layout is analogical to the one used by Petya.



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

Encryption process

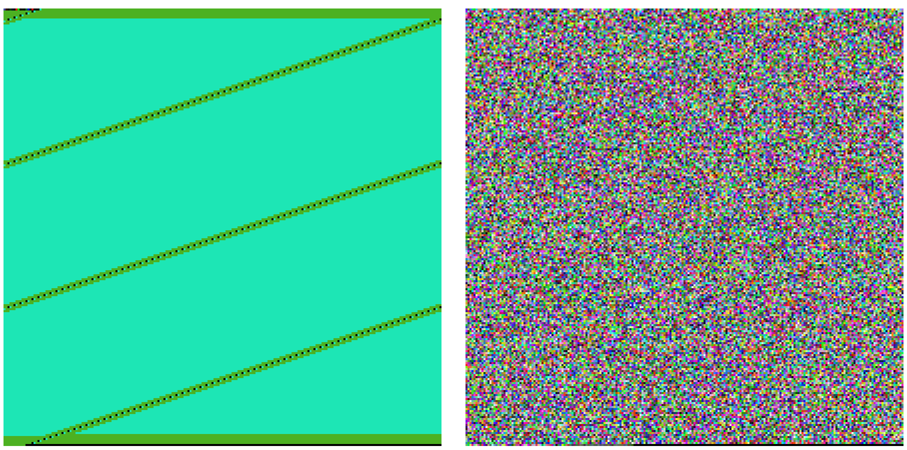
Mischa does not need to download a key from the CnC server – data can be encrypted offline as well. Extensions given to the encrypted files are random, generated at runtime (they are same like a part of the tor address):



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

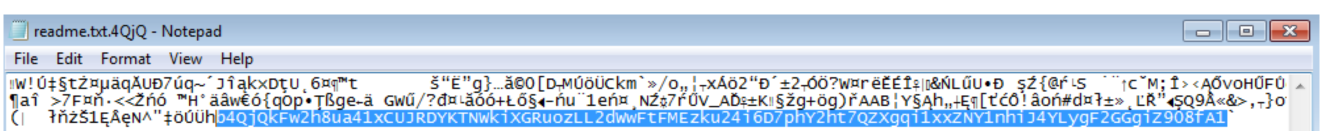
Entropy of encrypted samples is high and no patterns are visible. See below a visualization of bytes.

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

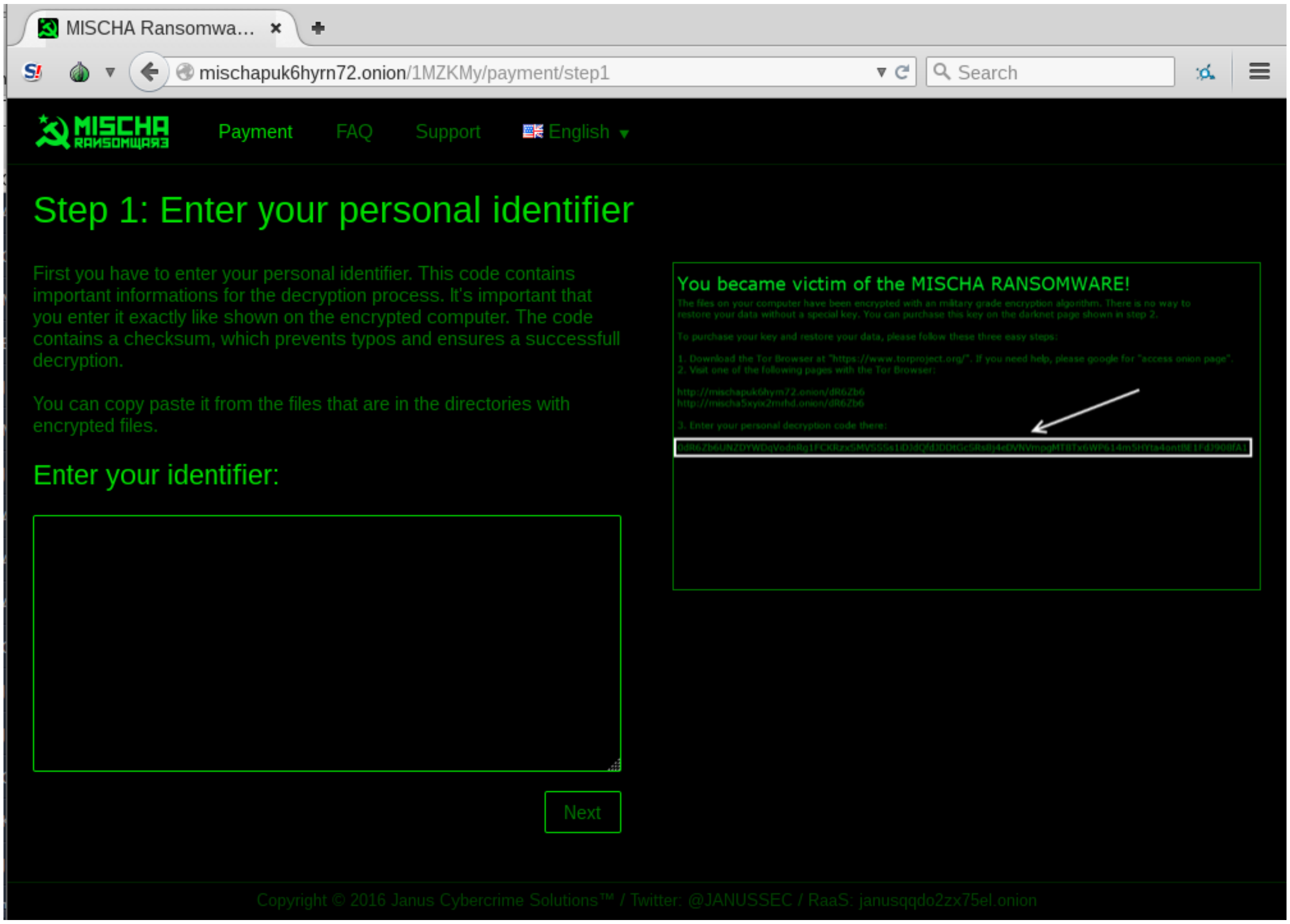


The same input does not produce the same output – that suggest that every file is encrypted with an individual key (or initialization vector).

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



Page for the victim:



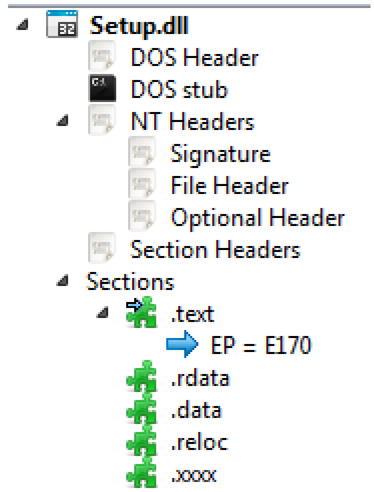
Inside

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).

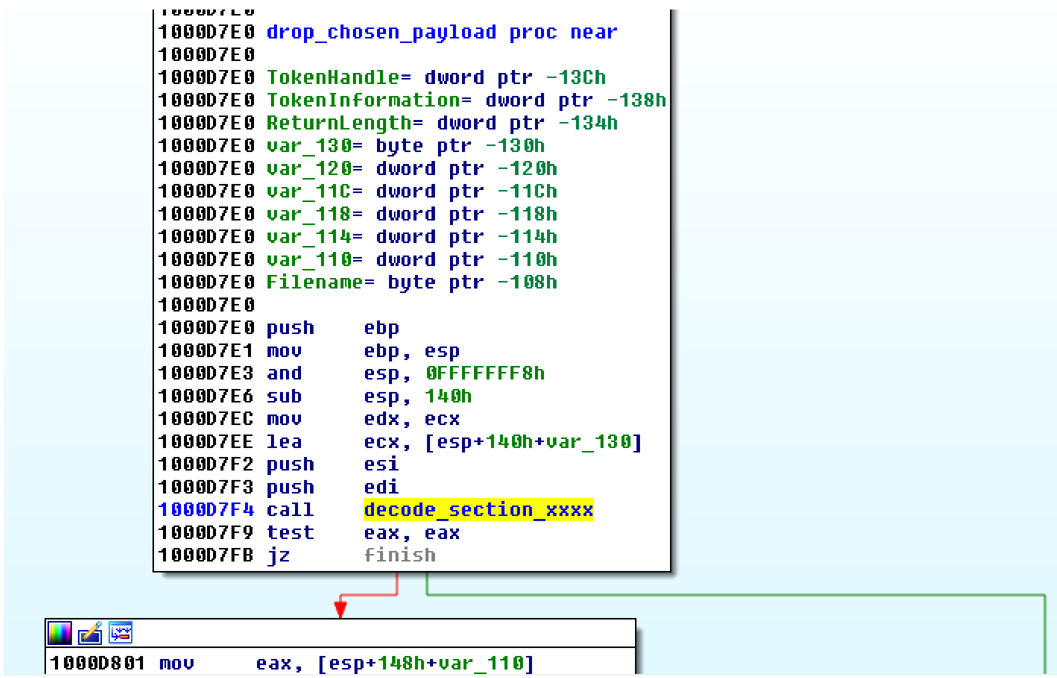
Setup.dll

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.

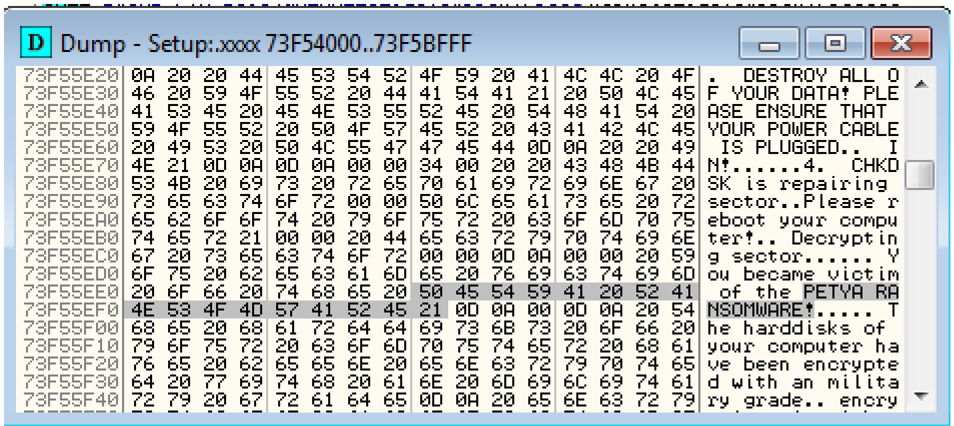
Similarly to the dropper of the previous Petya, it comes with a section *.xxxx:*



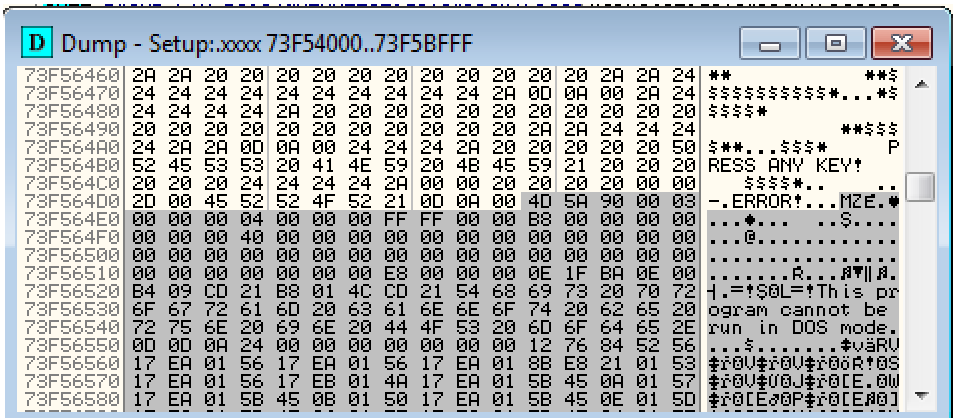
This section is very important, because it contains both payloads – Petya and Mischa (encrypted by simple XOR based algorithm). At the beginning of the execution they are being decrypted:



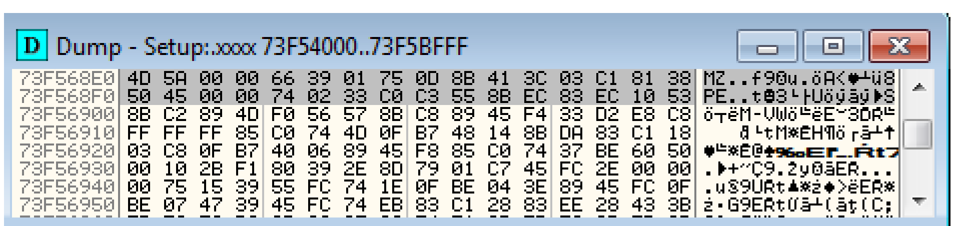
We can see a stub similar to the previous Petya:



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

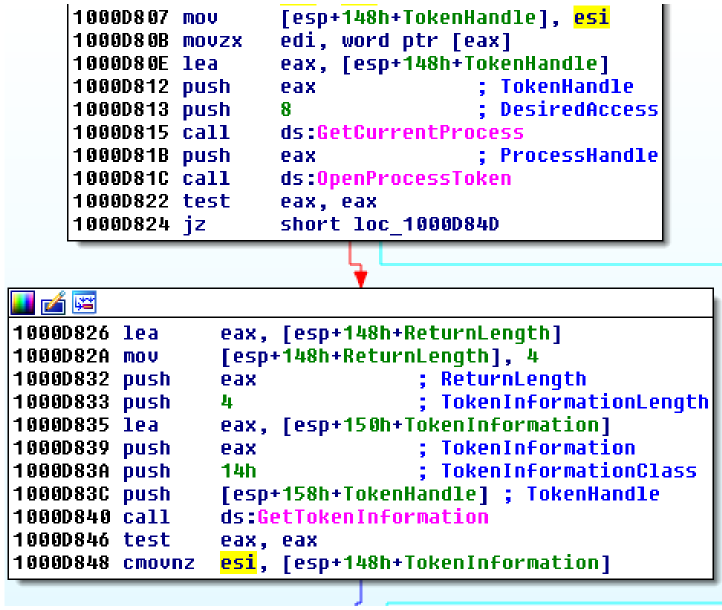


Authors tried to deceive tools for automated dumping of PE files from the memory, and provided fake “MZ”…”PE” patterns:



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:

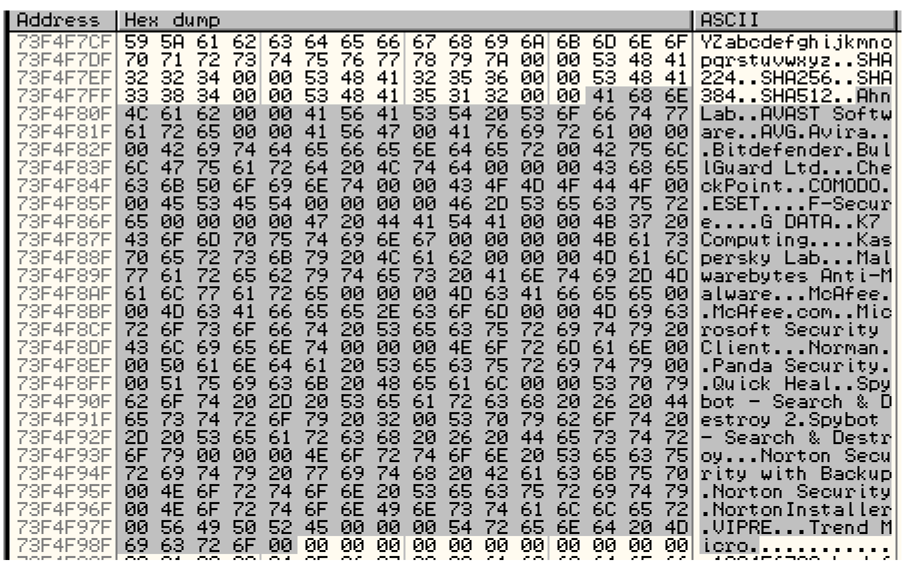


```
1000D807 mov     [esp+148h+TokenHandle], esi
1000D80B movzx   edi, word ptr [eax]
1000D80E lea     eax, [esp+148h+TokenHandle]
1000D812 push    eax                ; TokenHandle
1000D813 push    8                  ; DesiredAccess
1000D815 call    ds:GetCurrentProcess
1000D81B push    eax                ; 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]
```

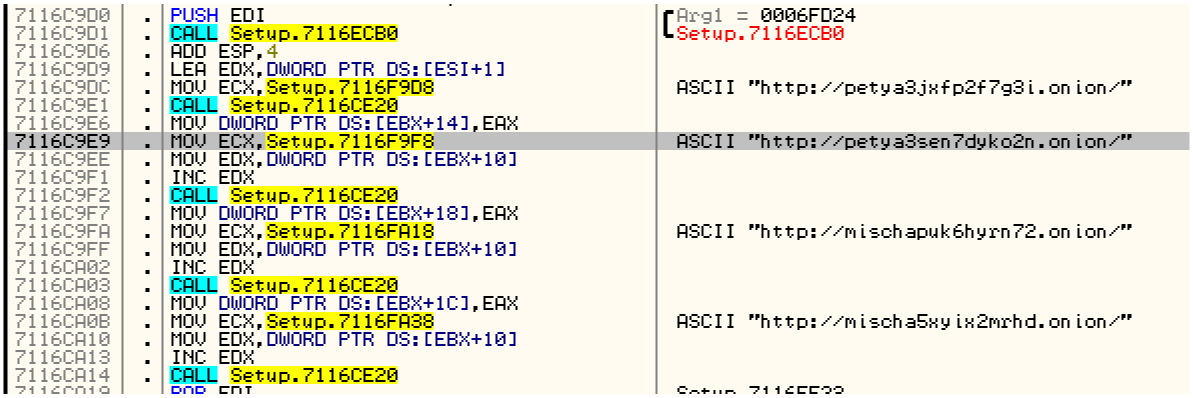
Choosing between **Petya** and **Mischa** is done in few steps. First, the token check is used to get information if the application is deployed with administrative rights. If it is not, then the it tries to run it's new copy with higher privileges (using *runas* command). If this attempt failed, Mischa is dropped (otherwise – Petya).

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



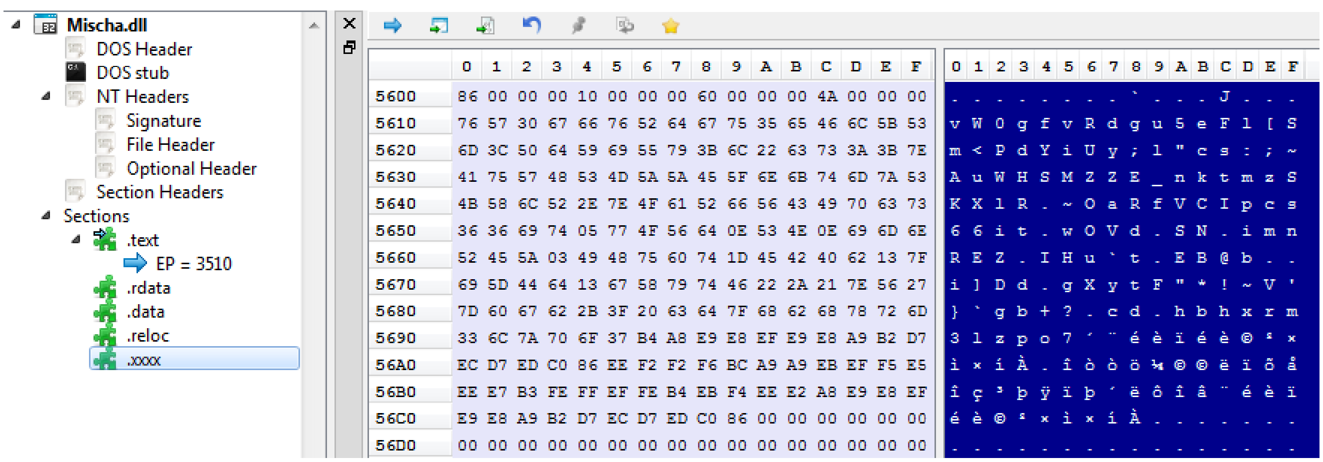
```
Address   Hex dump                                         ASCII
73F4F7CF  59 5A 61 62 63 64 65 66 67 68 69 6A 6B 6D 6E 6F  YZabcdefghijkmno
73F4F7DF  70 71 72 73 74 75 76 77 78 79 7A 00 00 53 48 41  pqrstuvwxyz..SHA
73F4F7EF  32 32 34 00 00 53 48 41 32 35 36 00 00 53 48 41  224..SHA256..SHA
73F4F7FF  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 6B 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 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  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 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:



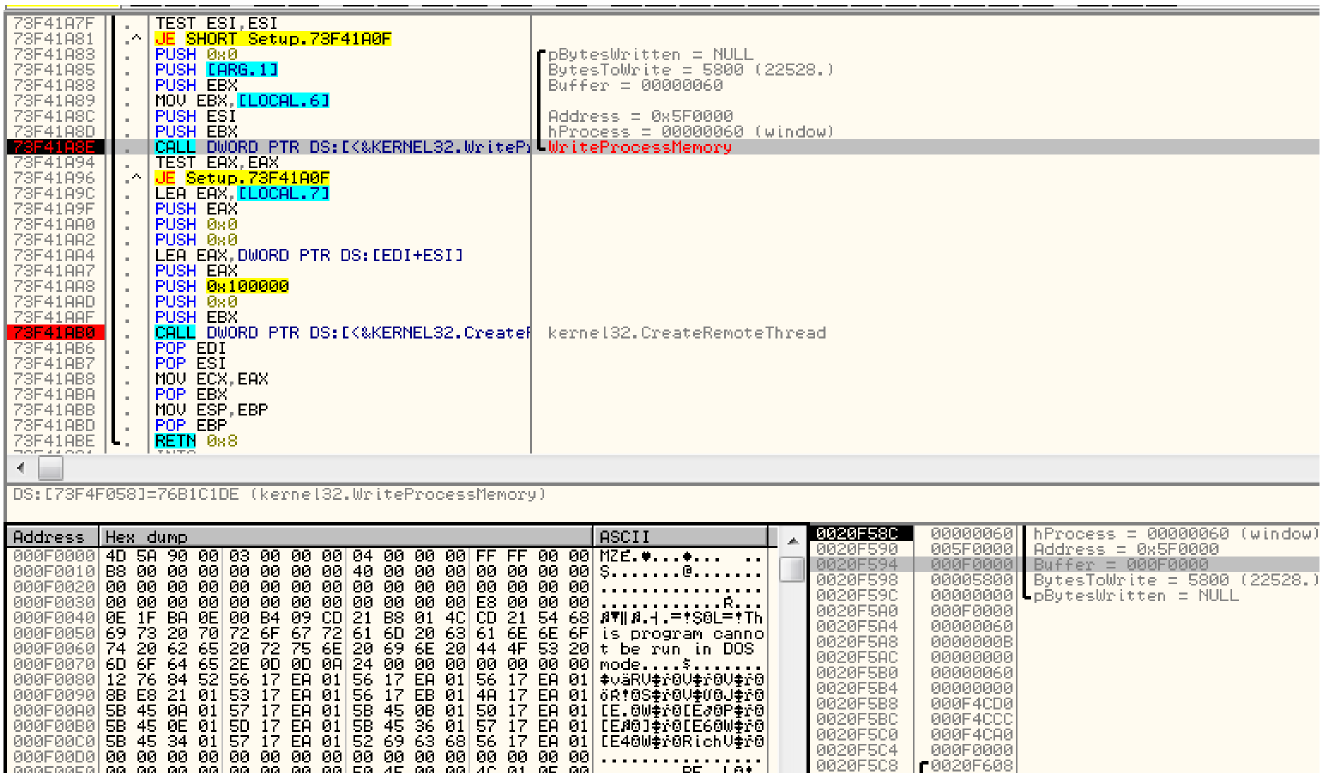
Inside the dropper, Mischa's DLL (similarly to Petya's stub) is being filled with additional, unique data. Similarly to Petya, Mischa gets a random key that will be used in further encryption process. This key is encrypted using ECC and transformed into a victim ID. Then, part of this victim ID becomes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

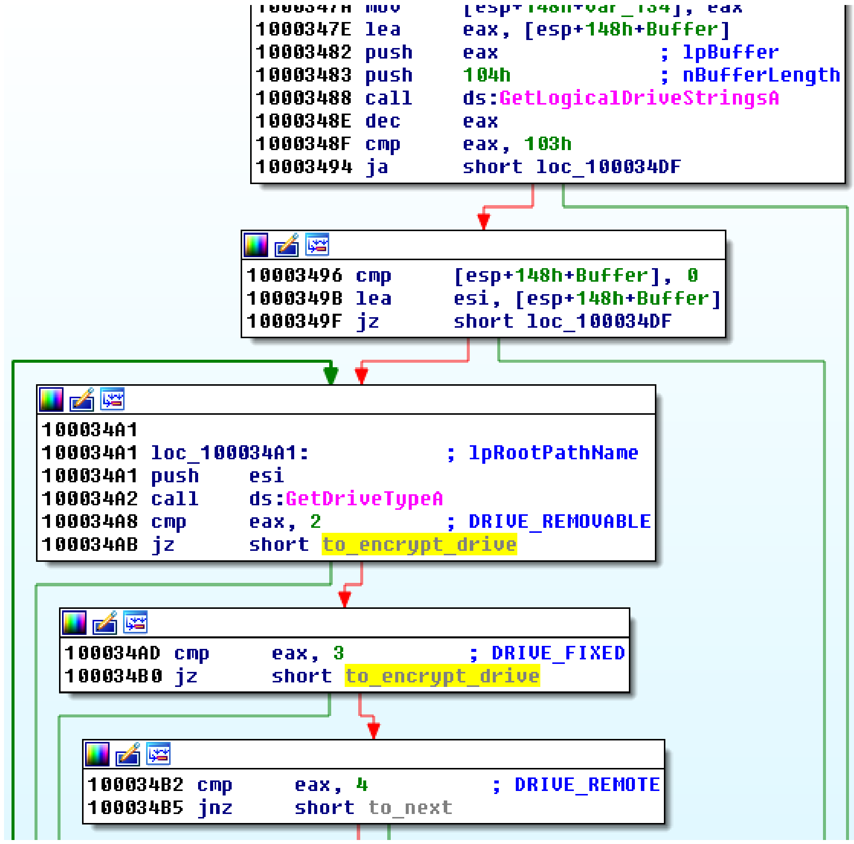
Details

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 it's 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 xlsm 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 sqllite 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.

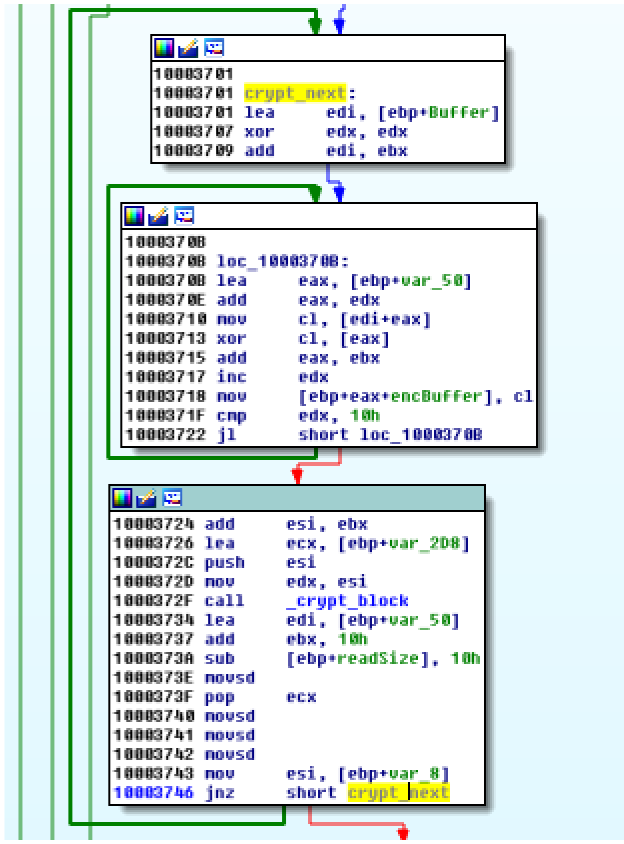
```
0FB62466  .  MOV DWORD PTR DS:[EDI],EBX
0FB62468  .  CALL DWORD PTR DS:[<&ADVAPI32.CryptAcquireContextA>]  advapi32.CryptAcquireContextA
0FB6246E  .  TEST EAX,EAX
0FB62470  .v JNZ SHORT Mischa.0FB62477
0FB62472  .  PUSH -0x3C
0FB62474  .  POP EAX                                               003B89E0
0FB62475  .v 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  .v JNZ SHORT Mischa.0FB62491
0FB6248C  .  PUSH -0x3C
0FB6248E  .  POP EAX                                               003B89E0
0FB6248F  .v 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:

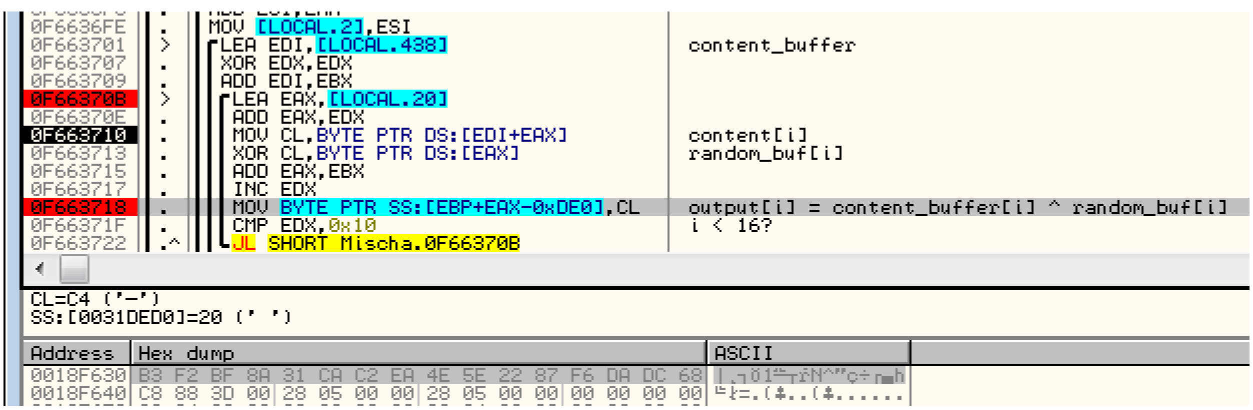
and then, encrypted by the locally implemented algorithm:



Encryption process is divided in 2 phases.

Phase 1:

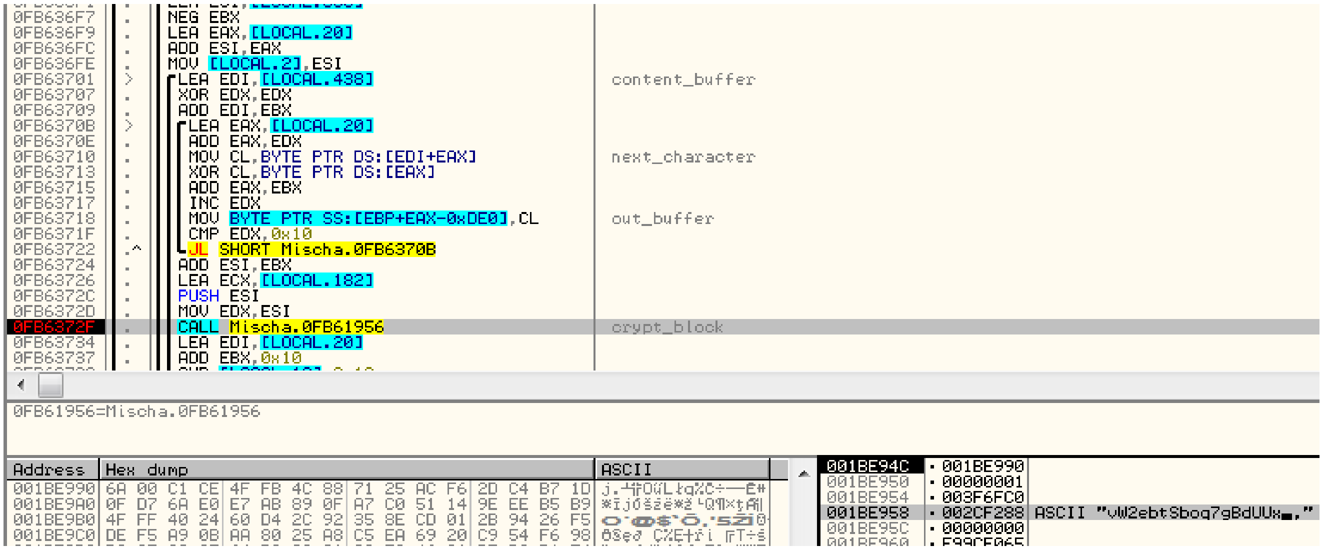
Each 16 bytes of the read chunk is preprocessed by XOR with a 16 byte long buffer:



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)

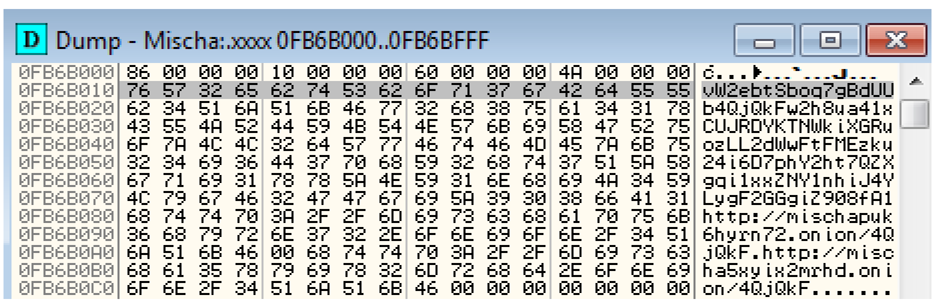
Phase 2:

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



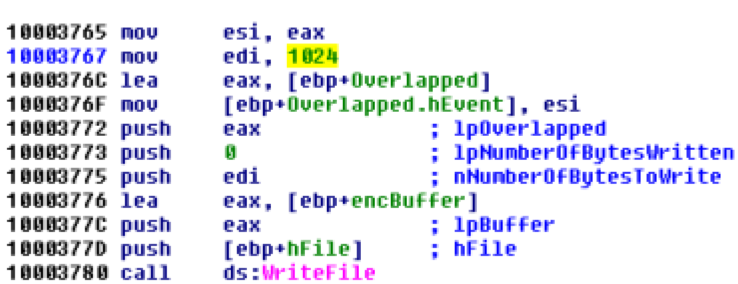
This block cipher processes 16 bytes of the input and gives as a result 16 bytes of encrypted output. Encryption involves a 16 byte long key (that was hardcoded in the appended section) – in a given example it is ***vW2ebtSboq7gBdUU***.

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):

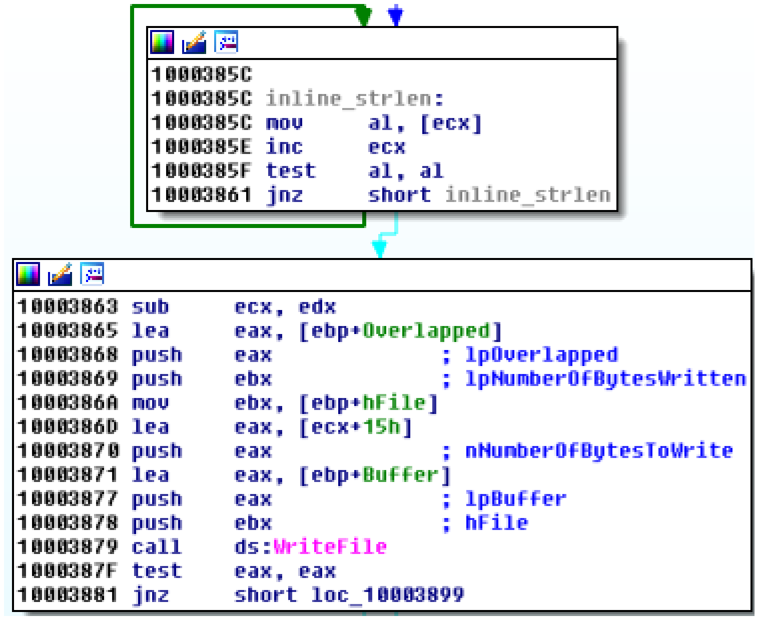


As long as Mischa is running, this key is in memory in open text. But once it finishes, this data is being destroyed and only the encrypted form of the key is left – user receives it in the ransom note. (It is somehow similar to the logic of Petya).

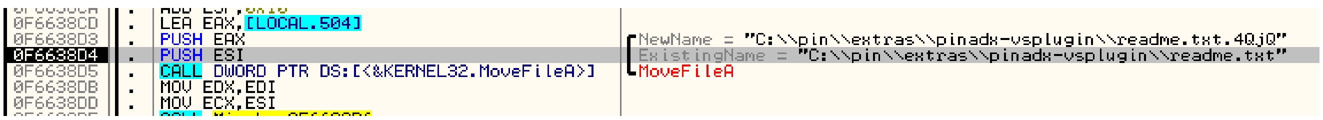
Encrypted chunks are being written into the file one by one:



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

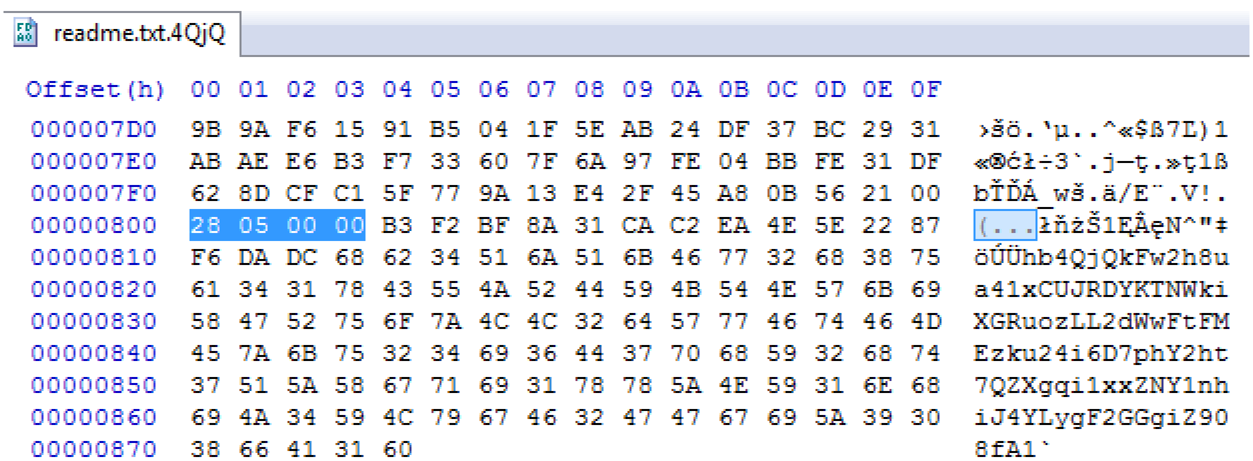


Then, file is moved under the new name.

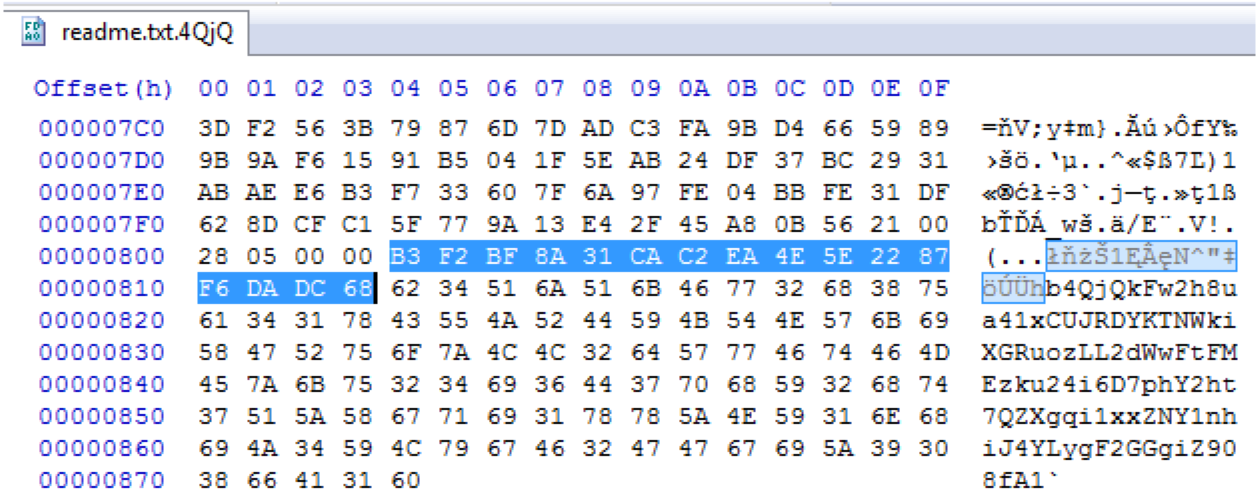


Let’s have a look at the appended data and it’s role in decoding the file. At the end of the encrypted file we can find:

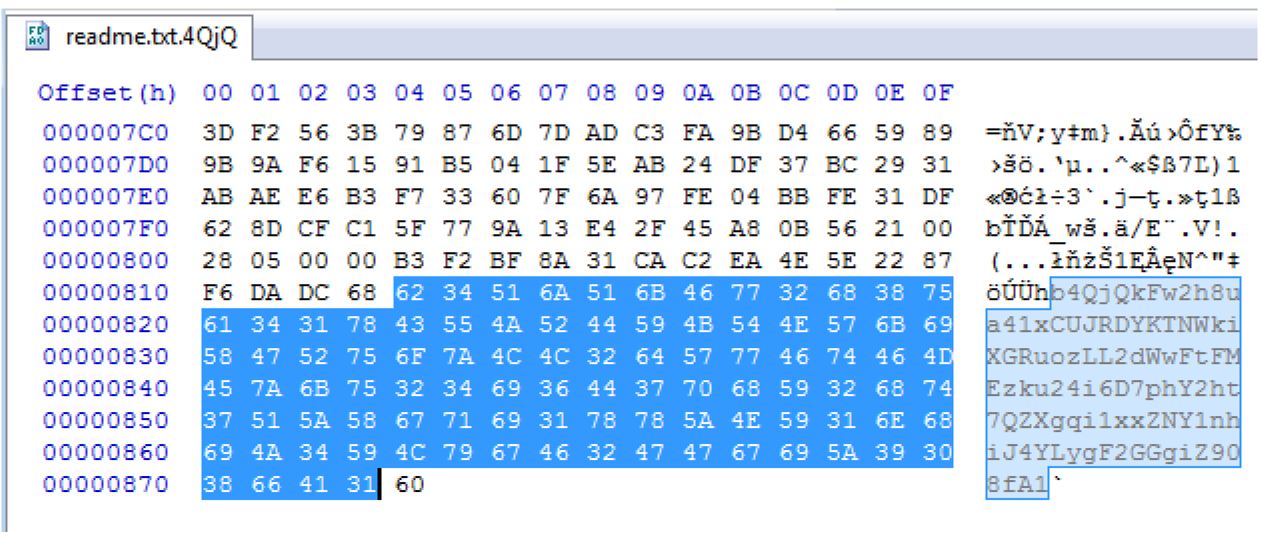
1. Length of the original file (0x528 -> 1320)



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



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***



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/