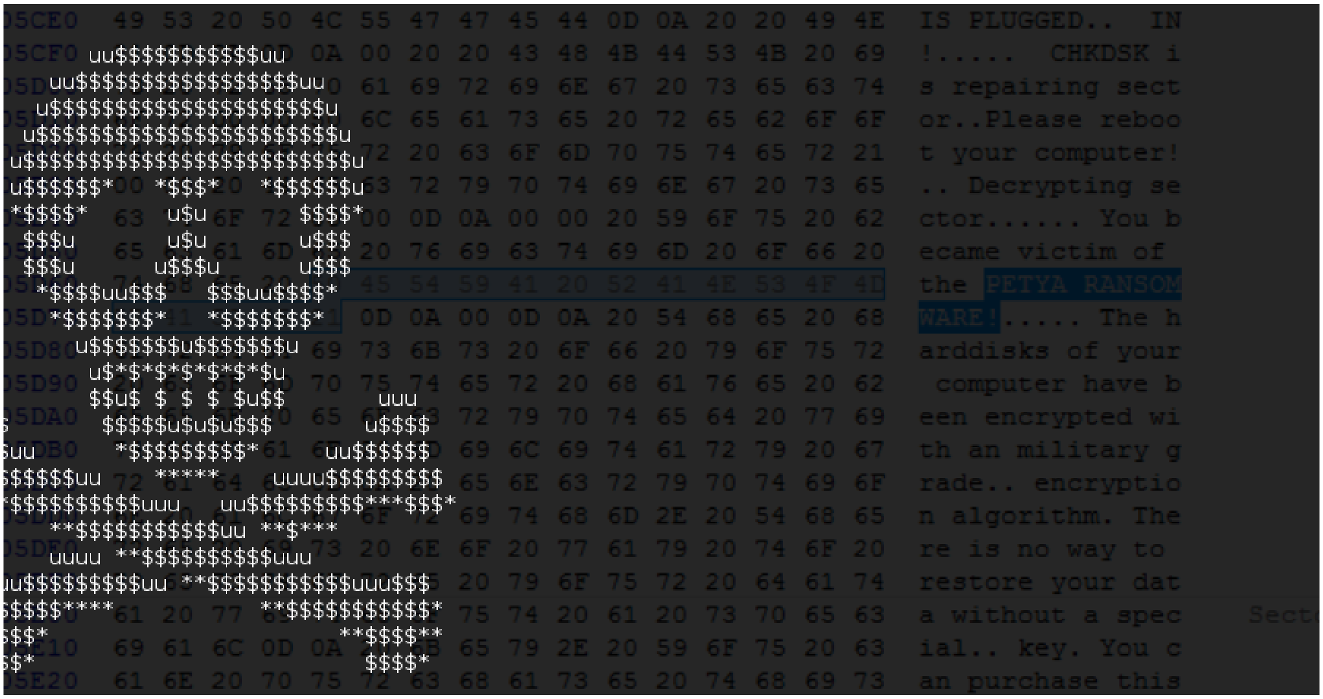
Petya and Mischa – Ransomware Duet (Part 1)

blog.malwarebytes.com/threat-analysis/2016/05/petya-and-mischa-ransomware-duet-p1/

hasherezade

May 19, 2016



```
     uu$$$$$$$$$$$uu
   uu$$$$$$$$$$$$$$$$$uu
  u$$$$$$$$$$$$$$$$$$$$$u
 u$$$$$$$$$$$$$$$$$$$$$$$u
u$$$$$$$$$$$$$$$$$$$$$$$$$u
u$$$$$$*   *$$$*   *$$$$$$u
*$$$$*      u$u       $$$$*
 $$$u       u$u       u$$$
 $$$u      u$$$u      u$$$
  *$$$$uu$$$   $$$uu$$$$*
   *$$$$$$$*   *$$$$$$$*
     u$$$$$$$u$$$$$$$u
      u$*$*$*$*$*$*$u
      $$u$ $ $ $ $u$$       uuu
$      $$$$$u$u$u$$$       u$$$$
$uu     *$$$$$$$$$*     uu$$$$$$
$$$$$$uu    *****    uuuu$$$$$$$$$
*$$$$$$$$$$uuu   uu$$$$$$$$$***$$$*
   **$$$$$$$$$$$uu **$***
   uuuu **$$$$$$$$$$uuu
uu$$$$$$$$$uu **$$$$$$$$$$$uuu$$$
$$$$$****          **$$$$$$$$$$$*
$$$*                      **$$$$**
$$*                         $$$$*
```

```
IS PLUGGED..  IN
!.....  CHKDSK i
s repairing sect
or..Please reboo
t your computer!
.. Decrypting se
ctor...... You b
ecame victim of
the PETYA RANSOM
WARE!..... The h
arddisks of your
 computer have b
een encrypted wi
th an military g
rade.. encryptio
n algorithm. The
re is no way to
restore your dat
a without a spec
ial.. key. You c
an purchase this
```

After being defeated about a month ago, 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.

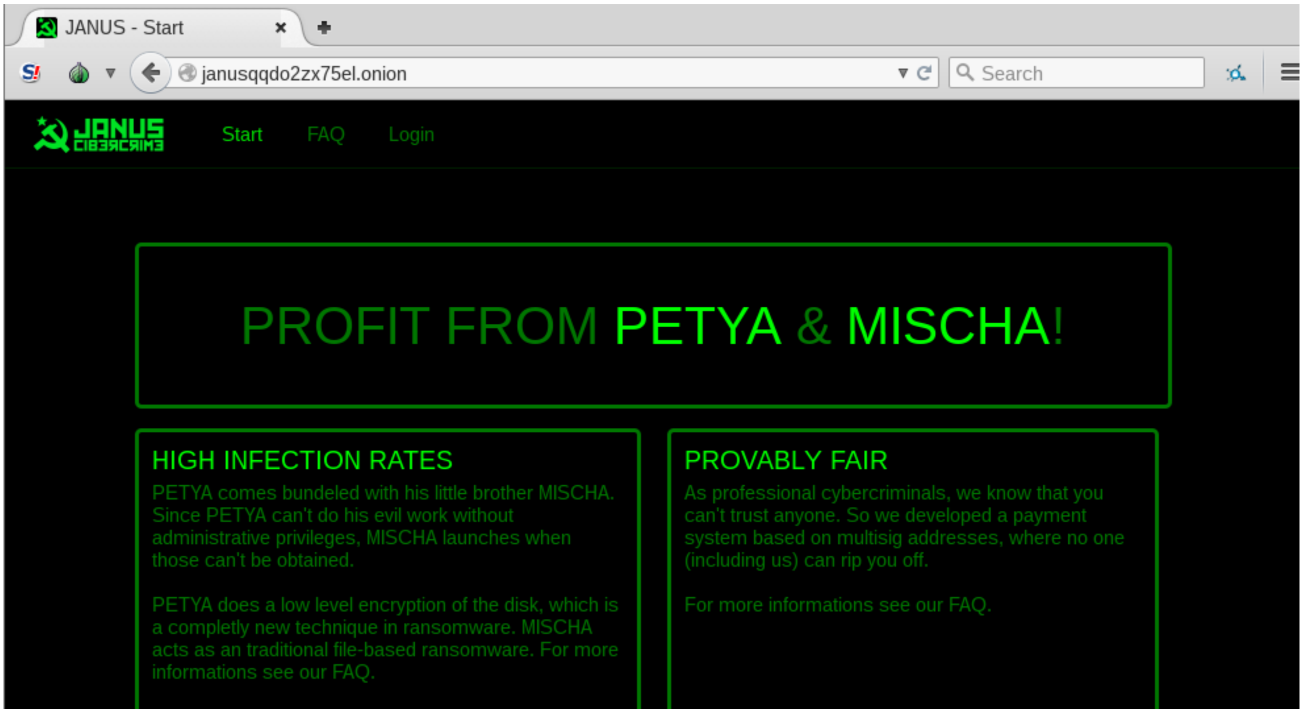
They deploy attacks on different layers of the system and are used as alternatives. That’s why, we decided to dedicate more than one post to this phenomenon. Welcome to part one! **The main focus of this analysis is Petya (the Green version).**

The second part (about Mischa) you can read about it here.

UPDATE: Improved version of Green Petya is out. More details given in the new article.

Let’s start with some background information.

This time authors also deployed a page with information for potential clients of their Ransomware-As-A-Service:



PROFIT FROM PETYA & MISCHA!

HIGH INFECTION RATES

PETYA comes bundeled with his little brother MISCHA. Since PETYA can't do his evil work without administrative privileges, MISCHA launches when those can't be obtained.

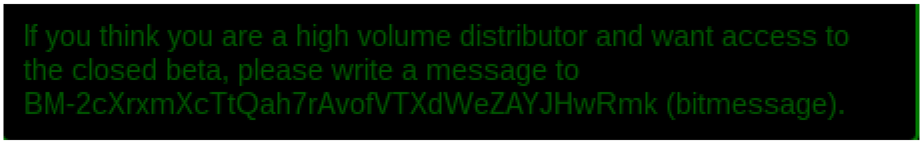
PETYA does a low level encryption of the disk, which is a completly new technique in ransomware. MISCHA acts as an traditional file-based ransomware. For more informations see our FAQ.

PROVABLY FAIR

As professional cybercriminals, we know that you can't trust anyone. So we developed a payment system based on multisig addresses, where no one (including us) can rip you off.

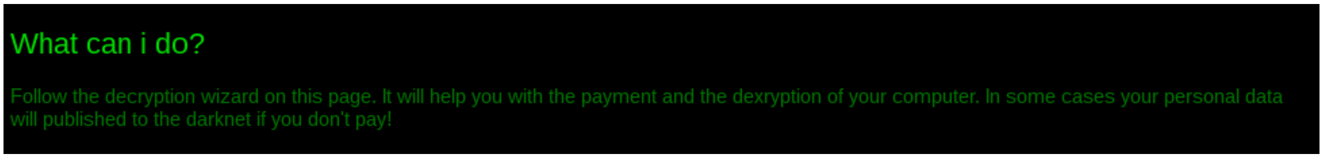
For more informations see our FAQ.

Just like in the case of Chimera, the authors use bitmessage for communication with the new recruits of the criminal cooperation:



If you think you are a high volume distributor and want access to the closed beta, please write a message to BM-2cXrxmXcTtQah7rAvofVTXdWeZAYJHwRmk (bitmessage).

And post doxing threats, also known from Chimera:



What can i do?

Follow the decryption wizard on this page. It will help you with the payment and the dexryption of your computer. In some cases your personal data will published to the darknet if you don't pay!

Analyzed samples

8a241cfcc23dc740e1fadc7f2df3965e – main executable

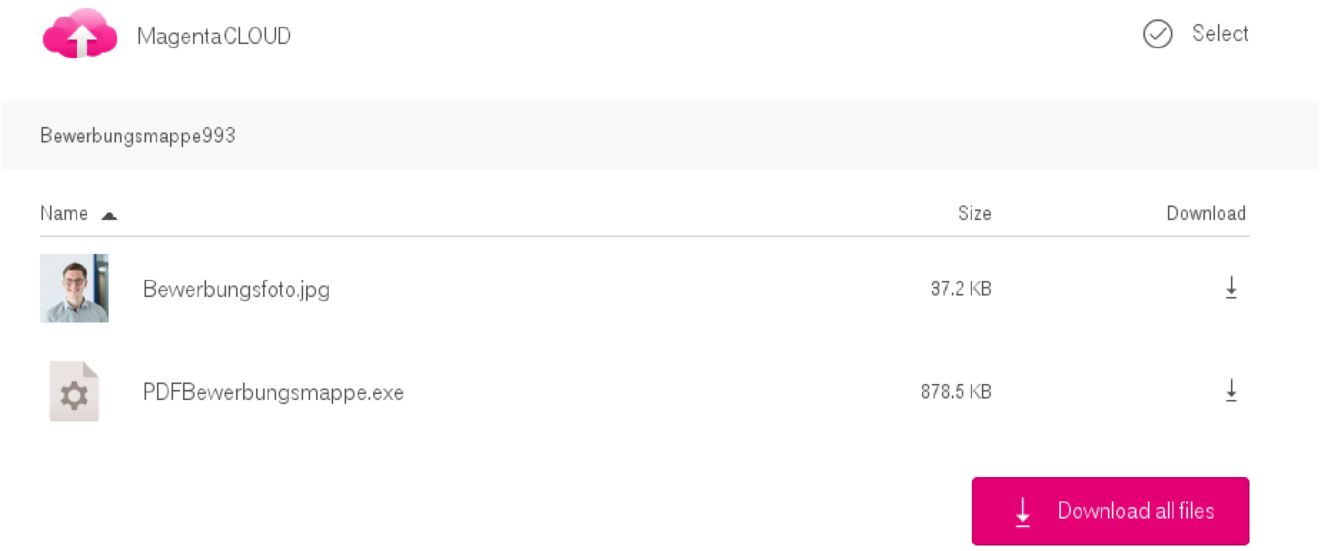
f7596666d8080922d786f5892dd70742– main executable (from a different campaign)

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

Just like the previous version, it is distributed via cloud storage and pretends to be a job application:



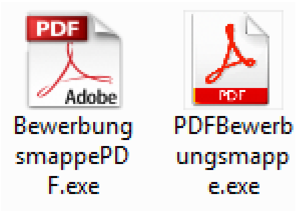
MagentaCLOUD ○ Select

Bewerbungsmappe993

Name ▲	Size	Download
Bewerbungsfoto.jpg	37.2 KB	↓
PDFBewerbungsmappe.exe	878.5 KB	↓

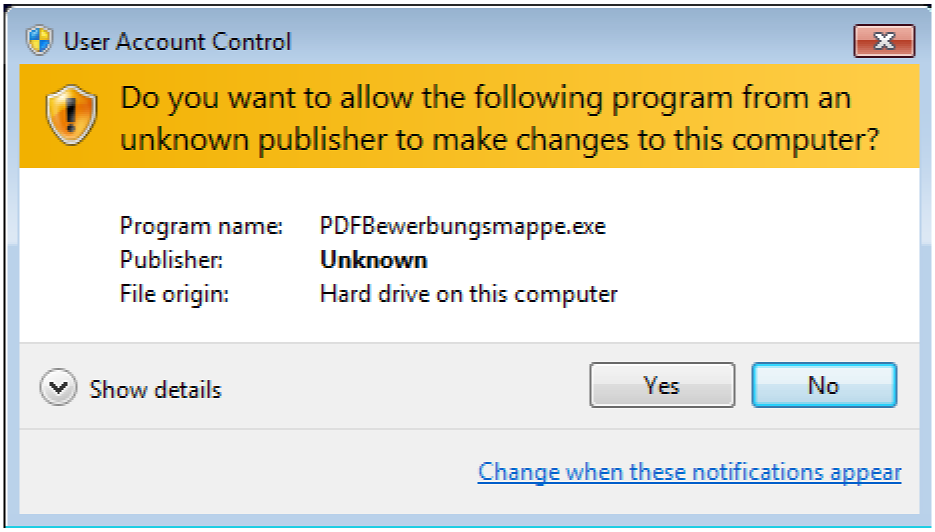
↓ Download all files

The executable is again packed with an icon of a PDF document:



After deploying it, it can drop one of the two payloads – Petya – that works similarly to the previously described version, or Mischa – that has features of a typical ransomware. The decision which payload to deploy is based on privileges with which the sample runs – that implies accessibility to write to the MBR. If writing there is not possible, authors decided not to miss the chance of infecting the system and deploy more typical userland attack with Mischa.

From the point of view of the user – your decision taken on UAC pop-up will result in deploying one out of the two payloads. If you choose “No” – you get Mischa. If you choose “Yes” – you get Petya.

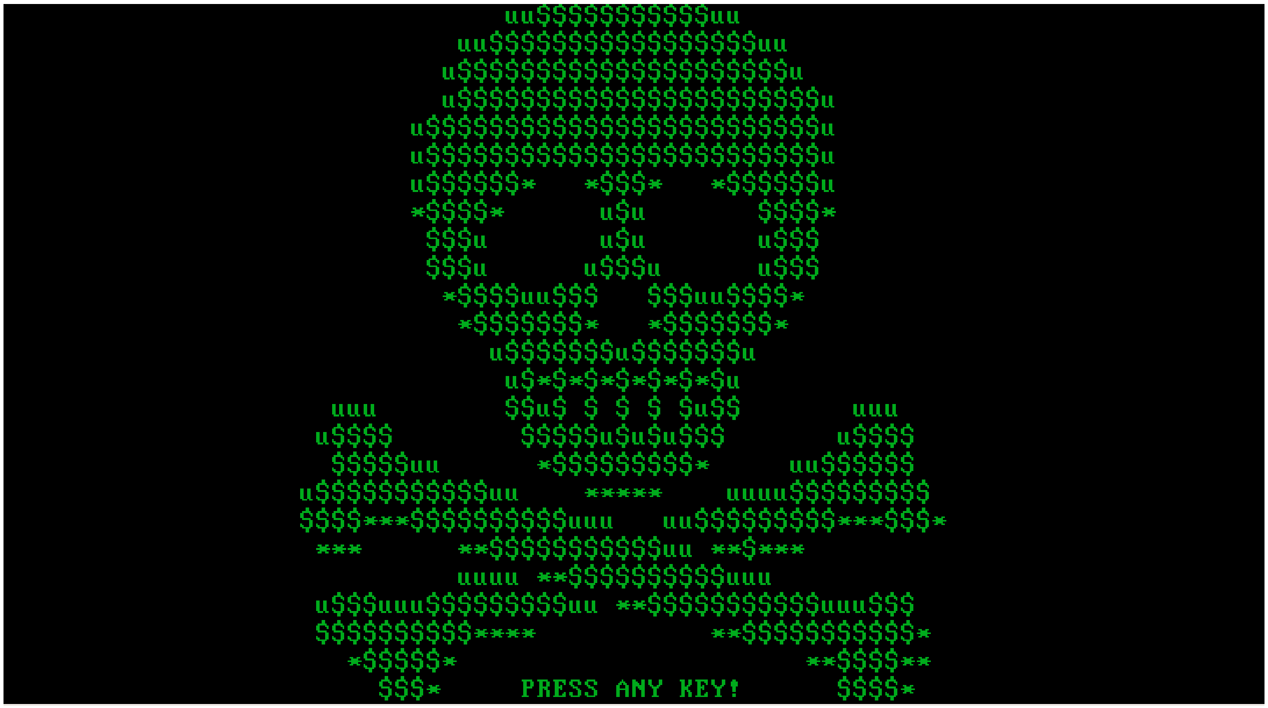

User Account Control

Do you want to allow the following program from an unknown publisher to make changes to this computer?

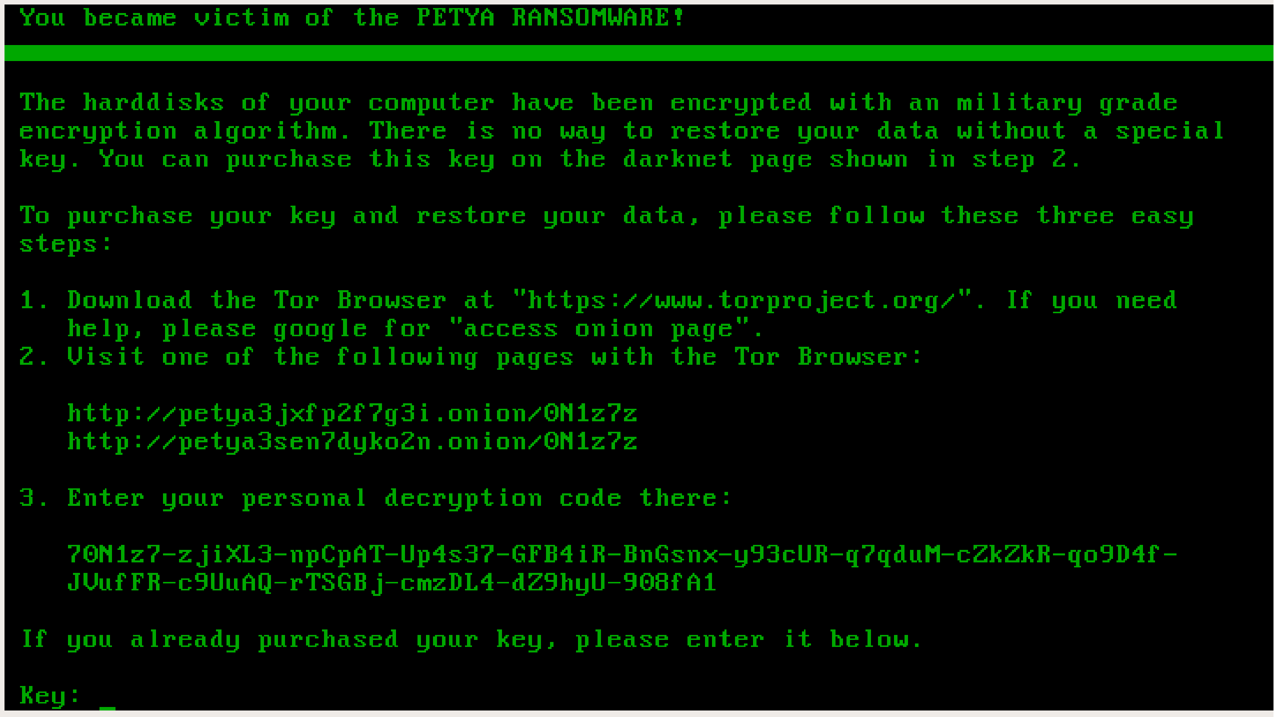
Program name: PDFBewerbungsmappe.exe
Publisher: **Unknown**
File origin: Hard drive on this computer

Petya

The infection process looks exactly the same as in the previous verison of Petya. User Account Control notification pops up, and in case the user accepts it, Petya installs itself in the MBR and crashes the system. Stage 2 of the infection also looks almost the same. First it runs the fake CHKDSK, that in reality encrypts the disc. Then, the user can see the ASCII art with the blinking skull and the ransom note. Only the color theme changed – instead of red, we have a black background with green text:



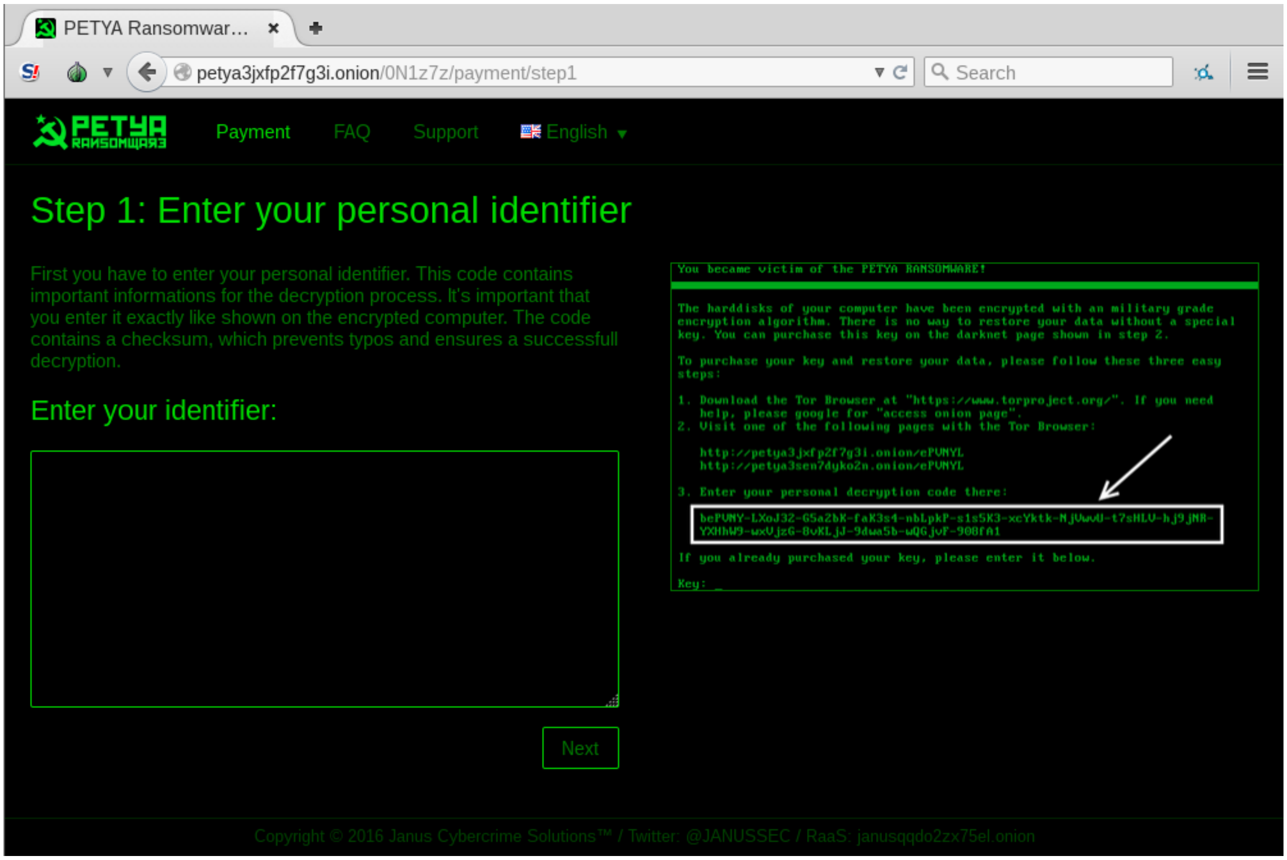
```
             uu$$$$$$$$$$$uu
          uu$$$$$$$$$$$$$$$$$uu
         u$$$$$$$$$$$$$$$$$$$$$u
         u$$$$$$$$$$$$$$$$$$$$$$$u
       u$$$$$$$$$$$$$$$$$$$$$$$$$u
       u$$$$$$$$$$$$$$$$$$$$$$$$$u
       u$$$$$$*   *$$$*   *$$$$$$u
       *$$$$*      u$u       $$$$*
        $$$u       u$u       u$$$
        $$$u      u$$$u      u$$$
         *$$$$uu$$$   $$$uu$$$$*
          *$$$$$$$*   *$$$$$$$*
            u$$$$$$$u$$$$$$$u
             u$*$*$*$*$*$*$u
  uuu        $$u$ $ $ $ $u$$       uuu
 u$$$$        $$$$$u$u$u$$$       u$$$$
  $$$$$uu      *$$$$$$$$$$*    uu$$$$$$$
u$$$$$$$$$$$uu    *****    uuuu$$$$$$$$$
$$$$***$$$$$$$$$$uuu   uu$$$$$$$$$***$$$*
 ***      **$$$$$$$$$$$uu **$***
          uuuu **$$$$$$$$$$uuu
 u$$$uuu$$$$$$$$$uu **$$$$$$$$$$$uuu$$$
 $$$$$$$$$$****           **$$$$$$$$$$$*
   *$$$$$*                      **$$$$**
     $$$*     PRESS ANY KEY!      $$$$*
```



```
You became victim of the PETYA RANSOMWARE!

The harddisks of your computer have been encrypted with an military grade
encryption algorithm. There is no way to restore your data without a special
key. You can purchase this key on the darknet page shown in step 2.

To purchase your key and restore your data, please follow these three easy
steps:

1. Download the Tor Browser at "https://www.torproject.org/". If you need
   help, please google for "access onion page".
2. Visit one of the following pages with the Tor Browser:

   http://petya3jxfp2f7g3i.onion/0N1z7z
   http://petya3sen7dyko2n.onion/0N1z7z

3. Enter your personal decryption code there:

   70N1z7-zjiXL3-npCpAT-Up4s37-GFB4iR-BnGsnx-y93cUR-q7qduM-cZkZkR-qo9D4f-
   JVufFR-c9UuAQ-rTSGBj-cmzDL4-dZ9hyU-908fA1

If you already purchased your key, please enter it below.

Key: _
```

This theme is consistent for the full ransomware – the same colors we can find on the page for the victim, and on the HTML with the ransom note dropped by Mischa.

Page for the victim:



Step 1: Enter your personal identifier

First you have to enter your personal identifier. This code contains important informations for the decryption process. It's important that you enter it exactly like shown on the encrypted computer. The code contains a checksum, which prevents typos and ensures a successfull decryption.

Enter your identifier:

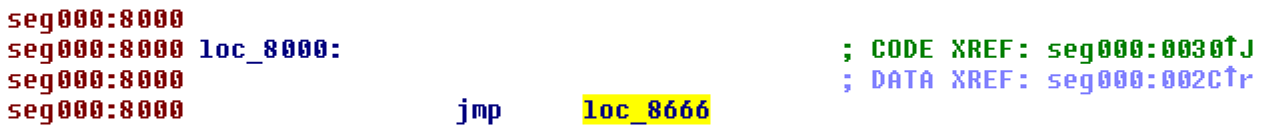
```
You became victim of the PETYA RANSOMWARE!

The harddisks of your computer have been encrypted with an military grade
encryption algorithm. There is no way to restore your data without a special
key. You can purchase this key on the darknet page shown in step 2.

To purchase your key and restore your data, please follow these three easy
steps:

1. Download the Tor Browser at "https://www.torproject.org/". If you need
   help, please google for "access onion page".
2. Visit one of the following pages with the Tor Browser:

   http://petya3jxfp2f7g3i.onion/ePVNYL
   http://petya3sen7dyko2n.onion/ePVNYL

3. Enter your personal decryption code there:

   bePVNY-LXoJ32-G5a2bK-faK3s4-nbLpkP-s1s5K3-xcYktk-NjVwvU-t7sHLV-hj9jNR-
   YXHhW9-wxVjzG-8vKLjJ-9dwa5b-wQGjvF-908fA1

If you already purchased your key, please enter it below.

Key: _
```

Copyright © 2016 Janus Cybercrime Solutions™ / Twitter: @JANUSSEC / RaaS: janusqqdo2zx75el.onion

Inside

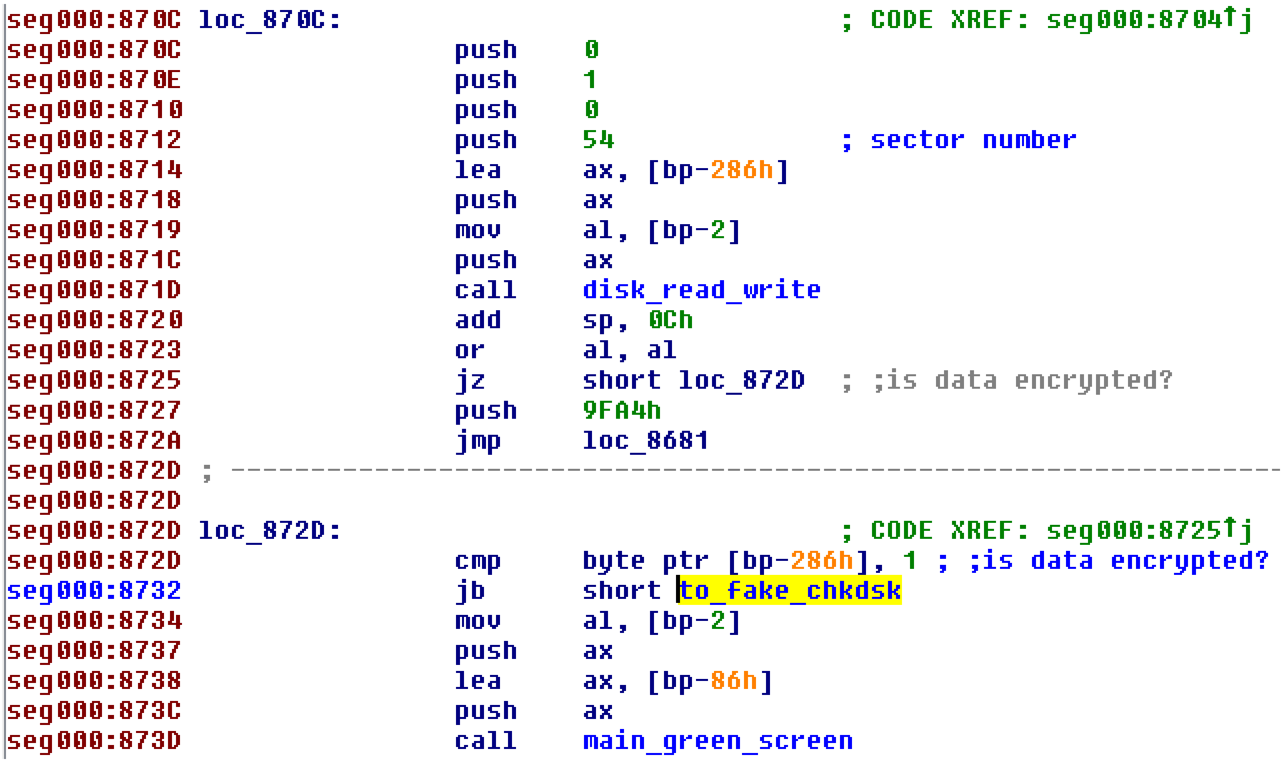
The new version of Petya uses exactly the same bootloader – again it loads 32 sectors starting from the sector 34 to the memory at 0x8000 and then jumps there.

Kernel start:



```
seg000:8000
seg000:8000 loc_8000:                               ; CODE XREF: seg000:0030↑J
seg000:8000                                         ; DATA XREF: seg000:002C↑r
seg000:8000                 jmp     loc_8666
```

Again, checking if the data is already encrypted is performed, using a one byte flag that is saved at the beginning of sector 54. If this flag is unset (the value of first byte is 0), program proceeds to the fake CHKDSK scan. Otherwise (if the value of the byte is 1), it displays the main green screen.



```
seg000:870C loc_870C:                               ; CODE XREF: seg000:8704↑j
seg000:870C                 push    0
seg000:870E                 push    1
seg000:8710                 push    0
seg000:8712                 push    54              ; sector number
seg000:8714                 lea     ax, [bp-286h]
seg000:8718                 push    ax
seg000:8719                 mov     al, [bp-2]
seg000:871C                 push    ax
seg000:871D                 call    disk_read_write
seg000:8720                 add     sp, 0Ch
seg000:8723                 or      al, al
seg000:8725                 jz      short loc_872D  ; ;is data encrypted?
seg000:8727                 push    9FA4h
seg000:872A                 jmp     loc_8681
seg000:872D ; ---------------------------------------------------------------------
seg000:872D
seg000:872D loc_872D:                               ; CODE XREF: seg000:8725↑j
seg000:872D                 cmp     byte ptr [bp-286h], 1 ; ;is data encrypted?
seg000:8732                 jb      short to_fake_chkdsk
seg000:8734                 mov     al, [bp-2]
seg000:8737                 push    ax
seg000:8738                 lea     ax, [bp-86h]
seg000:873C                 push    ax
seg000:873D                 call    main_green_screen
```

The key used for the encryption is again generated by the dropper and stored in the binary. That's why if we catch Petya at Stage 1 – before the fake CHKDSK run and erase it, recovering system is still easy. A live CD to support Stage 1 key recovery has been already released (here).

Key verification

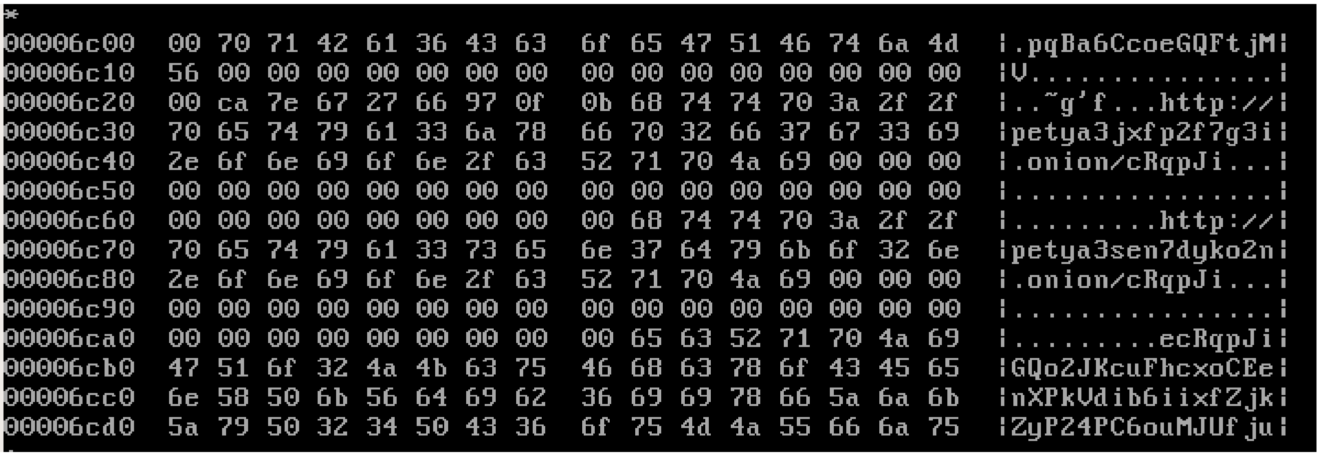
Key verification is performed in the following steps:

1. Input (***key***) from the user is read.
    - Accepted charset: *123456789abcdefghijkmnopqrstuvwxABCDEFGHJKLMNPQRSTUVWX* – if the character outside of this charset occurred, it is skipped.
    - Only the first **16 bytes** are stored
2. Data from sector 55 (512 bytes) is read into memory *// it will be denoted as **verification buffer***
3. The value stored at physical address 0x6c21 (just before the Tor address) is read into memory. It is an 8 byte long array, unique for a specific infection. *// it will be denoted as **nonce***
4. The ***verification buffer*** is encrypted by Salsa20 with the 16 byte long ***key*** and the ***nonce***
5. If, as a result of the applied procedure, ***verification buffer*** is fully filled with 0x7 – it means the supplied ***key*** is correct.

What changed in the new Petya?

Storing the Salsa key (stage1)

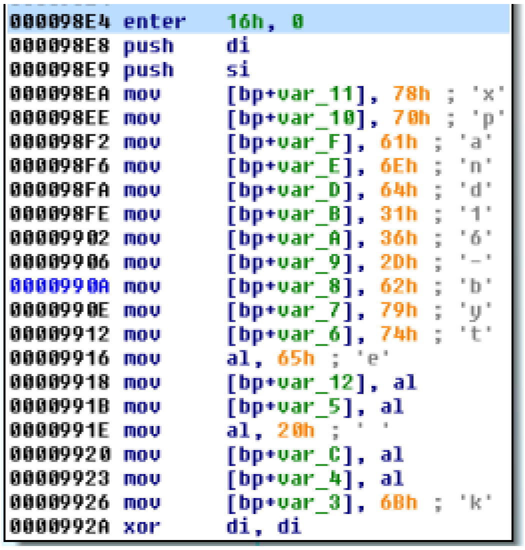
This time it the key is saved differently, without scrambling. Probably the authors realized that scrambling does not provide them any protection, so they gave up this idea completely:



```
*
00006c00  00 70 71 42 61 36 43 63  6f 65 47 51 46 74 6a 4d  |.pqBa6CcoeGQFtjM|
00006c10  56 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |V...............|
00006c20  00 ca 7e 67 27 66 97 0f  0b 68 74 74 70 3a 2f 2f  |..~g'f...http://|
00006c30  70 65 74 79 61 33 6a 78  66 70 32 66 37 67 33 69  |petya3jxfp2f7g3i|
00006c40  2e 6f 6e 69 6f 6e 2f 63  52 71 70 4a 69 00 00 00  |.onion/cRqpJi...|
00006c50  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00006c60  00 00 00 00 00 00 00 00  00 68 74 74 70 3a 2f 2f  |.........http://|
00006c70  70 65 74 79 61 33 73 65  6e 37 64 79 6b 6f 32 6e  |petya3sen7dyko2n|
00006c80  2e 6f 6e 69 6f 6e 2f 63  52 71 70 4a 69 00 00 00  |.onion/cRqpJi...|
00006c90  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00006ca0  00 00 00 00 00 00 00 00  00 65 63 52 71 70 4a 69  |.........ecRqpJi|
00006cb0  47 51 6f 32 4a 4b 63 75  46 68 63 78 6f 43 45 65  |GQo2JKcuFhcxoCEe|
00006cc0  6e 58 50 6b 56 64 69 62  36 69 69 78 66 5a 6a 6b  |nXPkVdib6iixfZjk|
00006cd0  5a 79 50 32 34 50 43 36  6f 75 4d 4a 55 66 6a 75  |ZyP24PC6ouMJUfju|
```

Length of Salsa key

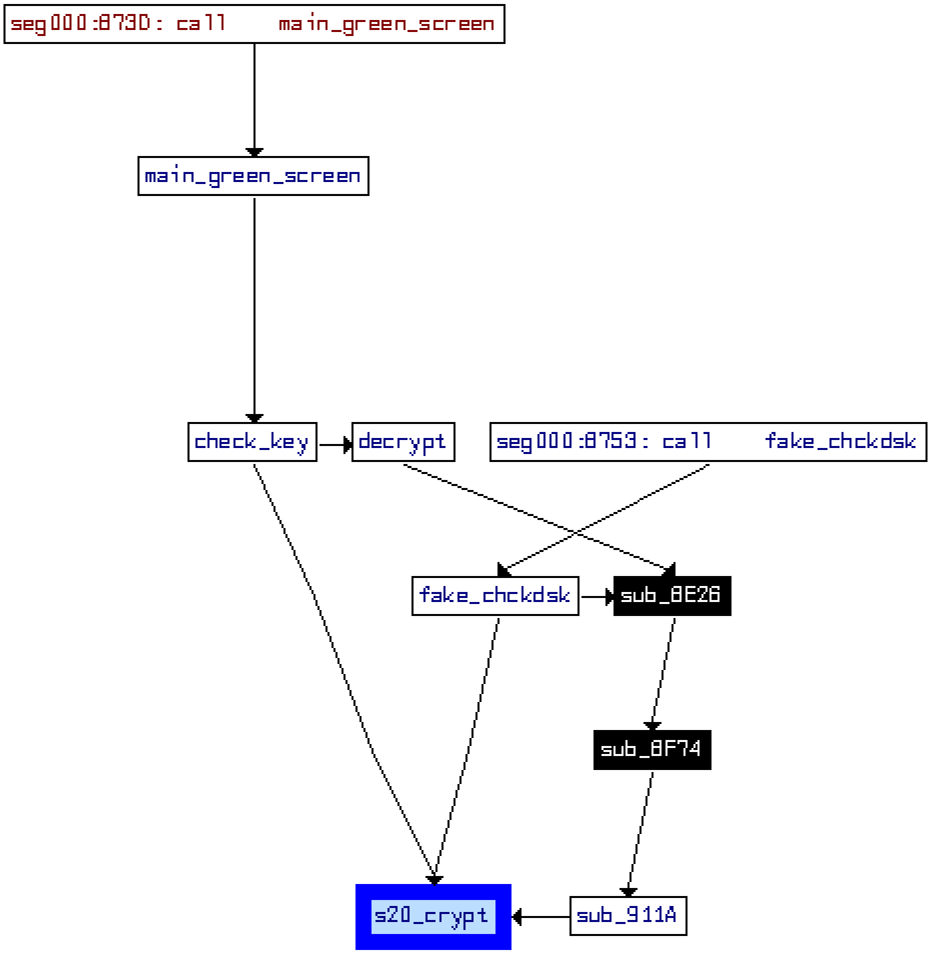
In the Red Petya authors used 16 byte long key – however, they were scrambling it and used it to make a 32 bit long key. Now they gave it up and they just use the 16 byte long key as it is. That's why, instead of functions **expand32** we will see in the new Petya the function **expand16**:



```
000098E4 enter   16h, 0
000098E8 push    di
000098E9 push    si
000098EA mov     [bp+var_11], 78h ; 'x'
000098EE mov     [bp+var_10], 70h ; 'p'
000098F2 mov     [bp+var_F], 61h ; 'a'
000098F6 mov     [bp+var_E], 6Eh ; 'n'
000098FA mov     [bp+var_D], 64h ; 'd'
000098FE mov     [bp+var_B], 31h ; '1'
00009902 mov     [bp+var_A], 36h ; '6'
00009906 mov     [bp+var_9], 2Dh ; '-'
0000990A mov     [bp+var_8], 62h ; 'b'
0000990E mov     [bp+var_7], 79h ; 'y'
00009912 mov     [bp+var_6], 74h ; 't'
00009916 mov     al, 65h ; 'e'
00009918 mov     [bp+var_12], al
0000991B mov     [bp+var_5], al
0000991E mov     al, 20h ; ' '
00009920 mov     [bp+var_C], al
00009923 mov     [bp+var_4], al
00009926 mov     [bp+var_3], 6Bh ; 'k'
0000992A xor     di, di
```

Salsa implementation

Also this time, Salsa20 is used in several places in Petya's code – for encryption, decryption and key verification. See the diagram below:



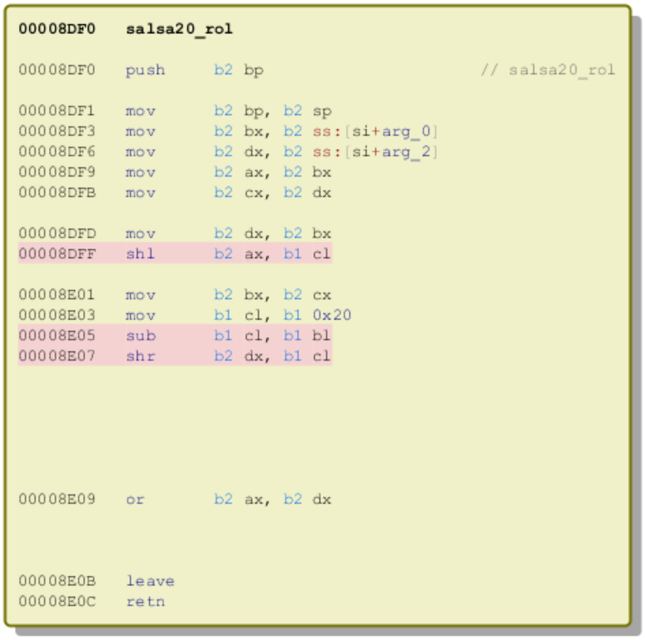
We made a comparison of Salsa implementation fragments, that have been found vulnerable in the previous implementation.

salsa20_rol

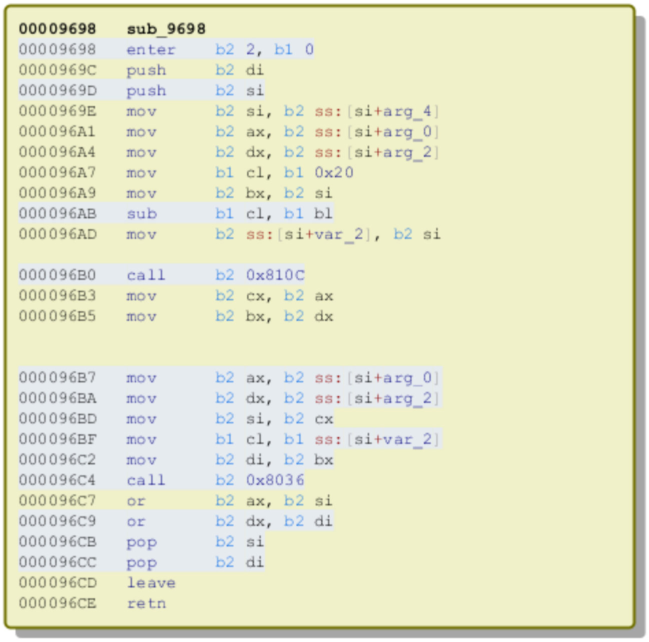
See below the original version of this function – copied from Salsa20 implementation:

```
static uint32_t rotl(uint32_t value, int shift)
{
  return (value << shift) | (value >> (32 - shift));
}
```

Code comparison – old one vs the new one:

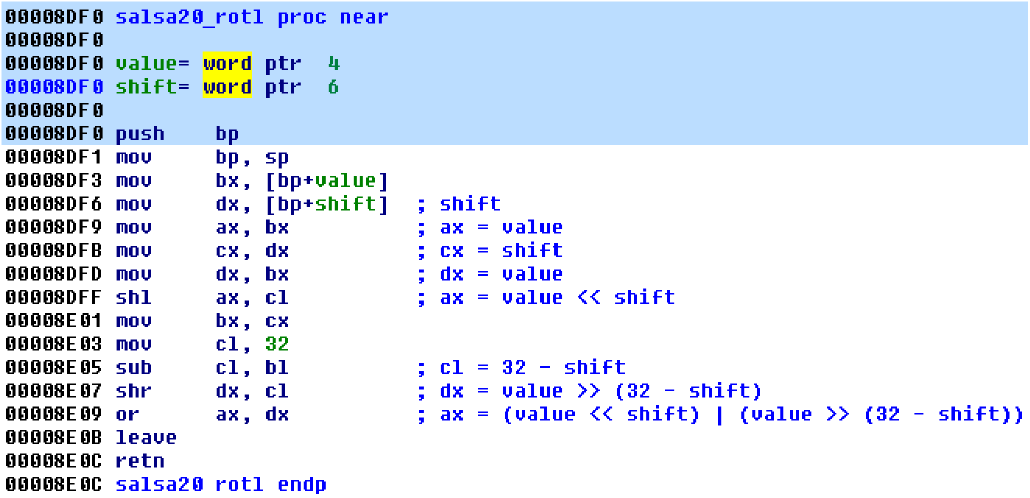


```
00008DF0   salsa20_rol

00008DF0   push      b2 bp                                // salsa20_rol

00008DF1   mov       b2 bp, b2 sp
00008DF3   mov       b2 bx, b2 ss:[si+arg_0]
00008DF6   mov       b2 dx, b2 ss:[si+arg_2]
00008DF9   mov       b2 ax, b2 bx
00008DFB   mov       b2 cx, b2 dx

00008DFD   mov       b2 dx, b2 bx
00008DFF   shl       b2 ax, b1 cl

00008E01   mov       b2 bx, b2 cx
00008E03   mov       b1 cl, b1 0x20
00008E05   sub       b1 cl, b1 bl
00008E07   shr       b2 dx, b1 cl






00008E09   or        b2 ax, b2 dx



00008E0B   leave
00008E0C   retn
```



```
00009698   sub_9698
00009698   enter     b2 2, b1 0
0000969C   push      b2 di
0000969D   push      b2 si
0000969E   mov       b2 si, b2 ss:[si+arg_4]
000096A1   mov       b2 ax, b2 ss:[si+arg_0]
000096A4   mov       b2 dx, b2 ss:[si+arg_2]
000096A7   mov       b1 cl, b1 0x20
000096A9   mov       b2 bx, b2 si
000096AB   sub       b1 cl, b1 bl
000096AD   mov       b2 ss:[si+var_2], b2 si

000096B0   call      b2 0x810C
000096B3   mov       b2 cx, b2 ax
000096B5   mov       b2 bx, b2 dx


000096B7   mov       b2 ax, b2 ss:[si+arg_0]
000096BA   mov       b2 dx, b2 ss:[si+arg_2]
000096BD   mov       b2 si, b2 cx
000096BF   mov       b1 cl, b1 ss:[si+var_2]
000096C2   mov       b2 di, b2 bx
000096C4   call      b2 0x8036
000096C7   or        b2 ax, b2 si
000096C9   or        b2 dx, b2 di
000096CB   pop       b2 si
000096CC   pop       b2 di
000096CD   leave
000096CE   retn
```

That’s how it has been implemented in the Red Petya:

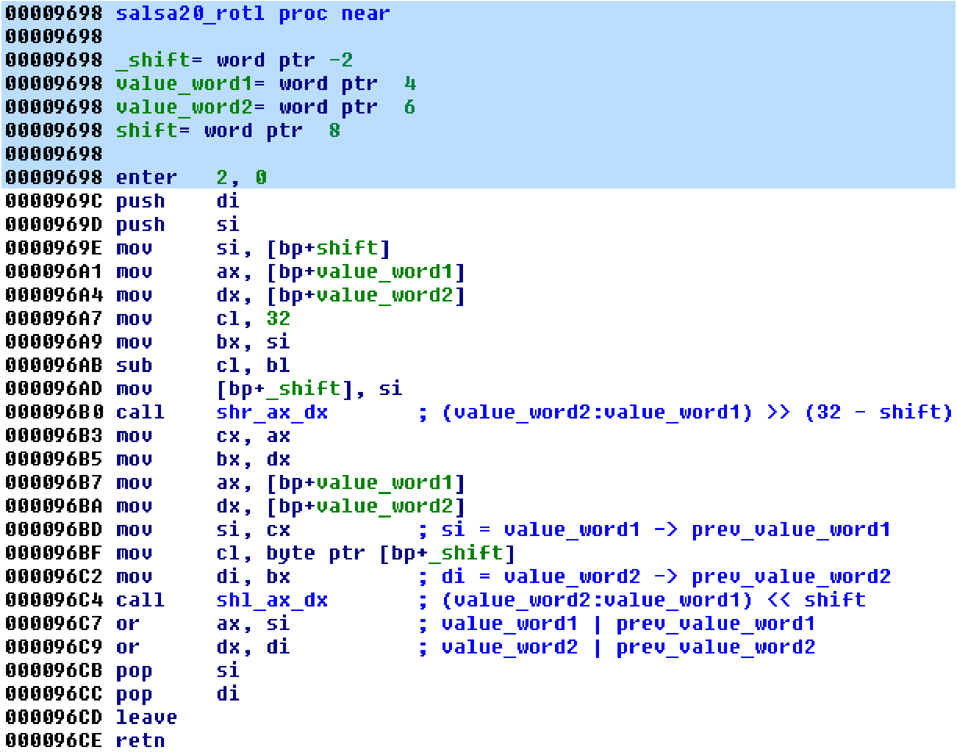


```
00008DF0 salsa20_rotl proc near
00008DF0
00008DF0 value= word ptr  4
00008DF0 shift= word ptr  6
00008DF0
00008DF0 push    bp
00008DF1 mov     bp, sp
00008DF3 mov     bx, [bp+value]
00008DF6 mov     dx, [bp+shift]  ; shift
00008DF9 mov     ax, bx          ; ax = value
00008DFB mov     cx, dx          ; cx = shift
00008DFD mov     dx, bx          ; dx = value
00008DFF shl     ax, cl          ; ax = value << shift
00008E01 mov     bx, cx
00008E03 mov     cl, 32
00008E05 sub     cl, bl          ; cl = 32 - shift
00008E07 shr     dx, cl          ; dx = value >> (32 - shift)
00008E09 or      ax, dx          ; ax = (value << shift) | (value >> (32 - shift))
00008E0B leave
00008E0C retn
00008E0C salsa20_rotl endp
```

The old version of this function was taking 2 arguments and it was an almost exact clone of the original function – the only difference was that it was using 16 bit variables. Reconstruction of the code:

```
static uint16_t rotl(uint16_t value, int16_t shift)
{
    return (value << shift) | (value >> (32 - shift));
}
```

And the new version – from Green Petya:



```
00009698 salsa20_rotl proc near
00009698
00009698 _shift= word ptr -2
00009698 value_word1= word ptr  4
00009698 value_word2= word ptr  6
00009698 shift= word ptr  8
00009698
00009698 enter   2, 0
0000969C push    di
0000969D push    si
0000969E mov     si, [bp+shift]
000096A1 mov     ax, [bp+value_word1]
000096A4 mov     dx, [bp+value_word2]
000096A7 mov     cl, 32
000096A9 mov     bx, si
000096AB sub     cl, bl
000096AD mov     [bp+_shift], si
000096B0 call    shr_ax_dx       ; (value_word2:value_word1) >> (32 - shift)
000096B3 mov     cx, ax
000096B5 mov     bx, dx
000096B7 mov     ax, [bp+value_word1]
000096BA mov     dx, [bp+value_word2]
000096BD mov     si, cx          ; si = value_word1 -> prev_value_word1
000096BF mov     cl, byte ptr [bp+_shift]
000096C2 mov     di, bx          ; di = value_word2 -> prev_value_word2
000096C4 call    shl_ax_dx       ; (value_word2:value_word1) << shift
000096C7 or      ax, si          ; value_word1 | prev_value_word1
000096C9 or      dx, di          ; value_word2 | prev_value_word2
000096CB pop     si
000096CC pop     di
000096CD leave
000096CE retn
```

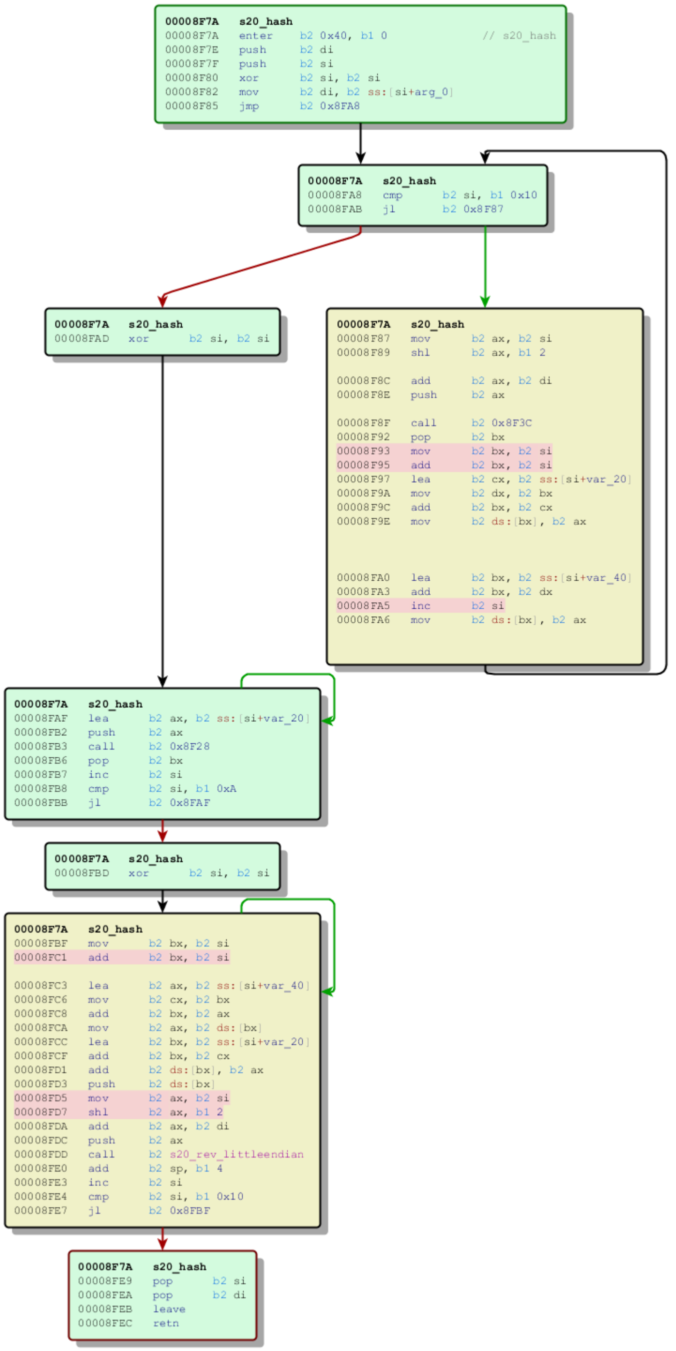
The new version is more complex – it takes 3 arguments and uses calls to additional helper functions. To achieve the functionality of shifting a DWORD, two WORD-sized parameters have been used (one representing the lower part of the DWORD and another representing the higher):

```
static uint16_t rotl(uint16_t value_word1, uint16_t value_word2, int16_t shift)
{
    return (shr(value_word1, value_word2, (32 - shift)) | shl(value_word1,
value_word2, shift));
}
```

s20_hash

original version: https://github.com/alexwebr/salsa20/blob/master/salsa20.c#L59

Code comparison – old one vs the new one:



```
00008F7A   s20_hash
00008F7A   enter     b2 0x40, b1 0                // s20_hash
00008F7E   push      b2 di
00008F7F   push      b2 si
00008F80   xor       b2 si, b2 si
00008F82   mov       b2 di, b2 ss:[si+arg_0]
00008F85   jmp       b2 0x8FA8
```

```
00008F7A   s20_hash
00008FA8   cmp       b2 si, b1 0x10
00008FAB   jl        b2 0x8F87
```

```
00008F7A   s20_hash
00008FAD   xor       b2 si, b2 si
```

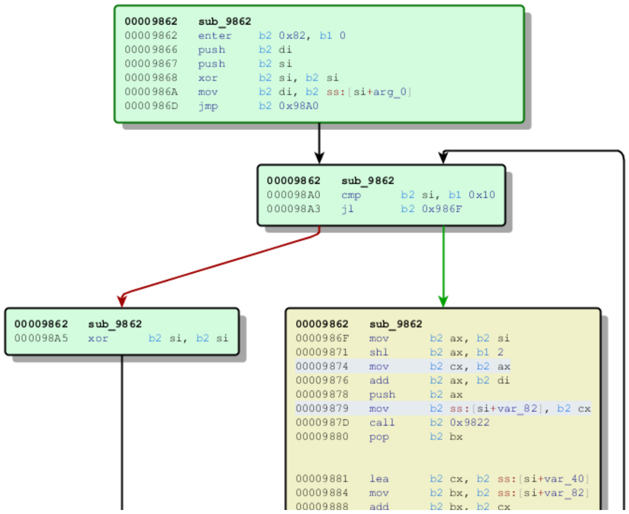
```
00008F7A   s20_hash
00008F87   mov       b2 ax, b2 si
00008F89   shl       b2 ax, b1 2

00008F8C   add       b2 ax, b2 di
00008F8E   push      b2 ax

00008F8F   call      b2 0x8F3C
00008F92   pop       b2 bx
00008F93   mov       b2 bx, b2 si
00008F95   add       b2 bx, b2 si
00008F97   lea       b2 cx, b2 ss:[si+var_20]
00008F9A   mov       b2 dx, b2 bx
00008F9C   add       b2 bx, b2 cx
00008F9E   mov       b2 ds:[bx], b2 ax



00008FA0   lea       b2 bx, b2 ss:[si+var_40]
00008FA3   add       b2 bx, b2 dx
00008FA5   inc       b2 si
00008FA6   mov       b2 ds:[bx], b2 ax
```

```
00008F7A   s20_hash
00008FAF   lea       b2 ax, b2 ss:[si+var_20]
00008FB2   push      b2 ax
00008FB3   call      b2 0x8F28
00008FB6   pop       b2 bx
00008FB7   inc       b2 si
00008FB8   cmp       b2 si, b1 0xA
00008FBB   jl        b2 0x8FAF
```

```
00008F7A   s20_hash
00008FBD   xor       b2 si, b2 si
```

```
00008F7A   s20_hash
00008FBF   mov       b2 bx, b2 si
00008FC1   add       b2 bx, b2 si

00008FC3   lea       b2 ax, b2 ss:[si+var_40]
00008FC6   mov       b2 cx, b2 bx
00008FC8   add       b2 bx, b2 ax
00008FCA   mov       b2 ax, b2 ds:[bx]
00008FCC   lea       b2 bx, b2 ss:[si+var_20]
00008FCF   add       b2 bx, b2 cx
00008FD1   add       b2 ds:[bx], b2 ax
00008FD3   push      b2 ds:[bx]
00008FD5   mov       b2 ax, b2 si
00008FD7   shl       b2 ax, b1 2
00008FDA   add       b2 ax, b2 di
00008FDC   push      b2 ax
00008FDD   call      b2 s20_rev_littleendian
00008FE0   add       b2 sp, b1 4
00008FE3   inc       b2 si
00008FE4   cmp       b2 si, b1 0x10
00008FE7   jl        b2 0x8FBF
```

```
00008F7A   s20_hash
00008FE9   pop       b2 si
00008FEA   pop       b2 di
00008FEB   leave
00008FEC   retn
```

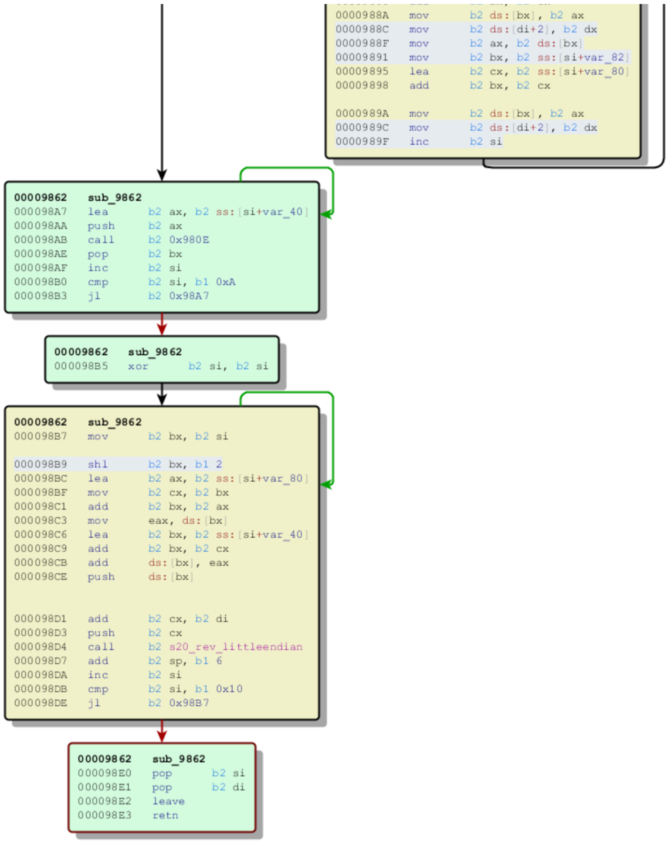


```
00009862   sub_9862
00009862   enter     b2 0x82, b1 0
00009866   push      b2 di
00009867   push      b2 si
00009868   xor       b2 si, b2 si
0000986A   mov       b2 di, b2 ss:[si+arg_0]
0000986D   jmp       b2 0x98A0
```

```
00009862   sub_9862
000098A0   cmp       b2 si, b1 0x10
000098A3   jl        b2 0x986F
```

```
00009862   sub_9862
000098A5   xor       b2 si, b2 si
```

```
00009862   sub_9862
0000986F   mov       b2 ax, b2 si
00009871   shl       b2 ax, b1 2
00009874   mov       b2 cx, b2 ax
00009876   add       b2 ax, b2 di
00009878   push      b2 ax
00009879   mov       b2 ss:[si+var_82], b2 cx
0000987D   call      b2 0x9822
00009880   pop       b2 bx


00009881   lea       b2 cx, b2 ss:[si+var_40]
00009884   mov       b2 bx, b2 ss:[si+var_82]
00009888   add       b2 bx, b2 cx
```



```
0000988A   mov      b2 ds:[bx], b2 ax
0000988C   mov      b2 ds:[di+2], b2 dx
0000988F   mov      b2 ax, b2 ds:[bx]
00009891   mov      b2 bx, b2 ss:[si+var_82]
00009895   lea      b2 cx, b2 ss:[si+var_80]
00009898   add      b2 bx, b2 cx

0000989A   mov      b2 ds:[bx], b2 ax
0000989C   mov      b2 ds:[di+2], b2 dx
0000989F   inc      b2 si
```

```
00009862   sub_9862
000098A7   lea      b2 ax, b2 ss:[si+var_40]
000098AA   push     b2 ax
000098AB   call     b2 0x980E
000098AE   pop      b2 bx
000098AF   inc      b2 si
000098B0   cmp      b2 si, b1 0xA
000098B3   jl       b2 0x98A7
```

```
00009862   sub_9862
000098B5   xor      b2 si, b2 si
```

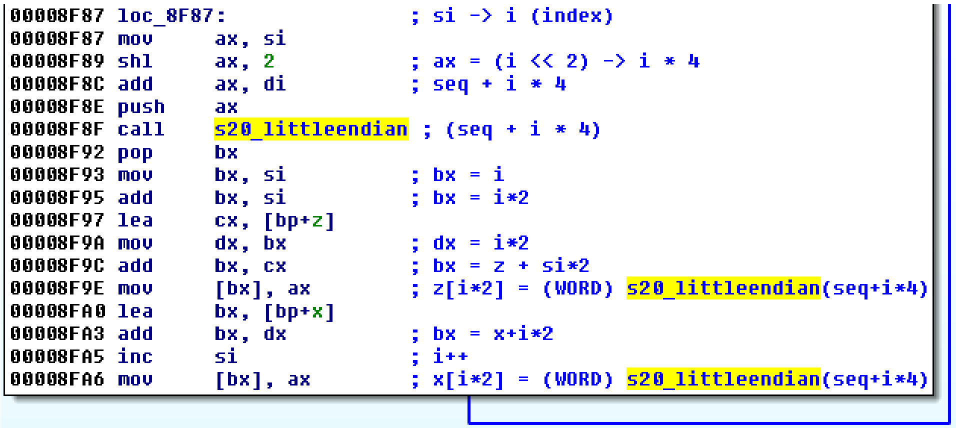
```
00009862   sub_9862
000098B7   mov      b2 bx, b2 si

000098B9   shl      b2 bx, b1 2
000098BC   lea      b2 ax, b2 ss:[si+var_80]
000098BF   mov      b2 cx, b2 bx
000098C1   add      b2 bx, b2 ax
000098C3   mov      eax, ds:[bx]
000098C6   lea      b2 bx, b2 ss:[si+var_40]
000098C9   add      b2 bx, b2 cx
000098CB   add      ds:[bx], eax
000098CE   push     ds:[bx]


000098D1   add      b2 cx, b2 di
000098D3   push     b2 cx
000098D4   call     b2 s20_rev_littleendian
000098D7   add      b2 sp, b1 6
000098DA   inc      b2 si
000098DB   cmp      b2 si, b1 0x10
000098DE   jl       b2 0x98B7
```

```
00009862   sub_9862
000098E0   pop      b2 si
000098E1   pop      b2 di
000098E2   leave
000098E3   retn
```

First changed fragment corresponds to this part of the original Salsa20 implementation:

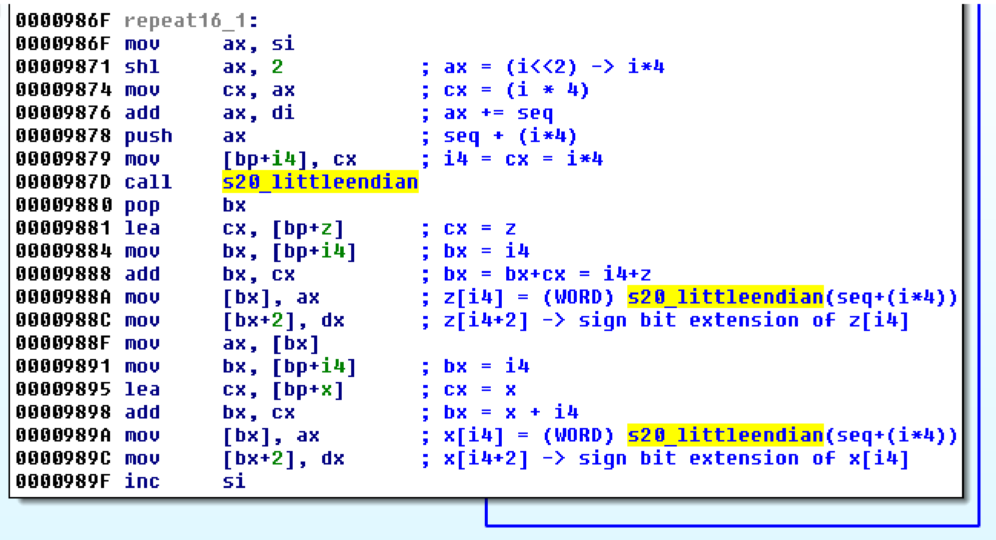
```
for (i = 0; i < 16; ++i)
  x[i] = z[i] = s20_littleendian(seq + (4 * i));
```

That's how it has been implemented in the Red Petya:



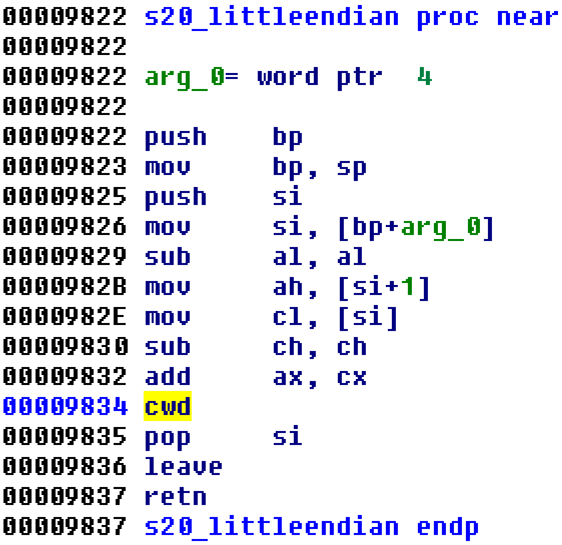
```
00008F87 loc_8F87:                ; si -> i (index)
00008F87 mov     ax, si
00008F89 shl     ax, 2            ; ax = (i << 2) -> i * 4
00008F8C add     ax, di           ; seq + i * 4
00008F8E push    ax
00008F8F call    s20_littleendian ; (seq + i * 4)
00008F92 pop     bx
00008F93 mov     bx, si           ; bx = i
00008F95 add     bx, si           ; bx = i*2
00008F97 lea     cx, [bp+z]
00008F9A mov     dx, bx           ; dx = i*2
00008F9C add     bx, cx           ; bx = z + si*2
00008F9E mov     [bx], ax         ; z[i*2] = (WORD) s20_littleendian(seq+i*4)
00008FA0 lea     bx, [bp+x]
00008FA3 add     bx, dx           ; bx = x+i*2
00008FA5 inc     si               ; i++
00008FA6 mov     [bx], ax         ; x[i*2] = (WORD) s20_littleendian(seq+i*4)
```

And the new version – from the Green Petya:



```
0000986F repeat16_1:
0000986F mov     ax, si
00009871 shl     ax, 2             ; ax = (i<<2) -> i*4
00009874 mov     cx, ax            ; cx = (i * 4)
00009876 add     ax, di            ; ax += seq
00009878 push    ax                ; seq + (i*4)
00009879 mov     [bp+i4], cx       ; i4 = cx = i*4
0000987D call    s20_littleendian
00009880 pop     bx
00009881 lea     cx, [bp+z]        ; cx = z
00009884 mov     bx, [bp+i4]       ; bx = i4
00009888 add     bx, cx            ; bx = bx+cx = i4+z
0000988A mov     [bx], ax          ; z[i4] = (WORD) s20_littleendian(seq+(i*4))
0000988C mov     [bx+2], dx        ; z[i4+2] -> sign bit extension of z[i4]
0000988F mov     ax, [bx]
00009891 mov     bx, [bp+i4]       ; bx = i4
00009895 lea     cx, [bp+x]        ; cx = x
00009898 add     bx, cx            ; bx = x + i4
0000989A mov     [bx], ax          ; x[i4] = (WORD) s20_littleendian(seq+(i*4))
0000989C mov     [bx+2], dx        ; x[i4+2] -> sign bit extension of x[i4]
0000989F inc     si
```

s20_littleendian also changed, but very slightly – a bit extension has been added:

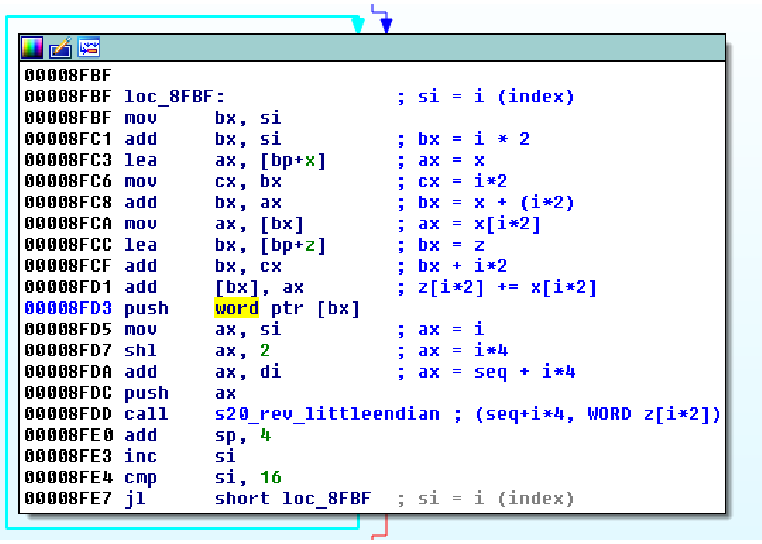


```
00009822 s20_littleendian proc near
00009822
00009822 arg_0= word ptr  4
00009822
00009822 push    bp
00009823 mov     bp, sp
00009825 push    si
00009826 mov     si, [bp+arg_0]
00009829 sub     al, al
0000982B mov     ah, [si+1]
0000982E mov     cl, [si]
00009830 sub     ch, ch
00009832 add     ax, cx
00009834 cwd
00009835 pop     si
00009836 leave
00009837 retn
00009837 s20_littleendian endp
```

Second changed fragment corresponds to this part of the original Salsa20 implementation:

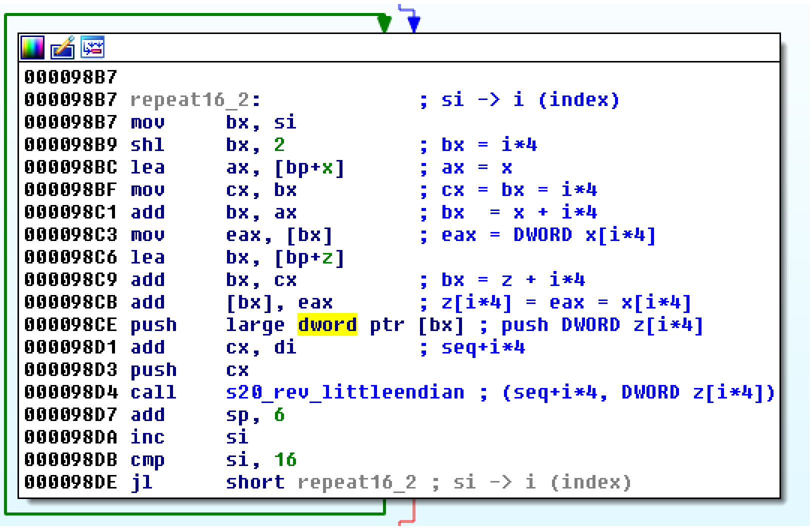
```
for (i = 0; i < 16; ++i) {
  z[i] += x[i];
  s20_rev_littleendian(seq + (4 * i), z[i]);
}
```

That’s how it has been implemented in the Red Petya:



```
00008FBF
00008FBF loc_8FBF:                  ; si = i (index)
00008FBF mov     bx, si
00008FC1 add     bx, si             ; bx = i * 2
00008FC3 lea     ax, [bp+x]         ; ax = x
00008FC6 mov     cx, bx             ; cx = i*2
00008FC8 add     bx, ax             ; bx = x + (i*2)
00008FCA mov     ax, [bx]           ; ax = x[i*2]
00008FCC lea     bx, [bp+z]         ; bx = z
00008FCF add     bx, cx             ; bx + i*2
00008FD1 add     [bx], ax           ; z[i*2] += x[i*2]
00008FD3 push    word ptr [bx]
00008FD5 mov     ax, si             ; ax = i
00008FD7 shl     ax, 2              ; ax = i*4
00008FDA add     ax, di             ; ax = seq + i*4
00008FDC push    ax
00008FDD call    s20_rev_littleendian ; (seq+i*4, WORD z[i*2])
00008FE0 add     sp, 4
00008FE3 inc     si
00008FE4 cmp     si, 16
00008FE7 jl      short loc_8FBF     ; si = i (index)
```

And the new version – from Green Petya:



```
000098B7
000098B7 repeat16_2:                ; si -> i (index)
000098B7 mov     bx, si
000098B9 shl     bx, 2              ; bx = i*4
000098BC lea     ax, [bp+x]         ; ax = x
000098BF mov     cx, bx             ; cx = bx = i*4
000098C1 add     bx, ax             ; bx  = x + i*4
000098C3 mov     eax, [bx]          ; eax = DWORD x[i*4]
000098C6 lea     bx, [bp+z]
000098C9 add     bx, cx             ; bx = z + i*4
000098CB add     [bx], eax          ; z[i*4] = eax = x[i*4]
000098CE push    large dword ptr [bx] ; push DWORD z[i*4]
000098D1 add     cx, di             ; seq+i*4
000098D3 push    cx
000098D4 call    s20_rev_littleendian ; (seq+i*4, DWORD z[i*4])
000098D7 add     sp, 6
000098DA inc     si
000098DB cmp     si, 16
000098DE jl      short repeat16_2 ; si -> i (index)
```

Full reconstruction and comparison of implementations of this function:
https://gist.github.com/hasherezade/f59939f5d20ebdfd36343dfcae66bfa9

New Petya, new bug

As we can see, the authors tried to fix the bug of using 16 bit long units where the 32 bit long units were required. The new implementation of Salsa looks *almost* correct… However, due to *just one bug*, it still needs only 8 valid characters of the key, out of 16!

The bug lies in invalid implementation of the function **s20_littleendian**. That’s how this function looks in the origial Salsa20:

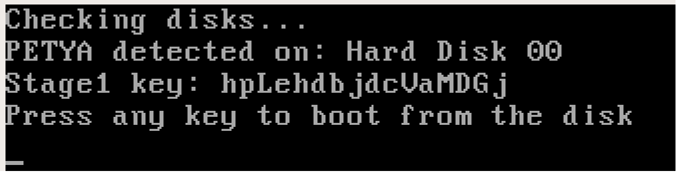
```
static uint32_t s20_littleendian(uint8_t *b)
{
  return b[0] +
         (b[1] << 8) +
         (b[2] << 16) +
         (b[3] << 24);
}
```

And that's how the Petya's version looks:

```
static int16_t s20_littleendian(uint8_t *b)
{
  return b[0] +
         (b[1] << 8);
}
```

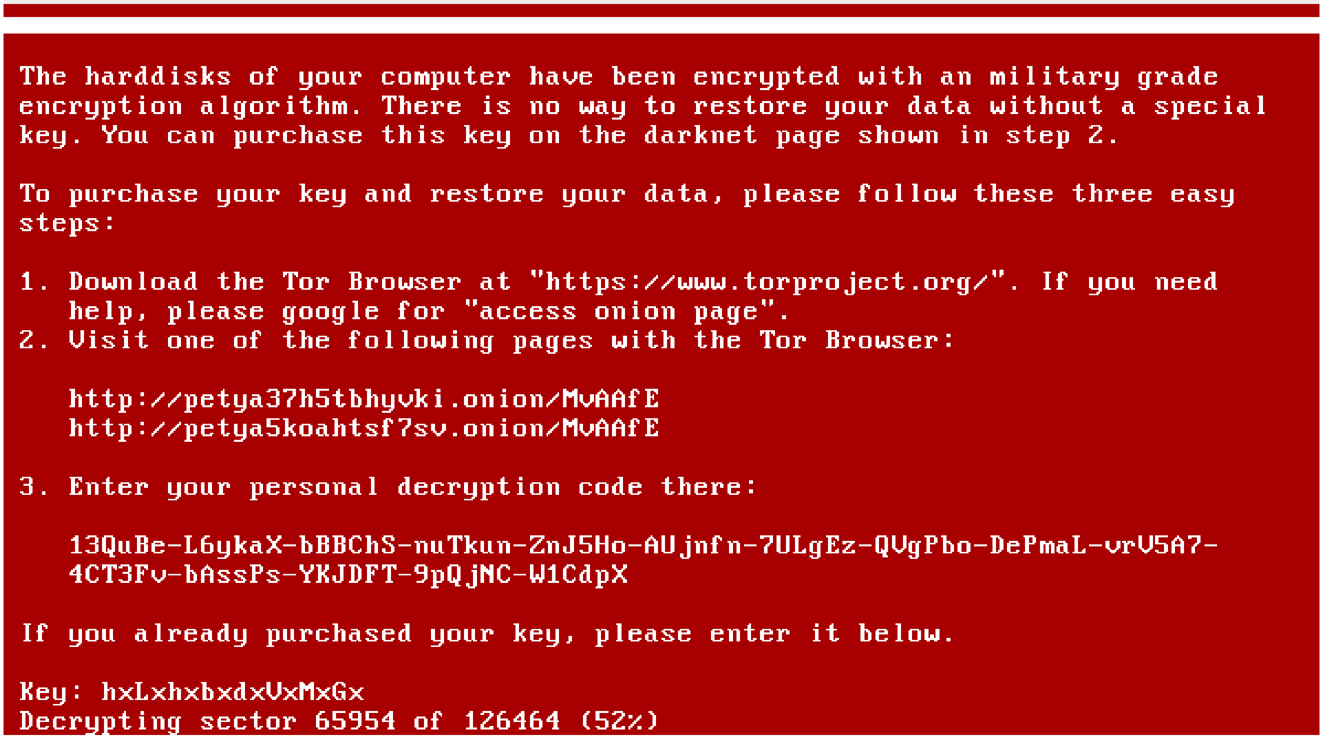
In the previous, Red Petya every second character of the key had no importance to the encryption/decryption. The pattern was: **c?c?c?c?c?c?c?c?** – where the **c** means a valid character and **?** means any random character from the set.

Valid key – **hpLehdbjdcVaMDGj** (revealed at Stage 1 by Antipetya Live CD)



```
Checking disks...
PETYA detected on: Hard Disk 00
Stage1 key: hpLehdbjdcVaMDGj
Press any key to boot from the disk
_
```

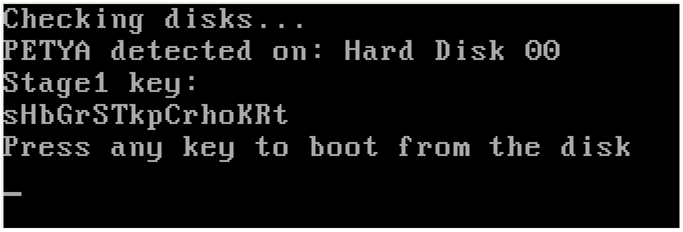
Accepted key **hxLxhxbxdxVxMxGx** :



```
The harddisks of your computer have been encrypted with an military grade
encryption algorithm. There is no way to restore your data without a special
key. You can purchase this key on the darknet page shown in step 2.

To purchase your key and restore your data, please follow these three easy
steps:

1. Download the Tor Browser at "https://www.torproject.org/". If you need
   help, please google for "access onion page".
2. Visit one of the following pages with the Tor Browser:

   http://petya37h5tbhyvki.onion/MvAAfE
   http://petya5koahtsf7sv.onion/MvAAfE

3. Enter your personal decryption code there:

   13QuBe-L6ykaX-bBBChS-nuTkun-ZnJ5Ho-AUjnfn-7ULgEz-QVgPbo-DePmaL-vrV5A7-
   4CT3Fv-bAssPs-YKJDFT-9pQjNC-W1CdpX

If you already purchased your key, please enter it below.

Key: hxLxhxbxdxVxMxGx
Decrypting sector 65954 of 126464 (52%)
```

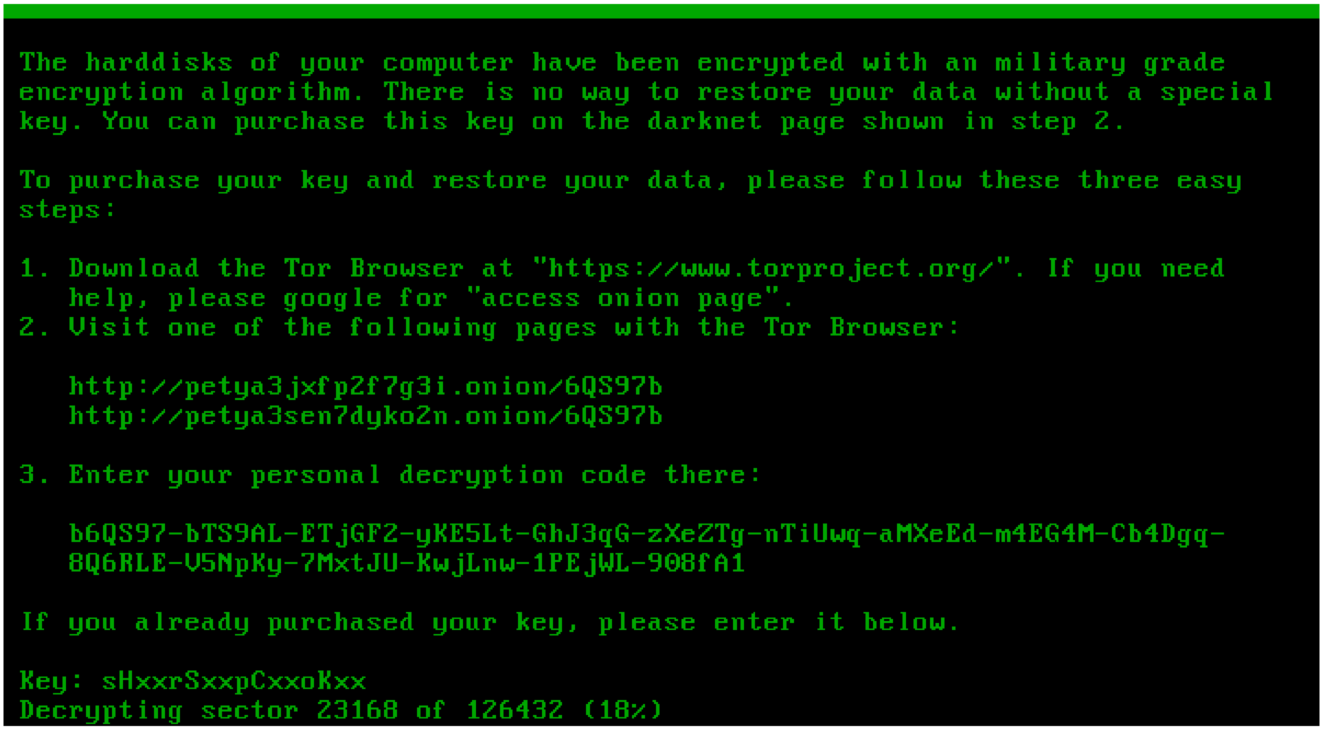
In the current case, the pattern is a bit different (and more difficult to exploit): **cc??cc??cc??cc??**. See example below:

Valid key – **sHbGrSTkpCrhoKRt** (revealed at Stage 1 by Antipetya Live CD)



```
Checking disks...
PETYA detected on: Hard Disk 00
Stage1 key:
sHbGrSTkpCrhoKRt
Press any key to boot from the disk

_
```

Accepted key – **sHxxrSxxpCxxoKxx** :



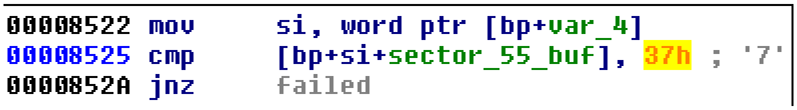
```
The harddisks of your computer have been encrypted with an military grade
encryption algorithm. There is no way to restore your data without a special
key. You can purchase this key on the darknet page shown in step 2.

To purchase your key and restore your data, please follow these three easy
steps:

1. Download the Tor Browser at "https://www.torproject.org/". If you need
   help, please google for "access onion page".
2. Visit one of the following pages with the Tor Browser:

   http://petya3jxfp2f7g3i.onion/6QS97b
   http://petya3sen7dyko2n.onion/6QS97b

3. Enter your personal decryption code there:

   b6QS97-bTS9AL-ETjGF2-yKE5Lt-GhJ3qG-zXeZTg-nTiUwq-aMXeEd-m4EG4M-Cb4Dgq-
   8Q6RLE-V5NpKy-7MxtJU-KwjLnw-1PEjWL-908fA1

If you already purchased your key, please enter it below.

Key: sHxxrSxxpCxxoKxx
Decrypting sector 23168 of 126432 (18%)
```

Verification buffer

Similar to the previous version, a verification buffer is used in order to check whether or not the provided key is valid. However, values expected in the verification buffer changed. In the green Petya, a key was passed as valid if the verification buffer got filled by ASCII character '7' (that is byte 0x37). Now, the byte 0x7 has been used.

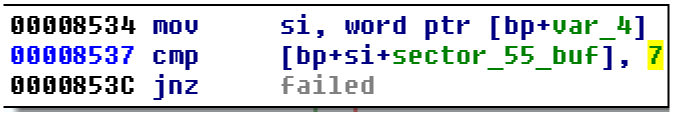
Checking characters of the validation buffer:

in red Petya:



```
00008522 mov     si, word ptr [bp+var_4]
00008525 cmp     [bp+si+sector_55_buf], 37h ; '7'
0000852A jnz     failed
```

in green Petya



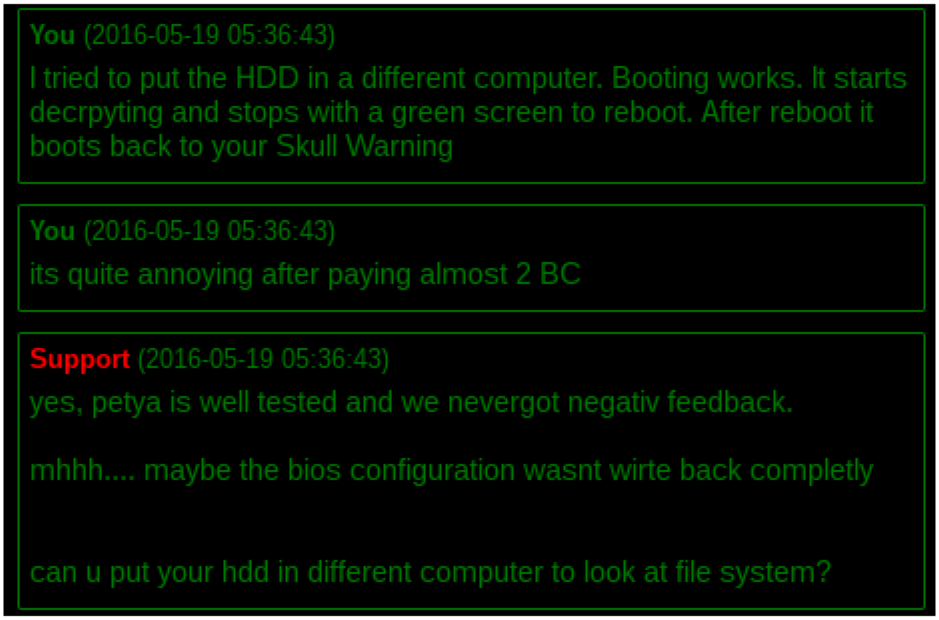
```
00008534 mov     si, word ptr [bp+var_4]
00008537 cmp     [bp+si+sector_55_buf], 7
0000853C jnz     failed
```

Stability

In the Red Petya, interrupting the fake CHKDSK caused real problems. Even after having a correct key full disk was not decrypted correctly – because Petya didn't check which sectors are encrypted and which are not, and was applying Salsa again on everything. In the Green edition authors improved it.

Yet, some victims of Petya reported to us, that they encountered the situation, where they bought a valid key but still the disk wasn't decrypted properly. That's why we recommend making a dump of the full disk as soon as you notice that you are infected with Petya (before trying any decryption methods).

Fragment of the talk between the victim and the attackers. Till now, user didn't got either his data or his bitcoins back:



You (2016-05-19 05:36:43)
I tried to put the HDD in a different computer. Booting works. It starts decrpyting and stops with a green screen to reboot. After reboot it boots back to your Skull Warning

You (2016-05-19 05:36:43)
its quite annoying after paying almost 2 BC

Support (2016-05-19 05:36:43)
yes, petya is well tested and we nevergot negativ feedback.

mhhh.... maybe the bios configuration wasnt wirte back completly

can u put your hdd in different computer to look at file system?

Conclusion

The new Petya comes with significant improvements. The authors realized the weakness and tried to make appropriate fixes in the code. However, they left another flaw that weakens the encryption. Unfortunately, the previous approach, based on genetic algorithm will not work this time – due to the different specifics of the generated output. The remaining solution seems to be only bruteforce of the 8 characters. Further research about the possibility of writing a decryptor is in progress.

The idea of making a bundle of two completely different ransomwares is new and creative. The group of cybercriminals who released it seems to do everything in order to gain clients on the black market. Probably the same group released other ransomware before: Chimera and Rokku. The main method of distribution used by them is via targeted campaigns of malicious e-mails. That's why we recommend to pay more attention on the received attachments and be very cautious. We can expect, that they will come with some new ideas in the future.

Appendix

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

About the previous (red) version of Petya:

https://blog.malwarebytes.org/threat-analysis/2016/04/petya-ransomware/