EternalPetya and the lost Salsa20 key

blog.malwarebytes.com/threat-analysis/2017/06/eternalpetya-lost-salsa20-key/

Malwarebytes Labs June 29, 2017

We have recently been facing a huge outbreak of a new Petya-like malware armed with an infector similar to WannaCry. The research is still in progress, and the full report will be published soon.

In this post, we will focus on some new important aspects of the current malware. The low-level attack works in the same style as the first Petya, described here. As before, the beginning of the disk is overwritten by the malicious Petya kernel and bootloader. When the malicious kernel is booted, it encrypts the Master File Table with Salsa20 and in this way, makes the disk inaccessible.

The code from Petya's kernel didn't change much, but the new logic implemented in the high-level part (the Windows executable) caused the change in the malware's mission. In the past, after paying the ransom, the Salsa key from the victim was restored and with its help, the Petya kernel was able to decrypt the Master File Table. Now, the necessary key seems to be lost for eternity. Thus, the malware appears to have only damaging intentions.

Let's have a look at the implementation and discuss the details.

Analyzed sample:

71b6a493388e7d0b40c83ce903bc6b04 – the main DLL
f3471d609077479891218b0f93a77ceb – the low level part (Petya bootloader + kernel)

[UPDATE] A small bug in the Salsa20 implementation has been found. Unfortunately, it is not significant enough to help restoring the key.

How is the disk encrypted?

The low level attack affecting the Master File Table hasn't changed since Goldeneye. It is executed by the Petya kernel.

The Salsa20 algorithm that was implemented incorrectly in the early versions of Petya and caused it to be cracked has been fixed in version 3 (read more here). Now it looks almost the same as in Goldeneye (that was the 4th step in the evolution) and it does not seem to have any significant bugs. Thus, once the data is encrypted, having the valid key is the only way to restore it.

Here's a comparison of the changes in the code between the current version and the Goldeneye one.

similarity	confide	change	EA primary	name primary	EA secondary
1.00	0.99	-------	000088C4	sub_88C4_13	000888C4
1.00	0.99	-------	00008972	sub_8972_19	00088972
1.00	0.99	-------	0000899A	sub_899A_20	0008899A
1.00	0.99	-------	000089B2	sub_89B2_21	000889B2
1.00	0.99	-------	000089CA	read_input	000889CA
1.00	0.99	-------	00008A64	sub_8A64_23	00088A64
1.00	0.99	-------	00008B9A	sub_8B9A_24	00088B9A
1.00	0.99	-------	00008BF2	sub_8BF2_25	00088BF2
1.00	0.99	-------	00008C98	enc_dec_disk	00088C98
1.00	0.99	-------	00009386	sub_9386_26	00089386
1.00	0.99	-------	00009652	s20_hash	00089652
1.00	0.99	-------	000096D4	s20_expand_key	000896D4
1.00	0.99	-------	00009798	s20_crypt	00089798
1.00	0.99	-------	0000998E	sub_998E_36	0008998E
1.00	0.99	-------	000099FC	sub_99FC_37	000899FC
1.00	0.99	-------	000082A2	sub_82A2_8	000882A2
1.00	0.99	-------	000098D6	sub_98D6_35	000898D6
1.00	0.99	-------	00008FA6	encrypt_mft	00088FA6
1.00	0.99	-------	00008DE2	find_and_encrypt_mft	00088DE2
1.00	0.99	-------	0000811A	fake_chkdsk	0008811A
1.00	0.99	-------	00008212	display_reboot_request	00088212
1.00	0.99	-------	000085CE	screen_output	000885CE
1.00	0.99	-------	00008726	sub_8726_12	00088726
1.00	0.99	-------	00008932	sub_8932_15	00088932
1.00	0.99	-------	00008A54	sub_8A54_22	00088A54
1.00	0.99	-------	00009462	sub_9462_27	00089462
1.00	0.99	-------	0000949A	sub_949A_28	0008949A
1.00	0.99	-------	000095D8	sub_95D8_31	000895D8
1.00	0.99	-------	000095EC	sub_95EC_32	000895EC
1.00	0.99	-------	00009628	s20_rev_little_endian	00089628
1.00	0.99	-------	00009878	sub_9878_33	00089878
1.00	0.99	-------	0000989C	sub_989C_34	0008989C
1.00	0.98	-------	00008684	display_strings	00088684
1.00	0.98	-------	0000891E	sub_891E_14	0008891E
1.00	0.98	-------	00008948	sub_8948_16	00088948
1.00	0.98	-------	00008950	sub_8950_17	00088950
1.00	0.98	-------	0000896A	sub_896A_18	0008896A
1.00	0.98	-------	00008C5A	disk_read_or_write	00088C5A
1.00	0.88	-------	00009518	sub_9518_29	00089518
1.00	0.88	-------	00009578	sub_9578_30	00089578
0.99	0.99	-I--E--	00008426	main_info_screen	00088426
0.16	0.38	GI--EL-	000086E0	sub_86E0_11	000886E0

Looking inside the code, we can see that the significant changes have been made only to the elements responsible for displaying the screen with information.

```
00008426 main_info_screen proc near
00008426
00008426 var_24C= byte ptr -24Ch
00008426 var_223= byte ptr -223h
00008426 var_1E3= byte ptr -1E3h
00008426 var_1A3= byte ptr -1A3h
00008426 var_4C= byte ptr -4Ch
00008426 var_1= byte ptr -1
00008426 arg_0= word ptr  4
00008426 arg_2= byte ptr  6
00008426
00008426 enter   24Ch, 0
0000842A push    di
0000842B push    si
0000842C call    sub_86E0
0000842F push    0
00008431 push    1
00008433 push    0
00008435 push    20h ; ' '
00008437 lea     ax, [bp+var_24C]
0000843B push    ax
0000843C mov     al, [bp+arg_2]
0000843F push    ax
00008440 call    disk_read_or_write
00008443 add     sp, 0Ch
00008446 push    9CA6h
00008449 call    display_string
0000844C pop     bx
0000844D push    50h ; 'P'
0000844F push    0FFDCh
00008451 call    sub_8660
00008454 add     sp, 4
00008457 push    9CD6h          ; "If you see this text, then your files..."
0000845A call    display_string
```

Another subtle, yet interesting change is in the Salsa20 key expansion function. Although the Salsa20 algorithm itself was not altered, there is one keyword that got changed in comparison to the original version. This is the fragment of the current sample's code:

```
seg000:96D4                 enter   16h, 0
seg000:96D8                 push    di
seg000:96D9                 push    si
seg000:96DA                 mov     [bp+var_11], '1' ; -1nvald s3ct-id
seg000:96DE                 mov     [bp+var_10], 'n'
seg000:96E2                 mov     [bp+var_F], 'v'
seg000:96E6                 mov     [bp+var_E], 'a'
seg000:96EA                 mov     [bp+var_D], 'l'
seg000:96EE                 mov     [bp+var_B], 'd'
seg000:96F2                 mov     [bp+var_A], ' '
seg000:96F6                 mov     [bp+var_9], 's'
seg000:96FA                 mov     [bp+var_8], '3'
seg000:96FE                 mov     [bp+var_7], 'c'
seg000:9702                 mov     [bp+var_6], 't'
seg000:9706                 mov     al, '-'
seg000:9708                 mov     [bp+var_12], al
seg000:970B                 mov     [bp+var_5], al
seg000:970E                 mov     al, 'i'
seg000:9710                 mov     [bp+var_C], al
seg000:9713                 mov     [bp+var_4], al
seg000:9716                 mov     [bp+var_3], 'd'
seg000:971A                 xor     di, di
```

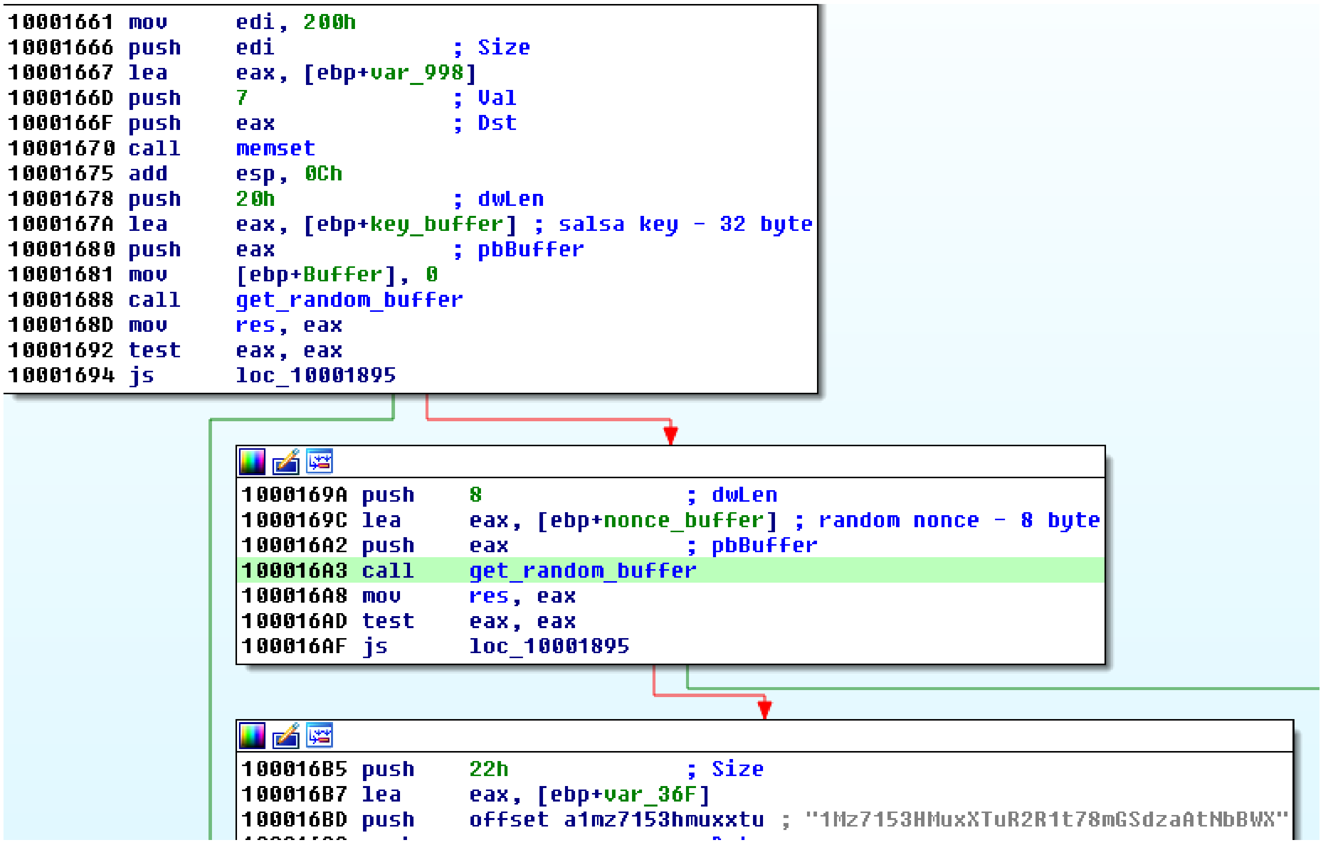
And this is a corresponding fragment from Goldeneye:

```
seg000:96D4                 enter   16h, 0
seg000:96D8                 push    di
seg000:96D9                 push    si
seg000:96DA                 mov     [bp+var_11], 'x'
seg000:96DE                 mov     [bp+var_10], 'p'
seg000:96E2                 mov     [bp+var_F], 'a'
seg000:96E6                 mov     [bp+var_E], 'n'
seg000:96EA                 mov     [bp+var_D], 'd'
seg000:96EE                 mov     [bp+var_B], '3'
seg000:96F2                 mov     [bp+var_A], '2'
seg000:96F6                 mov     [bp+var_9], '-'
seg000:96FA                 mov     [bp+var_8], 'b'
seg000:96FE                 mov     [bp+var_7], 'y'
seg000:9702                 mov     [bp+var_6], 't'
seg000:9706                 mov     al, 'e'
seg000:9708                 mov     [bp+var_12], al
seg000:970B                 mov     [bp+var_5], al
seg000:970E                 mov     al, ' '
seg000:9710                 mov     [bp+var_C], al
seg000:9713                 mov     [bp+var_4], al
seg000:9716                 mov     [bp+var_3], 'k'
seg000:971A                 xor     di, di
```

Instead of the keyword typical for Salsa20 ("*expand32-byte k*") we've got something custom: "*-1nvald s3ct-id*" (that can be interpreted as: "invalid sector id"). As we confirmed, the change of this keyword does not affect the strength of the crypto. However, it may be treated as a message about the real intentions of the attackers.

How is the Salsa key generated?

Generating the Salsa key and the nonce, as before, is done by the PE file (in the higher level of the infector), inside the function that is preparing the stub to be written on the disk beginning.



In all versions of Petya, a secure random generator was used. We can find it in the current version as well—it uses *CryptGenRandom.*

```
int __stdcall get_random_buffer(BYTE *buffer, DWORD dwLen)
{
  int v2; // eax@2
  int v3; // eax@6
  HCRYPTPROV phProv; // [sp+Ch] [bp-4h]@1

  phProv = 0;
  if ( CryptAcquireContextA(&phProv, 0, 0, 1u, 0xF0000000) )
    goto LABEL_14;
  v2 = GetLastError();
  if ( v2 > 0 )
    v2 = (unsigned __int16)v2 | 0x80070000;
  res = v2;
  if ( v2 >= 0 )
  {
LABEL_14:
    if ( !CryptGenRandom(phProv, dwLen, buffer) )
    {
      v3 = GetLastError();
      if ( v3 > 0 )
        v3 = (unsigned __int16)v3 | 0x80070000;
      res = v3;
    }
  }
  if ( phProv )
    CryptReleaseContext(phProv, 0);
  return res;
}
```

The generated Salsa key and nonce are stored in the dedicated sector for further use by the kernel during encryption.

Example of the stored data:

is_encrypted?

```
00003FF0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00004000  00 3D FE F2 0D 72 92 CC 5E 6F 01 15 78 93 07 00  .=ţň.r’Ě^o..x“..   Sector 32
00004010  3E 61 92 68 A8 EF 91 AD 10 7B CF 19 0A 7C C5 33  >a’h¨ď‘..{Ď..|Ĺ3
00004020  E0 E1 02 71 42 E4 09 F8 05 31 4D 7A 37 31 35 33  ŕá.qBä.ř.1Mz7153
00004030  48 4D 75 78 58 54 75 52 32 52 31 74 37 38 6D 47  HMuxXTuR2R1t78mG
00004040  53 64 7A 61 41 74 4E 62 42 57 58 00 00 00 00 00  SdzaAtNbBWX.....
00004050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00004060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00004070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00004080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00004090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000040A0  00 00 00 00 00 00 00 00 00 71 56 62 6E 64 42 70  .........qVbndBp
000040B0  36 57 59 73 6B 52 4A 5A 4A 35 51 53 51 34 6E 41  6WYskRJZJ5QSQ4nA
000040C0  51 53 38 6F 6D 51 79 4D 33 7A 4A 4C 64 4D 48 58  QS8omQyM3zJLdMHX
000040D0  68 41 63 51 50 68 44 58 55 76 51 70 53 58 34 5A  hAcQPhDXUvQpSX4Z
000040E0  33 52 66 67 77 00 00 00 00 00 00 00 00 00 00 00  3Rfgw...........
```

salsa key
nonce
bitcoin address
victim ID

The byte at the offset 0x4000 is the flag: 0 means that the disk is not encrypted yet, 1 means encrypted.

From the offset 0x4001, the Salsa20 key starts. It is 32 bytes long. After that, at offset 0x4021 there is the random Salsa20 nonce.

What happens with the Salsa key after the encryption?

After being read and used for the encrypting algorithm, the stored Salsa key is erased from the disk. You can see the comparison of the disk image before and after the encryption phase.

```
3FF8: 00 00 00 00 00 00 00 00 |........
4000: 00 3D FE F2 0D 72 92 CC |.=ţň.r’Ě
4008: 5E 6F 01 15 78 93 07 0C |^o..x“..
4010: 3E 61 92 68 A8 EF 91 AD |>a’h¨ď‘-
4018: 10 7B CF 19 0A 7C C5 33 |.{Ď..|Ĺ3
4020: E0 E1 02 71 42 E4 09 F8 |ŕá.qBä.ř
4028: 05 31 4D 7A 37 31 35 33 |.1Mz7153
4030: 48 4D 75 78 58 54 75 52 |HMuxXTuR
4038: 32 52 31 74 37 38 6D 47 |2R1t78mG
4040: 53 64 7A 61 41 74 4E 62 |SdzaAtNb
4048: 42 57 58 00 00 00 00 00 |BWX.....
4050: 00 00 00 00 00 00 00 00 |........
```

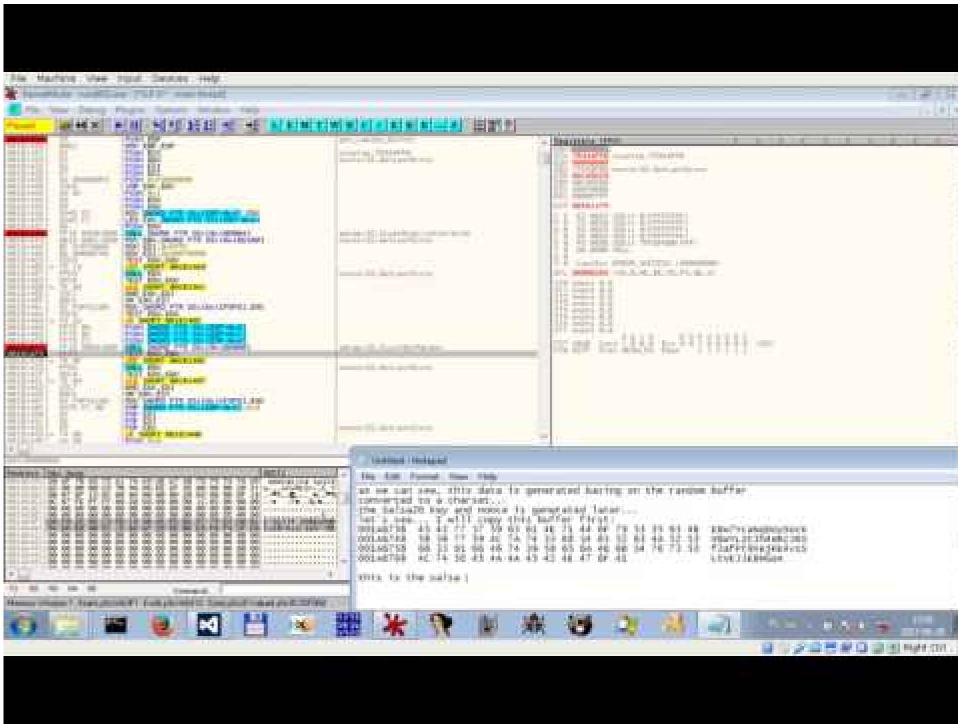
```
3FF8: 00 00 00 00 00 00 00 00 |........
4000: 01 00 00 00 00 00 00 00 |........
4008: 00 00 00 00 00 00 00 00 |........
4010: 00 00 00 00 00 00 00 00 |........
4018: 00 00 00 00 00 00 00 00 |........
4020: 00 E1 02 71 42 E4 09 F8 |.á.qBä.ř
4028: 05 31 4D 7A 37 31 35 33 |.1Mz7153
4030: 48 4D 75 78 58 54 75 52 |HMuxXTuR
4038: 32 52 31 74 37 38 6D 47 |2R1t78mG
4040: 53 64 7A 61 41 74 4E 62 |SdzaAtNb
4048: 42 57 58 00 00 00 00 00 |BWX.....
4050: 00 00 00 00 00 00 00 00 |........
```

As you can see, after use the key is erased.

What is the relationship between the victim ID and the Salsa key?

In the previous versions of Petya, the victim ID was, in fact, the victim’s Salsa20 key, encrypted with the attacker’s public key and converted to Base58 string. So, although the Salsa key is erased from the disk, a backup was still there, accessible only to the attackers, who had the private key to decrypt it.

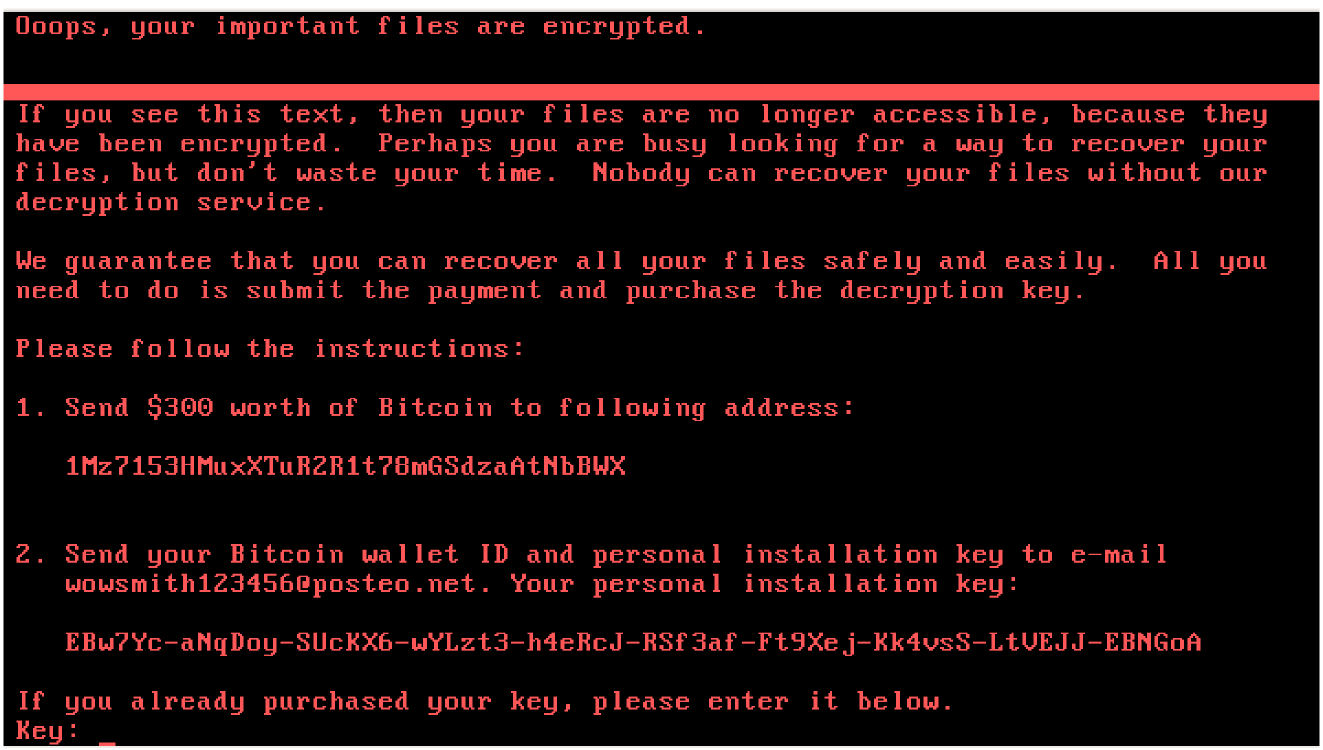
Now, it is no longer true. The victim ID is generated randomly, BEFORE the random Salsa key is even made. So, in the current version, the relationship of the Salsa key and the victim ID is none. The victim ID is just trash. You can see the process of generating it on the video.



Watch Video At:

https://youtu.be/LS0nWpRfVs8

After the reboot from the infected disk, we can confirm that the random string generated before Salsa key and nonce is the same as the one displayed on the screen as the victim ID (“personal installation key”):



Conclusion

According to our current knowledge, the malware is intentionally corrupt in a way that the Salsa key was never meant to be restored. Nevertheless, it is still effective in making people pay ransom. We have observed that new payments are being made to the bitcoin account. You can see the link to the bitcoin address here: https://blockchain.info/address/1Mz7153HMuxXTuR2R1t78mGSdzaAtNbBWX

If you are a victim of this malware and you are thinking about paying the ransom, we warn you: Don't do this. It is a scam and you will most probably never get your data back.

We will keep you posted with the updates about our findings.

Appendix

Microsoft's report about the new version of Petya

About the original version (Goldeneye):

> Goldeneye Ransomware – the Petya/Mischa combo rebranded

This video cannot be displayed because your *Functional Cookies* are currently disabled. To enable them, please visit our *privacy policy* and search for the Cookies section. Select *"Click Here"* to open the Privacy Preference Center and select *"Functional Cookies"* in the menu. You can switch the tab back to *"Active"* or disable by moving the tab to *"Inactive."* Click *"Save Settings."*

This was a guest post written by Hasherezade, an independent researcher and programmer with a strong interest in InfoSec. She loves going in details about malware and sharing threat information with the community. Check her out on Twitter @hasherezade and her personal blog: https://hshrzd.wordpress.com.