Breaking Petya - Solving a Poor Implementation of Salsa

Peixian Wang
May 10, 2016

Abstract

Ransomware has become a relatively profitable development in recent years with the surge in popularity of Bitcoin and other untraceable forms of money transactions. In this paper we detail Petya, a recent form of ransomware targeting Windows platforms and NTFS drives. We describe the construction of Petya and the underlying encryption algorithm, Salsa20, and also present one possible solution called Infestor, which utilizes Z3, an efficient satisfiability modulo theory solver, to defeat Petya.

1 Introduction

Online transactions through Tor (4) using anonymous cryptocurrencies allow for a certain level of privacy when it comes to payments, but also allow themselves to be used in a malicious fashion. Since Bitcoin, the most popular form of cryptocurrency, makes it extremely hard to track the transactions, Bitcoin has become the de facto standard for ransomware, malware which infects the victim's computer, holds files hostage through encryption, and extorts the victims for money in return for the files. Petya is one such ransomware, except rather than targeting the files of the victim, it targets the master boot record (MBR) and master file table (MFT) (9).

2 Petya

2.1 Overview

Petya is a relatively new ransomware variant, only starting to appear within the early months of 2016. In order to bypass the lengthy process of encrypting each file on the victim's hard drive, Petya simply seeks to write malicious code to the start of the disk. This code overwrites the MBR of the hard drive with a small kernel that then encrypts the MFT.

2.2 Behavioral Analysis

Petya is usually distributed through a zip file (3), containing two other files: 1. a photo of a young man, purporting to be an applicant and 2. an executable, disguised as a csv file, shown in figure 1. After opening the executable, Petya calls an undocumented API called `NtRaiseHardError`. The computer then promptly crashes and boots into a fake CHKDSK scan, shown in figure 2, which starts

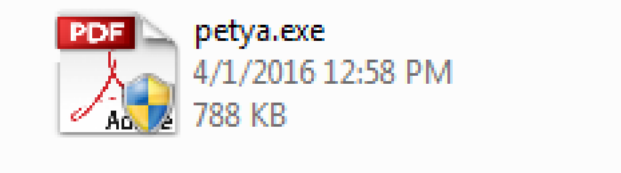

Figure 1: The petya executable disguised as a .pdf file.

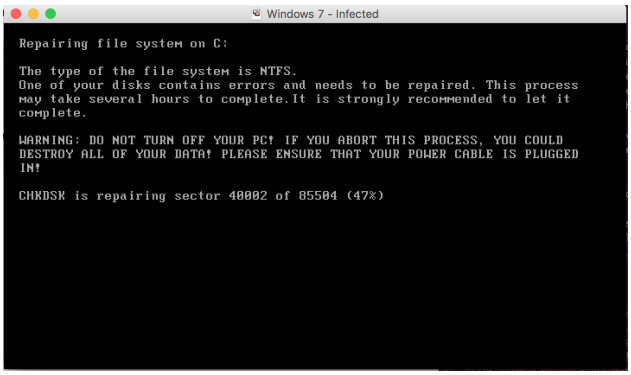

Figure 2: The fake CHKDSK scan made by Petya while it encrypts the MFT.

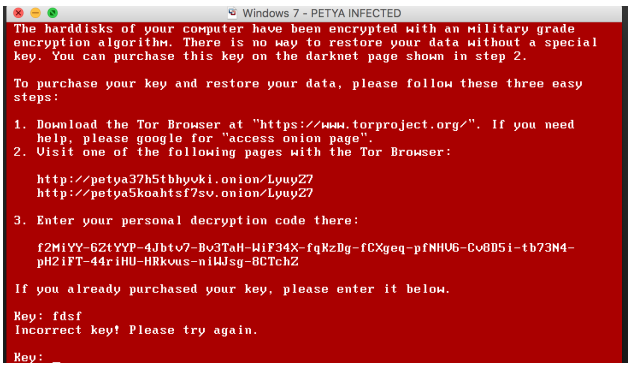

Figure 3: The ransom note by Petya.

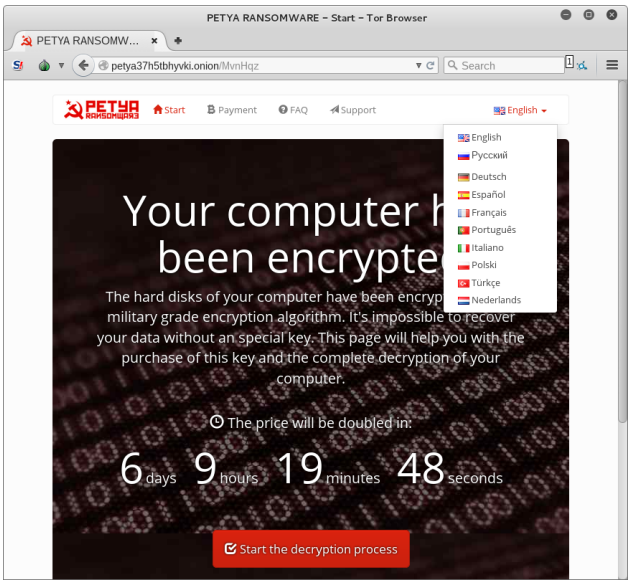

Figure 4: Accessibility is important, even for malware. From (3).

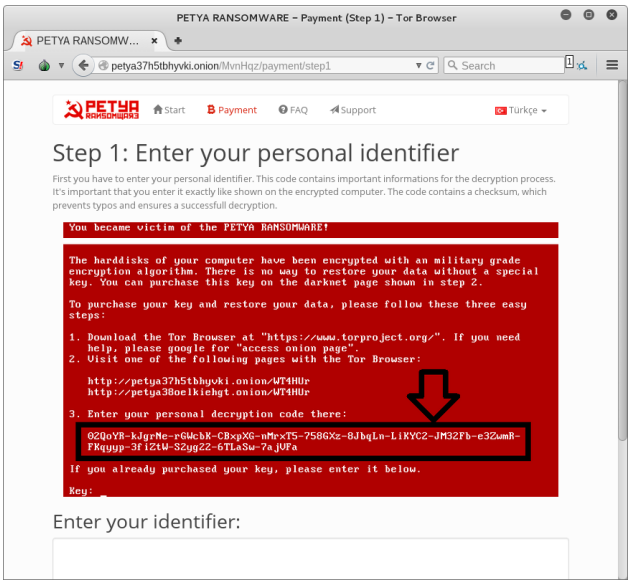

Figure 5: Petya Tutorial on purchasing bitcoins and performing a transaction. From (3).

```
00000000 ONION_SECTOR struc      ; (sizeof=0x200)
00000000 eEncrypted db ?         ; is drive encrypted flag, enum ( NotEncrypted = 0,
                                 ; Encrypted = 1, Decrypted = 2 )
00000001 key db 32 dup(?)        ; ENCRYPTION KEY
00000021 iv db 8 dup (?)         ; IV (stays permanent in the system)
00000029 szURLs db 128 dup(?)    ; onion links
000000A9 szPubKey db 343 dup(?) ; public key string, will be displayed at ransom page
00000200 ONION_SECTOR ends
```

Figure 6: The `ONION_SECTOR` struct, from (9).

the encryption on the MFT. When the encryption completes, the user is shown a ransom note screen, shown in figure 3. After visiting the website, the user is presented with a relatively upscale website, featuring multiple languages (figure 4) and a tutorial on how victims can perform a bitcoin transaction (figure 5).

2.3 Code Analysis

The Petya attack can be detailed into 3 stages, the MBR attack stage, the MFT attack stage, and the ransom demand stage.

2.3.1 Stage 0 - MBR attack

In order to start analysis on Petya, the Windows executable must be examined. The executable generates a 8-byte initialization vector and a unique 16-byte random key that is used for further encryption, which also must be known to attackers to decrypt the files (3). This key is expanded into the 32-byte encryption key using figure 7. Both of these values are used later on in the encryption process (9).

Petya then encrypts the original MBR by XOR'ing the contents of the MBR with `0x37` (7). The encrypted value is then saved to sector 56 in the disk (8). Disk sector 1-34 is encrypted in the same method. Petya then proceeds to write its own kernel, creating an `ONION_SECTOR` structure and writing it to sector 54, shown in figure 6. This data conveniently contains 8 byte `iv`, which is the initialization vector required later on to decrypt. Sector 55 is then filled with the byte `0x37`, which is checked after the decryption process for validation purposes.

Finally, the `NtRaiseHardError` is called, crashing the victim's system.

2.3.2 Stage 1

The system drive is queried by the Petya kernel, and upon returning a succesful value, Petya begins to overwrite the MFT while masquerading as a CHKDSK (8).

This stage can be divided into two steps: reading the key and deletion of the key, and encrypting sector 55.

In the first step, Petya reads sector 54 into memory, which currently contains the `ONION_SECTOR`. It copies the first 32 bytes of the `ONION_SECTOR` into memory, which is the encryption key, and then proceeds to zero the first 32 bytes of data. Afterwards, the 32-byte key is only present in memory.

The second step consists of Petya encrypting sec-

```
void key_expand(char key[16], char outKey[32])
{
        for (int i = 0; i < 16; ++i) {
                unsigned char uc = key[i];
                outKey[i * 2 + 0] = uc + 0x7A; // uc + "z"
        }

}
```

Figure 7: The key expansion method called by Petya. From (9).

tor 55, which is currently filled will all byte values of `0x37`. In this step Petya creates a master table of 64 bytes: a 32 byte key, 8 byte vector, and 8 byte value of 0. This table is added and rotated until a 16 DWORD is obtained, with the values of `(LOW_WORD, HIGH_WORD, 0, 0)`. This is Petya using a flawed version of Salsa20 (2), passing in the 32-byte encryption key and the 8-byte initialization vector, to encrypt sector 55, which we will detail in section 3.

2.3.3 Stage 2

Finally, Petya simply reboots into the red flashing skull screen, displaying the ransom note and providing an entry for the decryption key. By running the infected disk through a debugger, we can follow the verification process of the key:

1. Read sector 54 to read the 8 byte initialization vector.
2. Read sector 55.
3. Read the user entered key, extract the first 16 bytes, and use the same expansion function as 7 in order to retrieve a 32-byte key.
4. Use the 32 byte key and the 8 byte initialization vector to decrypt sector 55, if all the values equal `0x37`, then the user entered key is correct. Else throw an `Invalid Key!`.
5. If the user entered key is correct, proceed with the decryption process and ask user to restart their computer.

From this verification process, we can begin to construct a list of constraints the key must follow:

1. Only the first 16 bytes are extracted from the user entry, meaning the key must be at most 16 bytes long from the character set `[a-zA-Z0-9]`, i.e. all alphanumerical characters.
2. The key must enable all data in sector 55 to be able to xor with `0x37`.
3. The initialization vector must be used shuffled in somehow.

Thus, in order to break Petya, we must start by breaking its encryption algorithm: Salsa.

3 Salsa

3.1 Salsa20 Spec

Salsa20, at its core, is a function that takes a 64-byte string and produces a 64-byte string. The function follows an add-rotate-xor operation, whose operation can be found in exact detail in the Salsa20 spec (2), but the original implementation uses a 32-byte encryption key and an 8-byte initialization vector to produce a final 515-bit key-stream, shown in figure 8.

To create the output, the 64 byte input into Salsa is seen in little-endian form as 16 words, which are fed into 320 invertible modifications, where each modification changes one words. The resulting 16 words are added to the original input respectively mod 2^{32}, which then produces the output in little-endian form. The exact modification is predefined in the Salsa spec and implementations (10), which is a series of 10 identical double-rounds, each consisting of 4 parallel quarter-rounds, with each quarter-round modifying 4 words (1).

3.2 Petya Salsa

Fortunately, the implementation of Salsa within Petya is severely flawed, allowing us to easily break it (9).

3.2.1 Flaw 1

The original Salsa utilizes a 64-bit value (10), but Petya uses a 32-bit value, resulting in the high part

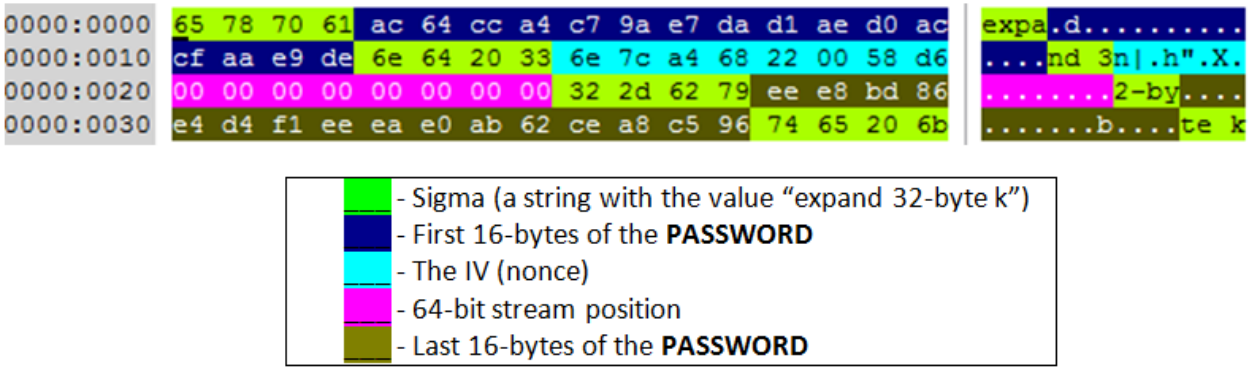

Figure 8: The original Salsa implementation. From (9).

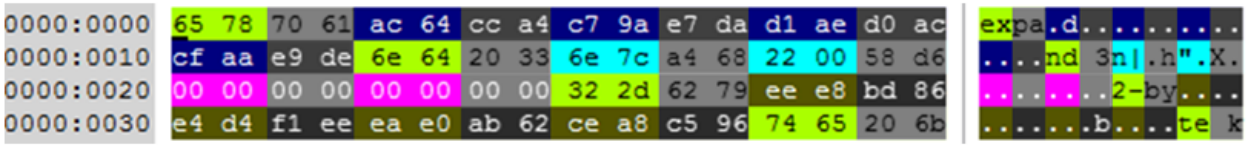

Figure 9: The flawed Salsa implementation, bytes in gray are unused values. From (9).

of the key being a constant value of 0.

3.2.2 Flaw 2

Salsa20 performs a rotate left operation utilizing unsigned 32 bit integers (`uint32_t`) (10) with a constant shift left of value 32, but Petya does this with 16 bit integers. This results in a far more predictable result.

3.2.3 Flaw 3

Salsa20 uses two internal 32 bit vectors, but Petya uses two internal 16 bit vectors. This results in the little-endian revolution resulting in half of the revolutions becoming 0, meaning that half of the key is constant and predictable, shown in figure 9. The bytes in gray represent unused values.

4 Infestor

With the various flaws in Petya's implementation of Salsa and the constraints based around the keys, we can begin to construct a constraint solver for Petya. Many of the values in the constraint solver were derived from a genetic solver written by Leo Stone (6), who had many constants already discovered.

4.1 Code Walkthrough

We begin solving with the `infestor.py` file, found on `https://github.com/peixian/Infestor/blob/master/infestor.py`.

We first define the steps Salsa takes, which are already defined in the Salsa20 core (1). Next, we define constants, including the target byte value of `0x37` for us to xor to, along with `STREAMHIGH` and `STREAMLOW`, which are a part of the 16 DWORD obtained in 2.3.2. We can then start to reconstruct the initial table referenced in the Salsa20 core (1). We use the constant values taken from the genetic solver (6), and then initialize it with constants and the key bytes. We can construct reconstruct the key expansion process with the `z3.ZeroExt` (5), which is always constant in figure 7.

Next, we define the various Salsa functions, which are simply taken from the Salsa implementation (10).

Afterwards, we need to extract the 8 byte initialization vector from sector 54, stored in `nonce.txt`, which is in actuality the nonce for Salsa. We also need to extract the 512 byte cipher-text from sector 55, which is then stored in codesrc.txt. We then set up z3 as a bit vector value to find the initial matrix and source words.

In order to complete the constraints, we can take additional constraints found in the genetic solver and add it to our z3 solver. To complete our solution, we need to apply the same function the

Salsa20 core does referenced in section 3, which is to add the resulting words to the initial words, and then modulo 2^{32}. Due to the flawed implementation of Salsa by Petya, we can simply respectively modulo the source words by 2^{16}, which is our final constraints for the solver.

References

[1] Bernstein, D. J. The salsa20 core. https://cr.yp.to/salsa20.html.

[2] Bernstein, D. J. Salsa20 specification. https://cr.yp.to/snuffle/spec.pdf.

[3] hasherezade. Petya – taking ransomware to the low level. https://blog.malwarebytes.org/threat-analysis/2016/04/petya-ransomware/, 2016.

[4] Project, T. Tor project. https://www.torproject.org/.

[5] rise4fun. Getting started with z3: A guide. http://rise4fun.com/z3/tutorial.

[6] Stone, L. hack-petya mission accomplished!!! https://github.com/leo-stone/hack-petya, 2016.

[7] Tonello, G. Breaking petya ransomware! http://www.tgsoft.it/english/news_archivio_eng.asp?id=718, 2016.

[8] Tonello, G. Petya ransomware x-rayed !!! http://www.tgsoft.it/english/news_archivio_eng.asp?id=712718, 2016.

[9] Trafimchuk, A. Decrypting the petya ransomware. http://blog.checkpoint.com/2016/04/11/decrypting-the-petya-ransomware/, 2016.

[10] Weber, A. Salsa20. https://github.com/alexwebr/salsa20/blob/master/salsa20.c, 2015.