Cracking Petya's cipher

With pen and paper

Francisco Blas Izquierdo Riera (klondike) *<klondike(a t)klondike.es>*

Abstract

In this paper, we present some techniques we applied using pen and paper to find a single-step solution requiring only one known plaintext to break the Salsa20 like cipher used by the first version of the ransomware known as Petya. Also, we explain how these techniques can be run programmatically with higher precision along with the time and memory cost of running them.

1 Introduction

In March 2016 Petya was sighted for the first time in the wild. Unlike most ransomwares Petya didn't encrypt the desired file types from userspace. Instead, it installed the encrypting payload on the MBR and triggered a reboot using a Blue Screen of Death. This encrypting payload would then be run on boot and show a fake chkdsk run whilst encrypting the MFT. The user would afterwards see a red screen explaining the ransom conditions where the encryption key could be introduced to decrypt the MFT.

Because of its innovative approach, Petya attracted a lot of attention from researchers who tried to reverse engineer Petya's code and the ciphers it used. All of these efforts crystallized first in a tool able to extract the key after the first stage of infection [1] and on detailed descriptions of Petya [2]. Despite that, the Salsa20 like algorithm used on the second stage remained unbroken until Leostone attempted, a brute force attack first and, a bit later, an attack using genetic algorithms. The code used for this last attack was soon published [3].

The fact genetic algorithms could be used to break the second stage cipher, implied there was some kind of correlation between the known plaintext and the resulting plaintext as the key approached the correct one. Despite that, doubt remained on whether the issue was caused by the way in which Salsa20 was implemented or if an unknown intrinsic flaw of Salsa20 was uncovered. Because of this, the researcher set off on cryptanalysing the cipher as implemented by Petya to uncover the flaws in it.

The choice of pen and paper techniques is deliberate. The researcher had only his free time to work on this project and, therefore, chose techniques he could work with anywhere as computer access was limited sporadically whilst the research was being carried. Despite that, we'll also expose how these techniques can be implemented in a computer program to find similar flaws on other ciphers.

1.1 Salsa20 as implemented by Petya

As Petya's encryption and decryption algorithm's run on the boot sector they were strongly limited by the available storage and by the CPU being in 16-bit real mode. For some reason, the authors behind Petya decided to use 16-bit operations on their implementation of Salsa20 (instead of using 32-bit operations as specified in [4] by prefixing the instructions with 0x66). Despite that, they used 32-bit rotations on their code as some of the rotation constants were larger than 16.

The mapping of attributes was the same as the one defined when presenting Salsa20 as a stream cipher [4] using 16-bit words for the state instead of 32-bit words. The nonce was provided by the dropper of the ransomware in a specific disk sector, the constants were cut to 16 bits, and the counter was initialized at 0. Finally, the key parts were derived from the user provided key using a custom algorithm.

The reference implementation which was used during this project can be found in [3].

1.2 Petya's key derivation algorithm

In order to map the characters of the Petya cipher to the words used by Salsa20, the authors of Petya used a custom made key expansion algorithm. This algorithm presented a flaw in that it did not use the even characters of the key.

The key derivation algorithm took the passphrase provided by the user and processed only the first of each two characters. For example for abcdef it would process only a, c and e. The character was then expanded to the 16 bits needed for a state word in the cipher using the following equation where c is the character we are currently processing:

uint16(c<<1)<<8 | uint16(c+'z')

1.3 Petya's key verification algorithm

In order to check if the key entered is valid, Petya uses its Salsa20 like cipher to decrypt a previously encrypted block. This block contains the encrypted result of the hexadecimal value 37, which is the ASCII code for 7, repeated up to a full disk-block size. If the decryption of the block results in the expected output, Petya then proceeds to decrypt the rest of the encrypted sections of the hard drive.

2 Cryptanalysis techniques

For finding the flaws in Petya's cipher an assortment of cryptanalytic techniques were used. All of which are described in the following sections.

2.1 Contribution analysis

The idea behind contribution analysis is determining if the different inputs of the algorithm contribute to the different outputs. In this aspect the objective is measuring and finding flaws in the diffusion of the cipher as those elements where an input doesn't contribute can be discarded when trying to find out that output..

Contribution analysis can be done using different accuracy levels ranging from modelling bits individually to modelling groups of related bits (for example words in Salsa20). Because of the quadratic memory cost of this algorithm and the execution cost being proportional to both the number of operations used by the cipher and the number of elements to model, the choice of larger groups of elements may seem interesting, but in doing so, resolution will be lost and results that would be uncovered by more precise analysis will be missed.

During our first try we used a Salsa20 word as the unit which showed that, all words had contributed to each other after the third Salsa20 round. This made us fall back to another technique although we discovered later that using bit level precision instead of word level would have uncovered the flaw as well.

For this technique a table with as many rows as input bits (or groups of them) and as many columns as output bits is created with additional columns for intermediate values on each step. Afterwards, each step of the cipher is analysed crossing, on each column, all the rows whose bit has contributed to that intermediate state.

For example, using words instead of single bits for grouping, the first column corresponding to the new value of word number 4 would have crossed the cells for rows 0, 4 and 12. This process would be repeated with a new column for each further step of the algorithm until finalization from where the columns corresponding to outputs would be extracted. In the cases where the algorithm makes use of intermediate states we will cross all the rows that are already crossed on that intermediate state in the column being processed. For example on the next step where word 8 receives contributions from words 0 and the new status of word 4 we will cross all words crossed in the previous round (0,4,12) then cross word 0 (which is already crossed) and then word 8.

Certain state based ciphers, like Salsa20, mix back into the current state a transformation of parts of the previous state instead of replacing it. For such ciphers, it is possible to optimize the algorithm by reusing the output cells instead. This results in a smaller amount of cells.

Another approach can use lists of elements instead of fixed size arrays but if the cipher is strong this will result in lists with all of the elements resulting in a more complex and error prone algorithm.

2.2 Unmodified bits analysis

This method tries to track those parts of the output that remain unmodified and thus are affected by a single bit of the input and, as a result of this analysing the confusion of the cipher. This method was used after the failure of the previous method and uncovered the extent of the rotation issues.

Like the previous technique, bits can be grouped resulting in faster executions at the cost of precision. Although this will not be as helpful as with the other algorithm as the memory requirement is lineal and proportional to the bits of output to track. Despite that, even with such groupings, the execution time is still proportional to both the number of groups of bits to track and the number of steps used by the cipher.

Word	Use	Unmodified bits
0	Const0	2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 , 11 , 12 , 13 , 14 , 15
1	Key0	0 , 1 , 2 , 3 , 4 , 5 , 6
2	Key2	0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8
3	Key4	0 , 1 , 2 , 3 , 4 , 5 , 6
4	Key6	0 , 1 , 2 , 3 , 4 , 5 , 6
5	Const2	2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 , 11 , 12 , 13 , 14 , 15
6	Nonce0	0 , 1 , 2 , 3 , 4 , 5 , 6
7	Nonce2	0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8
8	Counter_LSB	0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8
9	Counter_MSB	0 , 1 , 2 , 3 , 4 , 5 , 6
10	Const4	2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 , 11 , 12 , 13 , 14 , 15
11	Key8	0 , 1 , 2 , 3 , 4 , 5 , 6
12	Key10	0 , 1 , 2 , 3 , 4 , 5 , 6
13	Key12	0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8
14	Key14	0 , 1 , 2 , 3 , 4 , 5 , 6
15	Const6	2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 , 11 , 12 , 13 , 14 , 15

Table 1: Unmodified bits for each of the output words of Petya's cipher.

This technique can be implemented by setting up a succession of cells each one representing the output bits and intermediate states. In these cells the empty value represents that no processing has been made yet, a single annotation represents the single bit that contributed to that cell whilst a crossed annotation represents that more than a single bit contributed to that cell.

To use this technique we then run each step of the algorithm for each empty cell adding the first contribution to that output. Instead, if any other bit is found to contribute to that state the cell is crossed to mark that more than a contributor participate.

Like in the previous algorithm, this algorithm can be optimized to drop away any intermediate values, using only a single cell for each output state value and initializing them with the corresponding input state. Such optimization can only be carried in ciphers where a transformation of parts of the state is mixed back into the current state as done by Salsa20.

For example, in the case of Petya, we will start with a single cell for each bit of the cipher state and then initialize them with the bit of the state contained initially. After the first sixteenth of a round, we will cross out bits 7 to 15 of word 4 as they’ll receive contributions from the bits in words 0 and 12. Then for the next sixteenth of a round we will cross bits 9 to 15 of word 8 and so on. Eventually the only bits that will remain uncrossed are those receiving less than two contributions. Keep in mind that any contributions whose values are known (for example from constants) are not counted in this algorithm.

In the specific case of Petya this algorithm uncovered the issue with the 32-bit rotations resulting in the listings of unmodified bits shown in Table 1.

2.3 Algebraic modelling

Since Salsa20 is an ARX cipher, this was the first method considered. The main reason for the choice was that this method would provide a set of boolean equations that, once solved, could be reused as needed to solve all instances of the cipher independently of the key used. We started writing such equations modelling:

- Additions using the boolean logic of the equivalent adder circuit
- Rotations as a remapping of the different bits
- XORs as the boolean XOR of all the bits

The complexity of this method, showed up soon and we discarded it in favour of other approaches.

Once the 32-bit rotation issues had shown up we decided to apply this method only to the weak parts of the cipher with the objective of finding a method to obtain the key using a known plaintext attack. For this we only focused on the weak parts of the state containing key parts and used knowledge of the key space (in particular that valid keys contained characters without the MSB set). Using this we then could model the output of the least significant bits of each word as the following equation, where w is the corresponding bits of the word in the known ciphertext and c is the corresponding character of the key used in that word:

$w = (c + 'z') \oplus '7'$

Which using 7 bit unsigned integers we resolve as:

$c = (w \oplus '7') - 'z'$

Resulting in a simple equation that can be used to find each character of the key.

3 Cryptanalysis results

Although we have managed to find a simple equation to recover the key bits to decrypt the second stage of Petya, it is quite clear that Salsa20 should not be affected by the same cryptographic issues as Petya's Salsa20 like cipher.

On one side, had the key mixing issue, which dropped the even characters of the key, not been there; the best attack that would have been possible using the rotation issue would have been an informed brute force attack. Such attack would be carried either against 9 or 7 of each 16 bits of the key had a proper KDF been in place or against approximately 23 or 6 characters of the input set if mapping the password characters from the key words was easy. These attacks would have a complexity of approximately $2^{24.64}$ encryptions.

On the other hand, had only the character mixing issue been in place, the best possible attack would have been a brute force attack against a key using only the odd characters of the key. This was the first approach tested by Leostone before he attempted the genetic approach. This attack has a complexity of approximately $2^{46.04}$ encryptions.

It becomes clear now that neither of the issues found affect Salsa20. Salsa20 uses 32-bit words everywhere and is thus not affected by the rotated 0s inserted when casting the value to a 32-bit integer. Also, Salsa20 is agnostic to the way keys should be mapped to words in the state thus making the weak key issue an implementation specific issue.

In light of these results it is clear that it was the mix of both flaws, the key mapping algorithm used by Petya and the issue with the 32-bit rotations, that allowed the genetic algorithms applied by Leostone to succeed.

4 Computer implementation of the techniques

4.1 Contribution analysis

The following pseudocode can be used:

Let I be a list of inputs, M a list of the intermediate states, O a list of outputs and S an ordered (by step) list of cipher steps mapping a set of element from I+M to a single element in M+O
Create a bit array A with indexes I+M+O and I
For all i,j in I+M+O,I set A[i][j]=False
For all i in I set A[i][i]=True
For all IS, m in S: For all i in IS: For all j in I: A[m][j] = A[m][j] or A[i][j]
Return i,A[i] For all i in O

The difficult part of this algorithm is preparing S. For example, in the case of Salsa20, we will need an intermediate state for each operation result, hence we will need 32 (one per bit) for each addition, 32 for each rotation and 32 for each XOR. This can be optimized for this particular cipher by using more than two inputs carefully so that only 32 states are needed for each quarter round. This is a bit more complex though as MSB to LSB processing of the bits of the sum needs to be guaranteed when running the algorithm to prevent the introduction of false dependencies (as the MSB will also depend on the LSBs because of the addition's carry).

Like in the pen and paper version, this can be memory optimized when working on a state which is updated instead of replaced, as is the case of Salsa20, as follows:

Let I be a list of state parts and S an ordered (by step) list of cipher steps mapping a set of element from I to a single element in I
Create a bit array A with indexes I and I
For all i,j in I,I set A[i][j]=(i == j)
For all IS, m in S: For all i in IS: For all j in I: A[m][j] = A[m][j] or A[i][j]
Return i,A[i] For all i in I

In this second case care must also be taken that no cross dependences appear accidentally (for example, adds must be processed from the MSB to the LSB). This noticeably increases the difficulty of preparing S from the cipher specification as, in the case of Salsa20 it will also need to process sums from MSB to LSB.

4.2 Unmodified bit analysis

This analysis can be implemented using the following pseudocode:

Let I be a list of inputs, M a list of the intermediate states, O a list of outputs and S an ordered (by step) list of cipher steps mapping a set of element from I+M to a single element in M+O
Create a array A able to contain an element from I+None+Many with index I+M+O
For all i in M+O set A[i]=None
For all i in I set A[i]=i
For all IS, m in S: For all i in IS:
    If A[m] == None: A[m] = A[i]
    Else If A[m] != A[i]: A[m] = Many
Return i,A[i] For all i in O

As for the previous case, this can be memory optimized when working on a cipher whose state is updated instead of replaced, as is the case of Salsa20, as follows:

Let I be a list of state parts and S an ordered (by step) list of cipher steps mapping a set of element from I+M to a single element in M+O
Create a array A able to contain an element from I+None+Many with index I
For all i in I set A[i]=i
For all IS, m in S: For all i in IS:
    If A[m] == None: A[m] = A[i]
    Else If A[i] != None and A[m] != A[i]: A[m] = Many
Return i,A[i] For all i in O

This algorithm is also trickier in that back dependencies have to be taken care of beforehand as otherwise you may get bits marked incorrectly. In any case for the real Salsa-20 implementation, not Petya's, the output of this algorithm should be Many for each bit of the state.

5 Conclusions

We have documented the techniques used to uncover the 32-bit rotate issues using pen and paper. Also, we have proven why it was possible to use genetic algorithms to break Petya's cipher by combining the key generation issue with the 32-bit rotation issue. We have derived an equation that would achieve the same result in one step with a single plaintext. We have also shown that such issues should not happen in a correct implementation of Salsa20 and thus Salsa20 is not affected by them. Finally, we have provided the ways to implement programmatically some of the cryptanalytic techniques used.

Acknowledgements

Despite the author carried this research out independently, he was supported by other people. Thus, the author would like to thank:

- Leostone for his amazing work reverse engineering Petya and for disclosing the code to the author used to carry out the cryptanalysis.
- My parents for being supportive and having motivated my curiosity and my interests during my whole life.
- The SEC-T organizers for providing an open forum to share my research.
- Huzaifa Essajee for his incredibly useful comments on this paper and the talk.
- Hasherezade for her amazing work analysing and tracking Petya and her input which was used on this paper and the talk.
- Tero Hänninen for his moral support and listening to my strange comments about key entropy and boolean equations whilst I was working on this project. Also for many comments that have helped make this paper and prepare the talk.
- Meredith L. Pattersson for her input which helped shape this paper and the talk, and for her great advice.
- Niklas Andersson and Mikael Johannson for their input regarding the talk.
- Coresec Systems AB for giving me a place to rehearse my talk and being flexible so that I can combine my full time job with my private research projects.

The author would also like to make a note in memory of his late father Francisco Ángel Izquierdo Silvestre who died on 25 October 2015. He has always been an inspiration to the author and without all the support the author received during his whole life from his father, nor this research, nor this paper, nor the associated talk would have ever happened.

References

[1] 2016-03-31, Hasherezade, Petya key decoder, https://hshrzd.wordpress.com/2016/03/31/petya-key-decoder/
[2] 2016-04-01, Hasherezade, Petya – taking ransomware to the low level, https://blog.malwarebytes.com/threat-analysis/2016/04/petya-ransomware/
[3] 2016-04-09, Leostone, hack-petya: genetic, https://github.com/leo-stone/hack-petya/commit/1d42cebbef45235ba615d848c07f2be69fa8e97e
[4] 2007-12-25 Daniel J. Bernstein, The Salsa20 family of stream ciphers https://cr.yp.to/snuffle/salsafamily-20071225.pdf