# Randomware
## Les aléas d'un ransomware

Yoann Guillot
yoann.guillot@ssi.gouv.fr

ANSSI/SDO/DR/IPC

**Abstract.** In this article, we propose to share the methodology we used to recover most of the files encrypted by a ransomware where the encryption process was flawed by a very poor randomness source: time. The flaw itself is pretty trivial, but the process of recovering a large number of files by exploiting that weakness leaves a lot of freedom, which can have a big influence on the efficiency and therefore the time required to restore the files.

# 1 The malware

## 1.1 A ransomware attack

Ransomwares are a form of computer crime where an attacker takes control of a victim's network and uses that access to deploy a malware that will encrypt most of the files on the infected computers using strong cryptography. The attacker then asks for a ransom, usually through an anonymous payment method using some cryptocurrency, in exchange for the decryption software that will supposedly restore the victim's files.

Note that this attack was an artisanal process, where an attacking team progressed manually in the network to identify key servers (e.g. backup servers) and nonstandard software in use. This way, when they launched the attack, they destroyed the backups and killed the programs used by the victim's business applications so that the malware could encrypt the files used by those, in order to achieve maximum impact.

## 1.2 Encrypting files

Here, the malware works as follow: it lists all the drives of the machine, and crawls through every subdirectory. In each one, the files are listed, and the ones not matching a file extension blacklist are encrypted. The actual file encryption is done inside a thread pool, with each thread handling one file, then moving on to the next file in the queue.

### 1.3 Encrypting one file

To encrypt one file, the malware generates a random buffer of 117 bytes using a Mersenne Twister PRNG paired with an SBox, and uses that buffer as a key to initialize an RC4 stream cipher. Depending on the original file size, either the first 1Mo or 10Mo of the file contents are encrypted in-place using this cipher.

The file-specific RC4 key is then encrypted using an RSA public key embedded in the malware, and this encrypted blob is appended at the end of the file. The RSA encryption is performed using the Windows CryptoAPI, but the RC4 encryption is done directly in the malware binary.

This way, by knowing the private RSA key, the attacker is able to recover the RC4 key and he can restore the file contents.

### 1.4 Cryptographic flaw

The weakness here is that the file-specific RC4 key is generated from a Mersenne Twister pseudo-random number generator seeded using only the value of the GetTickCount() Windows API.

The GetTickCount() API is a standard function that returns the number of milliseconds elapsed since the boot of the machine as a 32-bit number. It has a resolution of around 15 milliseconds usually.

The Mersenne Twister PRNG is fully deterministic for a given seed value.

To generate one RC4 key, the function GetRandByte [1] is called 117 times in a row.

```
1: prevtick: global variable holding the last tick value
2: mt: global variable holding a Mersenne Twister state
3: SBox: a static substitution box, returns a byte
4: function GetRandByte
5:     curtick ← GetTickCount()
6:     if curtick ≠ prevtick then
7:         prevtick ← curtick
8:         mt ← NewMersenneTwister(curtick)
9:     return SBox[MersenneGetValueRange(mt, 0, 257)]
```

**Algorithm 1.** GetRandByte

1. The SBox as used by the malware takes an integer in the range 0 to 257 and returns a byte, due to an off-by-one error in the malware code

The time taken to generate one RC4 key, even on a busy computer, is much less than 15ms, so we can expect that one RC4 key is usually generated without reseeding the Mersenne Twister PRNG.

So the attacker expected to have a 117-byte (936-bit) random key, but he actually has a buffer fully determined by a not so random 32-bit value.

Knowing this, in theory we can try all $2^{32}$ possible tick values, use them to seed a PRNG, generate the matching RC4 keys, and one of those will be the same as that generated by the malware ; we can then proceed to decrypt the file.

# 2 Decrypting files

## 2.1 Identifying decrypted files

The first issue is that we have a large number of keys to try in order to decrypt one file. But for each key, how do we know if we succeeded in decrypting the file ?

The ideal solution would be to compare the RC4 key we generated to the one generated by the malware. Unfortunately, if we encrypt our key using the malware RSA public key, we cannot compare with the encrypted key blob stored at the end of the file. The Windows CryptoAPI uses the OAEP RSA padding scheme which includes a (strongly) random 64-bit value, therefore two RSA encryptions of the same RC4 key with the same RSA public key will create two distinct encrypted blobs.

The only other option is to check some properties of the clear-text obtained with each candidate key. We chose to use an entropy test on the first 256 bytes of the decrypted data, coupled with checking a few well-known file signature magic patterns for the most common files in our set.

This is a very real limitation: if we cannot identify a clear text pattern, we may very well try the correct RC4 key but fail to recognize it. It is therefore crucial to regularly check on the files, and add patterns to match cleartext data for files we strongly suspect should be decrypted but were not. Listing the decryption rate of files based on their extension may help in this.

Also the cleartext pattern must be strong enough to have a negligible chance of giving a false positive, i.e. identifying random data as decrypted data. For exemple, as we expect to test a large number of keys (at least $2^{32}$), if we check for a constant marker at the beginning of the file, that marker should have much more than 32 bits. We saw no false positive with markers of 8 bytes (64 bits), but we had a few with markers of up to

44 bits (pdf marker: 5 bytes magic plus a few constraints on some more bytes encoding the pdf specification version).

### 2.2 Key generation cases

There are a few cases to consider when we look at how the RC4 keys may be generated regarding the tick value.

The GetRandByte function can be considered as a stream for a given tick value, with a sudden change when the tick changes.

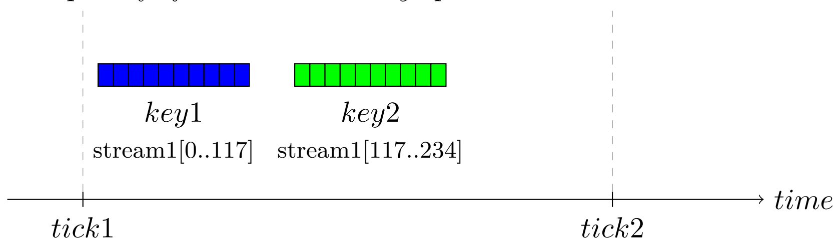
The good news is that this stream is used exclusively for generating RC4 keys to encrypt files: no other part of the malware calls this PRNG.

This leaves us with different cases for the malware to pick keys, presented from the easiest (for us) to recover to the hardest.

**Case 1** One or more keys generated sequencially in a single tick. The first key is the first 117 bytes from the stream for this tick, the 2nd key is the next 117 bytes, and so on.

To recover them, we only need to find the tick value and the number of keys.

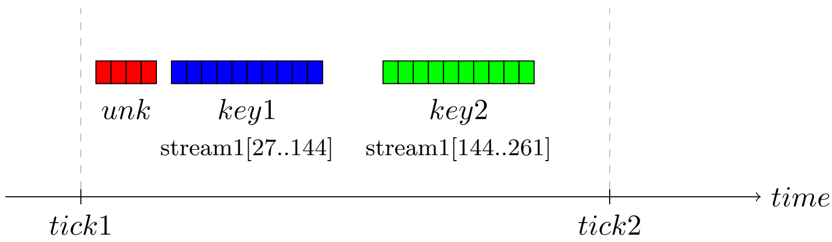
Complexity: $find_tick * n_key_per_tick$




**Case 2** Something happened at the beginning of the tick, consuming bytes from the PRNG, and then one or more keys are generated sequencially. These keys are sequences of 117 bytes from the stream, but the first byte index of the first key is not a multiple of 117.

To recover them, same as case 1, but we must also find the offset value (116 possible values).

Complexity: $find_tick * n_key_per_tick * 116$



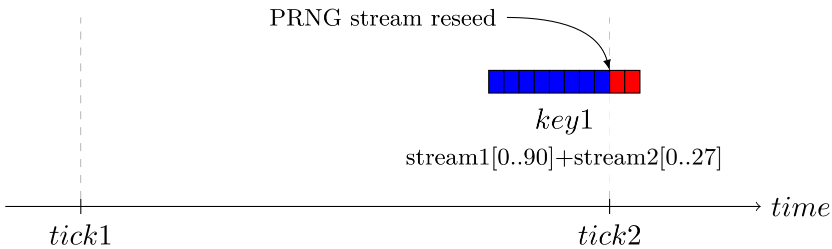

**Case 3** Tick change during key generation: one RC4 key was drawn near the end of a tick, and the tick value changed during the drawing of the key bytes.

To recover it, we need to find the tick value for the first half, how many bytes of key were drawn in that tick, and by how much the tick value advanced for the remainder of the key. This increment is usually 15 or 16, so we will test those two values.

This type of key may induce a shift in the stream for the next tick if more keys are drawn (cf case 2).

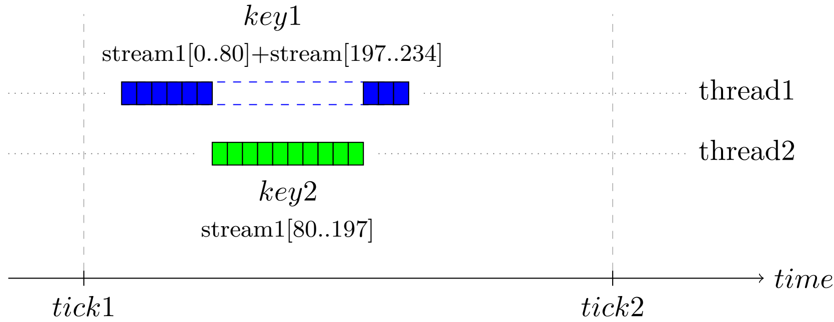
Complexity: $find_tick * n_key_per_tick * 116 * 2$

We may also want to search for keys who begins shifted (case 2).



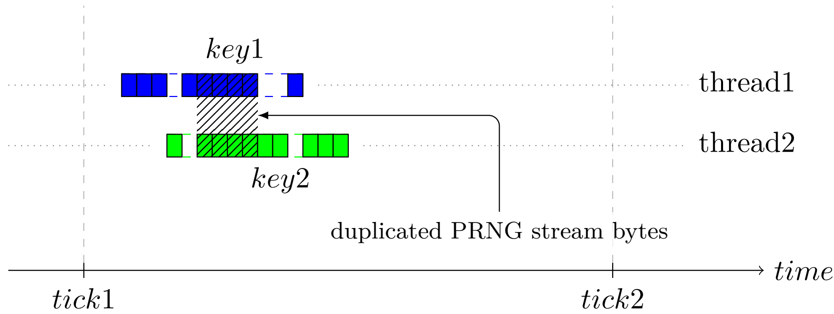

**Case 4** Two threads interrupted one another when drawing keys. We will need to figure at which point exactly the interruption happened, and how many keys were drawn during the interruption.

Complexity: $find_tick * n_key_per_tick * 116 * max_gap$




**Case 5** Two threads raced one another when drawing keys. The malware code is not threadsafe, so both threads may reuse the same portion of the PRNG stream. At this point we are at the end of our luck, we may try keys from random ranges of the PRNG stream for one tick value.

Complexity: hard




**Other cases** Those were cases we actually encountered and for which we found one or more keys in our data. But we can easily imagine others: more that 2 threads racing, maybe a few tick change here and there etc. Unfortunately, there are endless possibilities, and those are prohibitively computationaly expensive to test.

These unpredictabilities in the key generation do not impact the malware: whatever the key is, it is used to encrypt the data and is saved at the end of the file, so the owner of the private RSA key can always recover the RC4 key. It only hinders us when trying to bruteforce the RC4 key by reproducing the original (nondeterministic) malware behavior.

So we already know that there may be files whose key we will not be able to recover ; we can only hope that this will occur on files of lesser importance.

### 2.3 Decrypting many files

At this point, we received a first batch of files to decrypt (near 17000 files).

When decrypting one file, the bottleneck in computation may be the PRNG seeding, or the RC4 key scheduling or decryption. But when handling a lot of files, this becomes negligible compared to the check of each individual decrypted header, it is therefore not too useful to optimize heavily the crypto operations.

The approach we took is to have a fast bruteforcing binary that will take a range of ticks and a list of files. It will on startup load the headers of all the target files (we need 256 bytes of data to evaluate the entropy of the cleartext) and keep those in memory. It will then iterate over the given tick range in parallel using OpenMP, and for each tick value, seed a Mersenne Twister with it, draw some RC4 keys, and generate the 256-bytes RC4 streams. Finally, for each loaded file header, the RC4 streams are applied (with a xor) and the resulting cleartext is tested to check if a successful decryption was found.

When a header is successfully decrypted, we fully decrypt the file alongside its encrypted counterpart, and also generate a small textual information file to store some decryption metadata: the tick value, the index inside the tick (the nr of keys drawn before in this tick), the raw RC4 key etc.

This fast binary is then piloted by a smart script® whose job is to find the best ticks/index/files to test for best results. It works by running the binary, and then parsing the information files generated with the newly decrypted files to find the next ticks to test, in order to create a vertuous circle of continuous improvement.

### 2.4 Time estimates

Using our implementation, testing the first key of all possible tick values on one file header takes 12h on one CPU.

Testing the whole $2^{32}$ tick range with an index depth of 40 keys per tick for 20000 files would take 400 years on one CPU.

Note that the work is trivial to parallelize by distributing ticks: throw 100 CPUs at it to take 100 times less time.

Then you can do the same thing 116 times again for every possible offset, to identify keys in case 2.

## 2.5 Heuristics

We will now present how our script works.

The logic behind this is to limit the bruteforce to the minimum while maximizing the results. To that end, we focus on solving the highest number of files by targetting the least difficult, so that when we start doing the most computationnaly intensive tasks, we have less files to test against each key.

The ticks are not randomly distributed, they are time-based. The tick range covering 1 hour is 3600000ms ($\approx$ 0x370000). This is much less than the full range of possible ticks, which spans almost 50 days. So our goal will be to quickly find an initial tick value actually used during the malware execution, so we will be able to focus around that value to find most of the keys.

As we are trying to recover tick values actually used by the malware, it is best to work with encrypted files coming from only 1 machine at a time ; it does not make sense to try these keys on files coming from another run on another machine as the ticks will be mostly unrelated.

The script follows this algorithm. After each step, if we found new keys, we restart from step 1. Initially, as the set of known keys is empty, we start with step 4.

1. For all ticks used to actually decrypt one file
   - Try to find the highest index of keys in case 1 in this tick
   - Try to find the highest index of keys in case 2 in this tick
   - Try to find keys in case 3 in this tick
2. Compute a list of short ranges around actual tick values (0x10000)
   - Try to find case 1 keys in these ranges, searching with a maximum index value of 2
   - For each new tick found, go back to step 1
   - Search for keys in case 2 (same max index)
3. Compute a list of longer ranges (0x200000)
   - Search for case 1 keys with max index 2
4. Select a random small sample of encrypted files (100 files)
   - Search for case 1 keys with index 0 for those on the full tick range (0 - 0xFFFFFFFF)

The script stores all parameters of the bruteforce it starts, to avoid doing the same thing twice.

For exemple, it stores the range and maximum index of all runs, as well as the list of individual ticks with their maximum index. When trying new ranges, it will subtract the ranges already covered.

It also stores the list of files gone through step 4 ; the algorithm stops when all undecrypted files have gone through step 4.

A shortcut is made in step 2, where a first run is done on the subset of encrypted files whose path on the filesystem is close to the ones whose ticks were found to be in the range. If no new decryption is made, the run is made on all currently undecrypted files.

After the algorithm is done, we manually inspect the results, by listing all files either sorted by tick + index or by their path, to identify clusters of undecrypted files most likely to yield under the more intensive searches for keys in case 4 and 5 ; but this is mostly "last hope" style manual search.

### 2.6 Issues

This algorithm works well.

There are two notable issues that may affect a successfull decryption, both related to the ability to distinguish decrypted cleartext from random garbage.

Small files are naturally badly tailored for entropy calculation. The entropy calculating algorithm needs a minimum buffer size to output meaningful values. If we work on files whose cleartext length is under 256, the entropy score we get will be less reliable, and will have a much higher variance. As we try more keys, we will be more likely to produce a false positive, where the entropy of the result is low but does not map to the decrypted file.

For these small files, we apply a custom algorithm instead of relying on entropy. On our set, most of those files were text or text-encoded files (xml) ; so we identify cleartext based on the ratio of number of "space" characters to file size, the number of nul bytes, or if the file consists only of printable ascii bytes.

The second and more problematic kind of files fitting badly our methodology is files whose clear text data has a high entropy. For those, the bruteforcer is unable to distinguish the good decryption key.

We solve this under two angles. If we can identify an explicit clear text pattern for those files, e.g. by printing the undecrypted files extension sorted by count, then we can add that pattern to the bruteforcer so he will correctly identify the decrypted files, whatever the entropy score is. We did this for TIFF files : they have a short header (4 bytes signature), insufficient for directly identifying a cleartext, however still rare enough when bruteforcing that when we encounter it, we can run more costly tests

to validate the nature of the cleartext we have, without compromising speed.

The 2nd way we used is to allow a higher entropy score threshold for the files, but to balance the risk of false positive, we allow this higher threshold only on a very limited number of keys to test. We use the tick and index values of closely related files based on the file paths: as we know in which order the malware walks the files on disk, we assume that files close in this regard will also have close tick values.

This step is added between steps 3 and 4 of the algorithm.

## 3 Results

### 3.1 Batch 1

The first batch of files we had to decrypt consisted of 16849 files, mostly documents (MS office, pdf) and images (tiff).

We managed to recover 16056 of those files (95.3%).

It took us initially 2 weeks on one server with 56 threads to recover most of those files. We are now able to re-run the whole process in less than 8 hours on the same server using our improved heuristics.

We identified 4124 distinct ticks used for key generation.

The ticks values can be regrouped in 4 distinct clusters. This may be because the files we worked on were encrypted by distinct runs of the malware (e.g. on different machines) or because the files represent only a subset of all the files of the computer, and we have no data on the directories that may have been encrypted by the malware during the gaps in ticks we see.

Tick values are a kind of timestamp, and the length of the largest cluster indicates that the malware took at least 30 minutes to encrypt the files.

| tick values range | nr of keys |
|---|---|
| `0x01077285 - 0x012104E5` | 8870 |
| `0x02904119 - 0x02A3C79F` | 4777 |
| `0x1C389070 - 0x1C3CB992` | 1636 |
| `0x1D1AB958 - 0x1D1DC71E` | 2067 |

**Table 1.** Tick values distribution, batch 1

Most of the recovered keys are in case 1.

| case | nr of keys |
|---|---|
| case 1 | 15602 (92.6%) |
| case 2 | 447 (2.7%) |
| other | 7 |
| not decrypted | 793 (4.7%) |

**Table 2.** Key distribution per case, batch 1

For all case 1 keys, we can list the index at which each key was found. Nearly 1/4 of all keys were case 1 index 0, which strengthen our first heuristic used to find some initial tick values.

The largest index we find is 36.

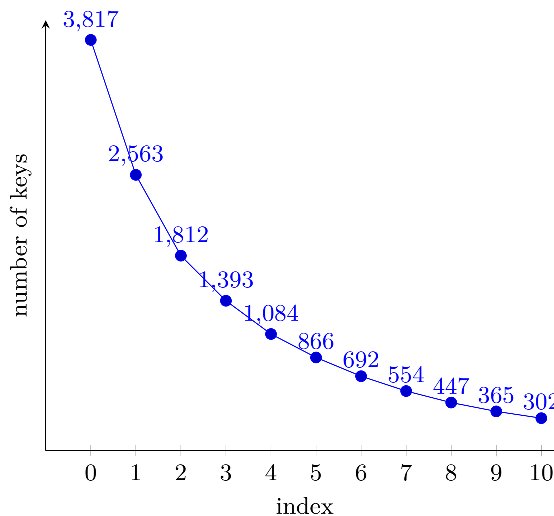



**Fig. 1.** Number of keys per index, batch 1

### 3.2 Batch 2

We next worked on a 2nd, larger batch of files. This one consists of 231513 encrypted files.

We recovered 207256 files (89.5%) using approx 1 month of computation on the server, which is still underway (working unsupervised as we speak due to COVID-19 contingency measures).

We identified 32718 distinct ticks used for key generation.

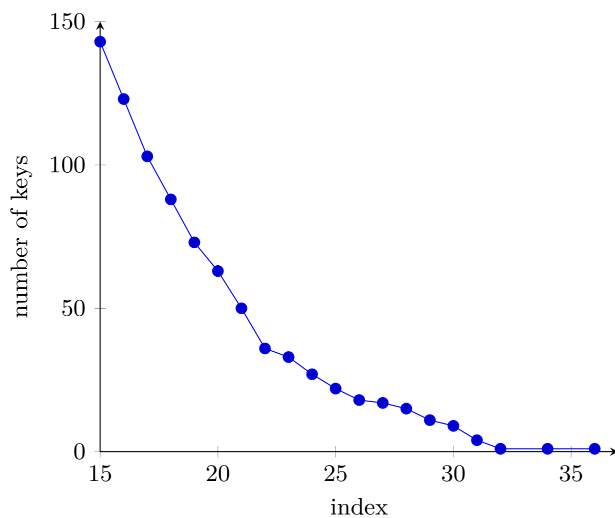



**Fig. 2.** Number of keys per index, batch 1, continued

The largest range indicates the ransomware worked more than 10h to encrypt the files (if they all come from the same machine).

| tick values range | nr of keys |
|---|---|
| `0x000f65f3 - 0x017128db` | 115528 |
| `0x02a0d69d - 0x02a450ea` | 4202 |
| `0x1c30d2ad - 0x1c3d1587` | 4238 |
| `0x1d1a7612 - 0x1f54c381` | 83288 |

**Table 3.** Tick values distribution, batch 2

The largest index we find is 55.

| case | nr of keys |
|---|---|
| case 1 | 198694 (85.8%) |
| case 2 | 8510 (3.7%) |
| other | 52 |
| not decrypted | 24257 (10.5%) |

**Table 4.** Key distribution per case, batch 2

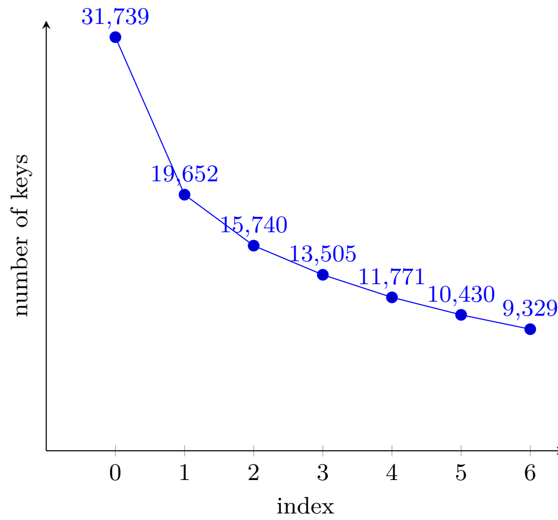



**Fig. 3.** Number of keys per index, batch 2

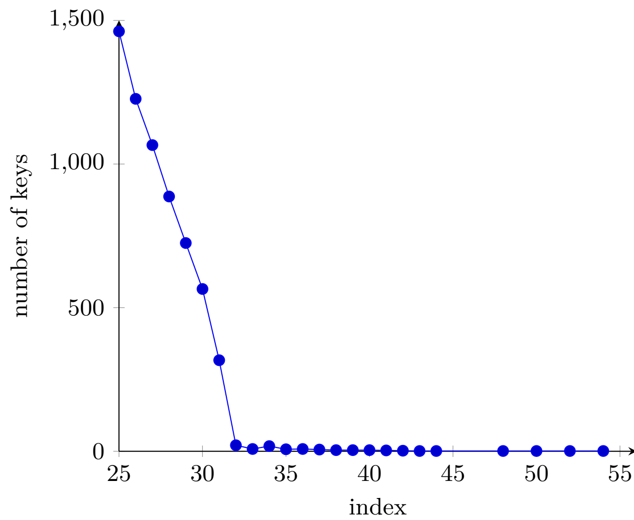



**Fig. 4.** Number of keys per index, batch 2, continued

## Conclusion

It was a challenging problem, for which we obtained pretty good results.

However in most ransomware cases, the crypto is not so weak and we simply cannot do anything to recover the files. Even there, where the crypto flaw is quite huge, we still needed a lot of time to recover the files, during which the victim had no access to its data, and some of the files may be lost forever.

The strongest protection against ransomwares is to have offline backups. Do you have those ? Does your company ?

Stay safe !