A way of decrypting particular malware payloads found in MZPE files

Tudorică Radu[1,2], **Radu Rareş-Aurelian**[1,2], **Emil Simion**[3]

[1]Faculty of Computer Science, Alexandru Ioan Cuza University of Iaşi
[2]Bitdefender, România
[3]Politehnica University of Bucharest

Abstract

Back in the 90s when the notion of malware first appeared, it was clear that the behaviour and purpose of such software should be closely analysed, such that systems all over the world should be patched, secured and ready to prevent other malicious activities to be happening in the future. Thus, malware analysis was born. In recent years, the rise of malware of all types, for example trojan, ransowmare, adware, spyware and so on, implies that deeper understanding of operating systems, attention to the details and perseverance are just some of the traits any malware analyst should have in their bag. With Windows being the worldwide go-to operating system, Windows' executable files represent the perfect way in which malware can be disguised to later be loaded and produce damage. In this paper we highlight how ciphers like Vigenère cipher or Caesar cipher can be extended to more complex classes, such that, when later broken, ways of decrypting malware payloads, that are disguised in Windows executable files, are found. Alongside the theoretical information present in this paper, based on a dataset provided by our team at Bitdefender, we describe our implementation on how the key to decryption of such payloads can be found, what techniques are present in our approach, how optimization can be done, what are the pitfalls of this implementation and, lastly, open a discussion on how to tackle these pitfalls.

1. Introduction

Malware creators are always trying to find new ways of evading anti-viruses to avoid being detected. This methods range from elementary ones, for example trying to impersonate a legitimate process, to sophisticated ones, like sandbox detection, anti-API hooking methods, detecting or disabling anti-virus products and so on.

One of the types of detections that they are trying to evade are file detections. These types of detections are based on the content of a file and their role is to prevent attempts of malicious activity.

Advanced malware, as most programs, is time consuming and hard to write, meaning that authors cannot rewrite or easily modify their code when their files start being detected. This is the context where the so-called crypters (also sometimes called packers, bootloaders or loaders, terms which we will use interchangeably throughout this work) were created. The sole purpose of a crypter is to hide the real malware within itself or somewhere else on the disk, thus being easily modifiable and re-writable to make it easy to avoid file detections.

The most common way these crypters hide the real malware is by having an encrypted payload of the malware inside them and then decrypting, loading the result into memory and executing it. However, since they need to be easily modifiable and re-writable, the encryption cannot be too

complex or use well known libraries such as OpenSSL. As a consequence, most of these crypters use very basic algorithms or use "home-brewed" algorithms. Although they are not advanced programs, crypters can be easily obfuscated, either by hiding their decryption functions, hiding their code with other open source software to make analysis harder by making analysts spend time on reversing unrelated code, using control flow graph flattening techniques or by using languages such as Rust, Go or Nim to make analysis harder and more cumbersome.

Ciphers that are similar to Vigenère's cipher are popular inside crypters, where, usually, the ADD operation is replaced with Bitwise XOR. In practice, if the XOR key is relatively short, or rather the payload or the malware contains lots of NULL characters, it is trivial to observe the key in a hex-editor. However, for long keys, it might be close to impossible to spot them visually.

In general, people such as researchers or incident responders might not have a complete view of how an attack happened, thus, they might only have the malware loader to analyse (which is time consuming) or might only have the encrypted payload without any clue on how it it being decrypted and then loaded in the memory. This got us thinking that, with certainty, there exists a cryptanalitic way of solving instances of such problems.

Throughout this paper we make references to the notions of Vigenère cipher [1], Ceaser cipher [2], Chi-square test [3], index of coincidence [4], magic numbers [5] and more. These concepts represent the basic knowledge on which this paper has its basis. The usage of each concept, alongside each concept's brief description, will be given when the context fits, underlining the fact that each one of them play an important role in this paper.

As stated before, static analysis of crypters can be hindered in many ways and can be time consuming without aiding in the real purpose of the work which represents analysing the complex malware inside the payload. As such, our work focused on building ways to decrypt malware payloads in which different forms of Vigenère ciphers are used. In this paper we show how the classical Vigenère and Caesar ciphers can be generalised and extended to a more complex class of ciphers, how to break some of those ciphers when the plaintext message is a Windows executable file and we show some open problems relating to these kinds of ciphers. We chose to limit ourselves to only Windows executables on the x86/x64 architecture because attackers are targeting Windows systems the most.

2. Prerequisites

2.1. MZPE Format

Windows executable files are structured in multiple sub-components that contain all the logic and information necessary to load the file into memory and run the machine code inside it. From now on, Windows executable files will be referred to as MZPE files and all the sub-components will be referred to as chunks.

The first chunk of a MZPE file actually represents a very small DOS executable which is included for backwards compatibility reasons, followed by a set of headers that define a series of sections which contain different types of data, alongside other resources. These sections have different purposes, such as encapsulating the machine code that needs to be ran, read only structures that are being used, resources, relocations, an so on. At the end of the file there might be other data that will not be loaded into memory, but might be accessed by the program itself. We will call this chunk the overlay. In between sections there can also be chunks that we call gaps which will be skipped when the file is loaded.

In depth knowledge about each section, the headers and data directories is not required to understand this paper, but nevertheless, it is important to know that each chunk has a different purpose and structure to it. As such, we will treat each chunk as a different "language" with sufficiently different values for their index of coincidence that the chance for a successful decryption is better.

2.2. Vigenère Cipher

Before talking about Vigenère cipher, we will first briefly talk about Caesar cipher since some aspects between the two are similar, thus some observations can be made when we look at them side by side. Caesar cipher's encryption is defined as the application: $c_i = m_i + k \pmod q$ where $c, m \in Z_q^n$ are the ciphertext and the plaintext with a length of n ($c_i, m_i \in Z_q$ are the i'th caracters) and $k \in Z_q$ is the cipher's key. The classic form of this cipher usually implies a conversion function from letters to numbers and back, setting $q = 26$ and using lowercase Latin alphabet letters, referring to the addition as "shifting" the letters by k.

Vigenère cipher's encryption is defined as the application $c_i = m_i + k_j \pmod q$ where $k \in Z_q^l$ is the Vigenère key, where l is the length of the key and $j = i \pmod l$. Essentially, extending the Caesar cipher by using l keys.

A very common encryption algorithm used by malware loaders is a modified Vigenère cipher $c_i = m_i \oplus k_j$ where $q = 256$, c_i, m_i and k_j are bytes and $\oplus$ is Bitwise XOR.

2.3. Extended Vigenère

We can build an extension of both ciphers by generalising them as follows:

Extended Caesar's encryption is defined as the application $c_i = P_k^f(m_i)$ where $P_k : Z_q \to Z_q$ is a permutation defined by the key $k \in K$, where K represents some keyspace that has at least q elements and $f : K \times Z_q \to Z_q$ represents a reversible function.

It is trivial to see that the extended form is a more powerful one, since we can represent the classic form as a subclass of the extended form by setting $f(k, m_i) = m_i + k \pmod q$. Not only that, but, for any input, the classic form can generate only q possible ciphertexts by setting the key to all possible combinations, whilst the extended form can generate $|K|! \geq q!$ ciphertexts.

Extended Vigenère's encryption is defined as the application $c_i = Pk_j^f(m_i)$ where Pk is a table of permutations defined by the key $k \in K^l$ of length l.

The same observation applies here. The extended form can now generate as many as $|K|!^l \geq q!^l$ ciphertexts for each plaintext, making these variations much harder to attack.

3. Data Analysis

The dataset that we used was provided by our team at Bitdefender, to whom we want to express our appreciation and gratitude for their endless help. It consists of a set of ten thousand executable files, either clean or potentially infected. The process of preparing this data was done with technologies and tools like C/C++, Python, YARA tool along with a set of rules to determine information about the files by looking at the content inside of them and, finally, a database to organise all the information that will be processed.

First of all, using YARA (Yet Another Ridiculous Acronym) [6] along with an adequate rule set, either on a chunk of data within the file or on the entire file , we can determine the following: the language in which a program was written, the files that are installers or self-extractors, the type of C compiler that was used, the version of the Visual Studio used based on the linker or header information, and so on. Afterwards, the output of the tool (the *tags* generated) is used to categorize these files and chunks of data.

The processing of the entire file can be viewed as processing a big chunk of data, and such, from now on, when talking about the content of an entire file, this special case of a chunk is to be remembered.

Second of all, all the files have gone through a processing step that does the scanning of each file, either per chunks or on the entire content. Each chunk corresponds to a certain part of the file. For example we have the header of the file or the section of the file, descriptions for these types of chunks are mentioned above in 2.1. The reasoning behind this approach is that we want to compute the frequencies of each byte. It is neither useful to compute the frequency based on

something less than a byte (i.e. splitting the byte in more parts), neither relevant or usable in practice to compute the frequency on more than a byte.

Having the first two steps done, we can now combine the information that we have in such way that two collections are created. One collection is dedicated to the information based on the scanning of the chunks and the other collection is dedicated to the information based on the scanning of the entire file. Each collection contains documents-like objects in which information such as the map of the frequencies of each byte, the size of the data processed or the tags generated from the output of YARA is stored.

Based on these two collections, taking into account the YARA tags computed and the types of chunks, we can now split our data in clusters based on these tags and types of chunks. Each tag combined with each type of chunk, will separately represent a cluster, on which further processing will be done. We will compute the index of coincidence [4] and the entropy [7] on the content available on the whole cluster, for each cluster. Currently, based on the dataset, we have reduced the number of clusters from around 800 to 136 by eliminating the clusters which have a high entropy value, close to the maximum possible value, meaning that the content of these clusters is completely random.

Next step, just like when trying to break Veginère's cipher, is to compute the best possible key length. This is done by applying the algorithm present and described in 4.1. Because the possible key length found may be bigger than the actual key length, certainly we can asses that the actually key length is found amongst the divisors of the key length outputted by our algorithm. Iterating through the divisors of the length of the key, from the greatest to the lowest divisor, we can now apply the algorithm that calculates the key described below in 4.2 with the input given by the targeted encrypted chunk, the length of the key and the clusters that found the best possible key value. Afterwards, the verification of the found key is done on the "header" of the chunk by looking for the magic numbers [5] that are specific to MZPE files.

In our implementation, we focused on decrypting the chunks that represent the content of the file where the MZPE files are located because we strongly believe they are the most of interest in our field. Of course, the environment and implementation on decrypting other specific chunks of a file is already done, hence the impact of the implementation can be easily extended.

The results come with two sides. On one side, we can easily break the encryption when the operations are simply XOR or ADD using the methods described above, however, on the other side, diverse operations, like chaining multiple logic operations and not only, can significantly increase the complexity space of finding the key.

4. Algorithms

In this section, we will describe the main algorithms present in our implementation, along with a bunch of observations that are present in 4.3

4.1. Determining the length of the key

This algorithm takes as input a ciphertext, in our case it will be a chunk of data, and outputs a pair that contains the best length possible that was found for the key, along with the cluster on which the algorithm was applied on. Essentially, we will use this function to create a list containing pairs that have the described structure.

We make use of the average index of coincidence of each cluster, along with the index of coincidence calculated on each Caesar Block. The method *SplitIntoCaesarBlocks* essentially iterates through the ciphertext and splits it starting from the index 0 up to the current key length with key length as an iteration step until the index is equal with the key length (i.e. $for\ i\ in\ range(key_length) -> c[i :: key_length]$). Having the blocks prepared, on each block we calculate the index of coincidence [4] and afterwards we calculate the average index of coincidence. At each iteration, we compare the best average found against the freshly computed average, if we find a better average, we update the best average.

For a slight optimisation, we use the parameter α to represent the minimum length that a Caesar Block needs to have such that the Index of Coincidence is statistically significant. The final decided key length is determined via a "voting" process, where we take the length determined by most clusters.

Input: the ciphertext $c \in Z_q^n$
Output: the length l of the key, the subset of clusters Clusters_l

$\text{Clusters}_{i \in \{1,2,...\frac{n}{\alpha}\}} \leftarrow \{\}$
foreach cluster $\in$ Clusters **do**
  Length $\leftarrow 0$
  MinIC $\leftarrow 0$
  for $i = 1...\frac{n}{\alpha}$ **do**
    Blocks $\leftarrow$ SplitIntoCaesarBlocks(c, i)
    AvgIC $\leftarrow \sum_{\text{Block} \in \text{Blocks}} \frac{IC(Block)}{\frac{n}{i}}$
    if $|\text{AvgIC} - IC_{cluster}| < |\text{MinIC} - IC_{cluster}|$ **then**
      MinIC = AvgIC
      Length = i
    end
  end
  $\text{Clusters}_{\text{Length}}$.append(cluster)
end
return argmax($\{l : \text{Clusters}_l.\text{count}() > \text{Clusters}_x.\text{count}(), \forall x \in \{1, 2, ...\frac{n}{\alpha}\}\}$)

Algorithm 1: Finding the key length using the index of coincidence

4.2. Finding the key

Finding the key algorithm takes as input the ciphertext, a possible length of the key, a set of clusters for which the best length of the key was the length of the key given as input and a family of functions which will be applied and outputs the key found.

These functions define a set of possible extended Caesar decryption functions (i.e. they generate the inverse permutation for an extended Caesar cipher). As we did previously, we split the ciphertext into Caesar Blocks. For each function, we take each block and each cluster, then we decrypt the block and apply the Chi-squared test on the bytes' frequency observed in the cluster with the observed bytes' frequency of the decrypted block and save these values.

By the definition, the minimum value that resulted after applying the Chi-squared test should show what key produced a plaintext who's distribution is closest to that of the cluster. The output is a set of multiple keys who create the closest distribution to each cluster for each function.

In practice, the results of this algorithm can then be used to check whether or not the decryption produced a valid result (may it be a Windows executable, a shellcode, etc.).

Variations of this algorithm include the usage of the Mutual Index of Coincidence instead of the Chi-squared test, applying the same reduced bruteforce method.

This algorithm, however, has a series of flaws and won't always result in a successful decryption. Besides it's speed, neither metrics (Chi-squared or Mutual Index of Coincidence) can't guarantee a successful decryption in every possible case. Another flaw is the fact that if the inverse of the function used to encrypt the plaintext isn't present in the set of functions given to the algorithm it will never produce the correct key.

A very interesting alternative to the algorithm below is the usage of frequency analysis to determine the best permutations for each block. This would give a much faster alternative and would remove the "guessing game" for what possible functions could have been used as the basis for the extended Caesar cipher.

Input: the ciphertext $c \in Z_q^n$, the length l of the key, the subset of clusters Clusters_l, p functions $f_i : K \times Z_q \to Z_q$

Output: the keys $k_{f_i} \in K^l$ for each function

```
Blocks ← SplitIntoCaesarBlocks(c, l)
foreach f_i do
    foreach cluster ∈ Clusters_l do
        k_{f_i} ← {}
        foreach Block ∈ Blocks do
            foreach k' ∈ K do
                ProposedPlaintext = f_i(k', Block)
                ChiTable^{fi}_{k'} = chi2(Frequency_cluster, Frequency_ProposedPlaintext)
            end
            k_{f_i}.append(k' : ChiTable^{fi}_{k'} < ChiTable^{fi}_{k''}, ∀k'' ∈ K)
        end
    end
end
return k_{f_i}
```

Algorithm 2: Finding the key using $O(l)$ bruteforce

4.3. Pitfalls of the Extended Vigenère

The methods described base themselves on the fact that the exponential hardness of Vigenère cipher can be reduced by knowing statistical data regarding the message space. As such, the $chi2$ method and the MIC method reduce the complexity from $O(256^l)$ to $O(256 * l)$ by guessing that the function describing each permutation only has one parameter. An extension of that, extended Vigenère cipher, makes it such that these methods are reducing the complexity only to $O(256^p * l)$, where p is the number of parameters for f.

The only generic way we could try to resolve an extended Vigenère cipher is by using the character frequency method which doesn't make any assumptions about how the permutation function works. The downside of this method is that it needs a reasonably long ciphertext of order $O(t * l)$, where t is the necessary length, such that it is likely to expect frequencies to appear in the same exact order.

With that said, finding a better method for solving an extended Vigenère cipher remains an open problem.

5. Conclusions

In this paper we have: - shown how Vigenère ciphers can be broken when they are being used to encrypt Windows executable files, - shown how these methods might not always work, - opened the discussion of decrypting malware payloads where Vigenère-like ciphers are used by extending Vigenère and Caesar ciphers when MZPE files represent the plaintexts and - offered a way of taking advantage in finding the decryption key of these extended variants by using different methods including clustering, the index of coincidence or Chi-squared test to reduce the space complexity of the operations.

References

[1] A.-A. M. Aliyu and A. Olaniyan, "Vigenere cipher: trends, review and possible modifications," *International Journal of Computer Applications*, vol. 135, no. 11, pp. 46–50, 2016.

[2] O. Omolara, A. Oludare, and S. Abdulahi, "Developing a modified hybrid caesar cipher and vigenere cipher for secure data communication," *Computer Engineering and Intelligent Systems*, vol. 5, no. 5, pp. 34–46, 2014.

[3] P. E. Greenwood and M. S. Nikulin, *A guide to chi-squared testing.* John Wiley & Sons, 1996, vol. 280.

[4] W. F. Friedman, *The index of coincidence and its applications in cryptanalysis.* Aegean Park Press California, 1987, vol. 49.

[5] Magic numbers (file signatures). [Online]. Available: https://en.wikipedia.org/wiki/List_of_file_signatures

[6] V. M. Alvarez. Yara tool. [Online]. Available: https://virustotal.github.io/yara/

[7] C. E. Shannon, "A mathematical theory of communication," *ACM SIGMOBILE mobile computing and communications review*, vol. 5, no. 1, pp. 3–55, 2001.