UNIVERSITA' DEGLI STUDI DI GENOVA

Towards the Detection of Low Entropy Packed Malware

by

Alessandro Mantovani

A thesis submitted in partial fulfillment for the degree of Computer Science Engineering

in
DIBRIS

October 2018

Contents

1 Introduction **7**

2 Background knowledge **11**
2.1 Basics of Operating systems . . . 11
2.1.1 Process creation. . . . 12
2.2 Malware Analysis: uses and practices . . . 12
2.3 Obfuscation and Packing . . . 16
2.3.1 Focusing on packing . . . 16
2.4 Frameworks and tools employed . . . 18

3 Low Entropy Packed Malware Schemes and Spread **21**
3.1 Entropy of Executable files . . . 21
3.1.1 Analysis . . . 23
3.2 Schemes Taxonomy . . . 24
3.2.1 Dataset . . . 29
3.2.2 Distribution and Spread . . . 29

4 Case studies **31**
4.1 Case study: the Beria packer . . . 31
4.2 Case study: xor encryption . . . 32
4.3 Case study: custom encoding . . . 32
4.4 Case study: transposition cipher . . . 35

5 Static analysis approaches **37**
5.1 Related work on static analysis . . . 37

6 Future work and conclusions **43**
6.1 Conclusions and remarks . . . 43
6.2 Future work . . . 43

A mia mamma Cinzia e a mio fratello Francesco per gli incoraggiamenti, il sostegno e per avermi sempre spinto a dare il meglio

UNIVERSITA' DEGLI STUDI DI GENOVA

Abstract

by Alessandro Mantovani

For several years now, we have observed thousands of malware which attempt to hide their malicious code through some form of obfuscation. A typical obfuscation technique is *packing* which is probably the preferred one by malware authors because it masks the code just to make life difficult to malware analysts and Antivirus (AV) software vendors. An open research problem on malware analysis is distinguishing between packed and non-packed executables since a correct separation between the two classes allows AV software to define the right heuristic to use for malware detection. In addition to this, a reliable classification between packed and not-packed binaries is useful for malware analysts and researchers as well, because it can improve the quality of their works creating sound datasets without false positive/negative samples, i.e., avoiding dataset pollution. However, the assumption *"packed implies malicious"* is absolutely wrong because a lot of legal software producers adopt packing or other obfuscation techniques in order to protect their code from abuses. Therefore it is evident how a wrong answer to the question *"is this executable packed?"* regarding a malware makes the difference between evasion and detection. Current state of the art relates entropy and packing, thereby assuming a high entropy of an executable file implies that the same is packed. Unfortunately, an increasing number of packed malware implement proper schemes to keep their entropy low. In this work, we try to point out how widespread these samples are and how they work, with the purpose of demonstrating that the problem of a correct classification between packed and non-packed binaries is still open and that it must not be ignored.

Chapter 1

Introduction

Both malware and goodware have many reasons to hide their behavior; usually the former have to face anti-malware engines, the latter deter reverse engineering from who tries to break their software protections. Among hiding techniques, *obfuscation* is the most widespread one and it is often included in top-notch development infrastructures, e.g. Obfuscator-LLVM [26] and Proguard [28]. The aim of obfuscation is to syntactically modify the code to make it difficult for humans to understand and for computers to analyze.

Packing is a type of obfuscation that compresses/encrypts, or in general, transforms, an executable file and generates another executable file composed of the previously transformed data and the routine needed to restore the original code. When a packed binary is executed, the decompression/decryption routine recreates the original code from the packed one before executing it.

The problem of correctly answering the question *"is this executable packed?"* is fundamental in malware analysis. While false positives waste analysis time (automated approaches [37] for unpacking are incredibly time-consuming because they cannot make assumptions on the packer architecture; therefore they have to perform extensive dynamic analysis on the sample), false negatives are not analyzed in the correct way and the likelihood that a potential malware evades detection is really high. Moreover, while benign files are sometimes packed as well, the sole fact of being packed is often considered as suspicious (i.e., still today several anti-virus mark packed files as potentially malicious). Finally, it is crucial to highlight the importance of packed samples for dataset pollution. Many studies explicitly exclude packed samples (or first need to unpack them) before extracting features and performing static analysis or machine learning. The *modus operandi* adopted to build the dataset in the previous state of the art work is depicted in the last column of Table 5.1.

Researchers often classify executables as packed or not according to their entropy: low entropy indicates unpacked machine code, while high entropy suggests that the code is likely compressed or encrypted. While early studies (e.g., [29]) classified executables according to their average entropy, more recent work often takes into consideration the entropy computation at a higher granularity, i.e., by recurring to sliding windows or for each section of the executable. In particular, authors in [24], achieved the 99% of accuracy and precision by relying on entropy computed for each section.

At this point, despite entropy became a discerning metric to discover packed code, creating a packed executable with a low entropy means to increase the chances of evasion. Thus, the first malware families using schemes to create low entropy data from code appeared. Even if this phenomenon was known to researchers, they decided not to take into account these samples for their studies based on the creation of proper datasets. Indeed they assumed that such low entropy malware could not invalidate their experiments as it was not sufficiently widespread.

From now on we will refer to malicious executables making use of dedicated packing techniques to evade entropy checks as LEPM —Low Entropy Packed Malware.

The first academic work to discuss the existence of LEPM is by Ugarte et al. in 2012 [46]. In their study, the authors encountered packed samples (including a few examples of the *Zeus* [43] family) which used naive techniques to minimize their entropy level like *byte padding* or *byte appending*. Such techniques consist in appending the same byte for a certain number of times so that it increases repetitiveness and decreases entropy. They proposed a method for calculating entropy profiles dividing the file into overlapping regions; thereafter they calculate the entropy of each region independently, obtaining an entropy profile that represents the randomness of the file in a fine-grained style.

Probably for this reason, works produced in the scope of packing, suppose that entropy is “good” enough to make a correct classification, without considering LEPM. For example, from the point of view of static analysis, authors of [29, 34, 40] tried to find packed malware employing entropy as the fundamental metric, without considering many families that use a dedicated technique to decrease the entropy.

Successive researches in packing detection suggest different classifications that can be divided into two categories: i) off-the-shelf packers (e.g., UPX described by [19]) and ii) custom packers (partially treated in [46]). But again, LEPM was not mentioned.

Our work is motivated by the need to address the following questions:

- There exists a packing technique that grants low entropy without the need to insert additional padding?
- Are LEPM frequent enough to produce an observable error if a classification between packed and not packed files is performed?
- Which are the schemes used by LEPM to decrease entropy?

- Are LEPM undetectable w.r.t. the features considered in state of the art static analysis?

We created an appropriate dataset of possible LEPM with strict requirements on entropy and we developed a dynamic analysis tool in order to classify our samples and separate the false positives. From this analysis we discovered that LEPM are a non negligible 15% among malware with low entropy (i.e., malware that should not be packed according to entropy) and this is a significant rate that should be considered in future works. Moreover, we have classified LEPM according to their low entropy schemes and we show the distribution of such schemes. We also show some case study for each schema in which we state the details of its internal functioning. Finally, we created another malware dataset composed by packed (off-the-shelf), not packed and LEPM to compare the features considered in all the previous work about static analysis for packed/not packed classification. Our work concludes by showing how such features are inadequate for LEPM detection.

In light of this, and considering what will be shown in the following sections, the final outcome of this work is that entropy is no longer a sufficiently discerning metric to solve the problem (at this point resumed) of the distinction between packed and not packed binaries. Future studies in this direction will be obliged to take a step forward with regard to previous work and to identify a reliable metric for the detection of packed executables.

This thesis is organized as follows: Section **??** introduces the formal definition of entropy, shows its properties and how it was used in previous works. Section 3 describes our initial dataset and it shows the result of our dynamic analysis, that is the LEPM taxonomy and its distribution. Section 4 presents some interesting LEPM case studies for each type of LEPM. Section 5 proposes a survey of the state of art with regard to the static analysis techniques to give intuition about how LEPM could evade such approaches. Finally, section 6 explains how we intend fo finalize this work with the aim of publishing it.

Chapter 2

Background knowledge

2.1 Basics of Operating systems

A malware is basically a software which is executed by an operating system (OS from now on). A correct and clear comprehension of the process of execution of a binary file is essential for understanding the techniques adopted by a malware to perform its exploit. Nowadays, malware is currently written for a lot of different OSes and architectures. In this work we focus on malicious software written for Windows OS running over Intel x86 architecture because this is the most widespread kind of malware. Thereby, in the following sections, we will consider Windows as reference OS.

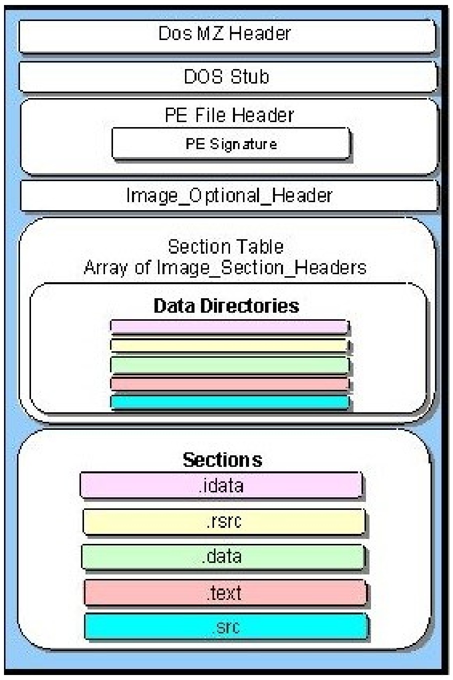
The format of Windows's executable file which is mostly used by both malicious and benign software is called PE (Portable Executable) [7]. It contains the structured information required by the loader for properly handle the code and the data belonging to that binary file. The wrapped information is structured according to a set of headers and sections which tell the OS how to map the file in memory and the protection to use. A header is a data structure containing a set of fields in a known location which describes how the remaining part of the file looks like. Essentially, these header fields contain information like the size of code or data, the address of the entry point, the stack size or other remarkable things which can be used by the OS. Differently from the headers, the sections mainly contain code or data.

For each section some memory protection flags are indicated:

1. The 'R' flag indicates that the current section can be read
2. The 'W' flag indicates that the current section can be written
3. The 'X' flag indicates that the current section can be executed

The main purpose of memory protection (which is enforced by the underlying OS) is to prevent a process from accessing memory that has not been allocated to it. Typical section names are '.text', the standard one used for the executable section containing the entry point, or '.data', the section containing read/write (RW) data. Figure 2.1 gives an insight about how headers and sections are organized in the PE file format.

FIGURE 2.1: simple view of the PE file format



2.1.1 Process creation.

Windows OS provides some useful APIs (Application Program Interfaces) to spawn a process. The most famous is *CreateProcess* and consists of some steps which are carried out by three components of Windows OS:

1. Kernel32.dll, a popular client-side library
2. Windows Executive, a kernel-mode service
3. Windows Subsystem Process, the user mode part dedicated to handle the I/O

When the *CreateProcess* is invoked, the image file is accessed at user-level by some procedures collected in the Kernel32.dll library. Once the image is open, the Windows Executive process is triggered by the process invoking the *CreateProcess* (it is a sort of system call) in order to allocate the address space and other data structures necessary for the OS to manage the process in the correct way. During this phase the fields and the previously described sections are parsed and reported in such data structures. Once the file is mapped into memory the main thread is started, beginning from the entry point.

2.2 Malware Analysis: uses and practices

Approaches to malware analysis

The arsenal of a malware analyst includes several techniques by means of which he can understand how a malicious sample work. We can group such techniques into three main categories:

1. Static analysis
2. Dynamic analysis
3. Reverse engineering

Static analysis is the practice of observing the malware without executing it ([30], [35]). In order to statically analyse a sample we can look at different values and metrics which are contained in the image file (i.e., in the PE headers and sections) or that, in some way, can help us to determine whether a file is malicious or not (such as hashes or recognition by other AV detection tools). Such values can indicate if an anomaly is present in the executable under analysis w.r.t. a legal software. The positive aspect of the *static analysis* is the performance: basically it only needs to parse or read some values from a stream of bytes and create some abstract data structures. This can be done for a large set of samples in a short period of time. The negative aspect is that malware authors can use obfuscation techniques to handle these values, masquerading them in a proper way.

This bring us to introduce a second technique: *dynamic analysis* ([35], [22], [13]). It consists of running the malicious sample typically in an isolated environment with the aim to detect some malicious interesting behaviors. For instance, analyst could be interested in examining the network traffic or the files which are modified by the sample. Of course, the environment which hosts the execution of the malware must be isolated to prevent malware from spread through other machines. We can choose among different kinds of so-called virtual environments (i.e., environments which emulate a physical machine with all the features of a real machine such as an OS, I/O devices, HW, etc. [23]). The advantage of *dynamic analysis* over *static analysis* is immediate: it allow us to directly observe what the malware is doing. The main disadvantage is that *dynamic analysis* introduce a remarkable lowering of performance because it requires a great amount of resources (RAM and CPU). Moreover, if dynamic analysis is carried out within a virtual environment, malware can do heuristic checks to detect if its execution is virtualized, changing its behavior accordingly to cheat the analysis tools.

As just explained, in some cases, both *static* and *dynamic analysis* cannot help us to answer our questions. In such cases the only road that can be followed is making use of *reverse engineering* ([31]) which means manually analyze the sample, taking advantage of some dedicated tools that we can define as *reverse engineering frameworks*. This is a complex and slow process which is strictly influenced by the personal abilities of the analyst and by the effectiveness of the tools employed. In general, *reverse engineering* means the analysis process by which it is possible to decompose an object or a product in order to understand how it was designed and built. Moreover, *reverse engineering* aims to discover the architecture of the object under analysis. In general, we can apply *reverse engineering* in different fields such as mechanical engineering, electronic engineering, chemical engineering and of course, software engineering. In such case, the object which has to be analysed and decomposed is a binary file, and there are some useful tools which have to be considered for this purpose.

A *disassembler* is a tool which takes as input a binary file (basically a stream of bytes) and produces as output the instructions in assembly language representing the input binary file.

A *decompiler* is a tool which takes as input a binary file, but differently from disassemblers, attempts to reconstruct the source code which can be easily read by humans.

Disassemblers and *Decompilers* are fundamental tools in the process of reverse engineering of software but they can also be improved. New trends in *reverse engineering* tend to adopt advanced *reverse engineering frameworks* ([6]), [1]). Such tools include not only a robust disassembly engine, but also a lot of advanced features which are extremely useful for handling binary code (both in malware analysis and vulnerability research of a program). For example, analysts employing [6] or [1] can take advantage of the 'debugger' function, which allow us to observe and modify the running state of a software in execution. In addition to this, lots of other functions are available for user of such frameworks for instance stack analysis, scripting interface, graph generation etc.

However, malware authors can introduce more robust obfuscation attacks so that disassemblers can not extract the real assembly code: in such situation, a malware analyst has to face a wrong assembly code which is unreadable and this increases significantly the complexity of the process of reverse engineering.

How an AV software work?

Now that we have some insights about the analysis methods which can be performed over an incoming (potentially unknown and unwanted) program, it is interesting to describe how AV software work and which are their weaknesses. AV software vendors organize their products through three analysis stages:

1. Signature based stage
2. Machine learning stage
3. Dynamic analysis stage

During the first stage, the AV software computes the signature of the executable under analysis to discover if such file has already been analysed by comparing it with a database of previously analysed malware signatures. This is a fast check which does not introduce overhead in the analysis process. The signature of an executable file can be computed according two approaches: the old approach, uses the hash of the file (MD5, SHA1, etc.), as signature of itself. Nowadays, thousands of malware variants are released everyday with a different hash and this is sufficient to bypass the first stage. A slight difference in the file (for instance the introduction of some 'nop' operations in the source code) can result in a different hash, bringing to the evasion of the first stage. Because of this reason, the currently used approach is based on the so called 'YARA' rules: it consists in scanning file and searching for some patterns of byte which are considered distinctive for that malware family. YARA allow us to create descriptions of malware families (or whatever you want to describe) based on textual or binary patterns. Each description, a.k.a rule, consists of a set of strings and a boolean expression which determine its logic. Nevertheless, this solution does not completely solve the problem because malware writers invented polymorphic malware (they will be discussed in next section) which properly mask these patterns just to evade the YARA rules.

The second stage, always based on static analysis, tries to solve the problem of the analysis of an unknown sample by looking at some features contained in the PE headers and sections. Taking advantage of some machine learning algorithms, it is possible to make a classification between malware and goodware. But also in this case, malware authors can bypass this check by obfuscating the PE file.

The third and final stage uses dynamic analysis to detect some suspicious behaviors through the use of heuristic rules. Obviously this phase is quite time-consuming given that it needs to set up an isolated environment (typically called 'sandbox') inside which the file can be dynamically analysed looking at libraries it calls, the actions it performs, whether it tries to hide itself, and/or if it makes registry entries. Moreover, we must consider that AV software is usually produced to run over desktop machines with strong performance and usability requirements: the final outcome is that also in this case a malware author can overcome this checks by adopting some advanced techniques based on the idea of hiding the behavior of the original program in a stream of benign behavior of a large number of processes.

The aforesaid considerations about the actions carried out by an AV software are important to point out a fact: a real war between AV vendors and malware authors exists and when the first ones introduce new analysis methods to detect some anomalies the second ones think up a way to bypass such new analysis. In this race, malware authors are always one step ahead.

2.3 Obfuscation and Packing

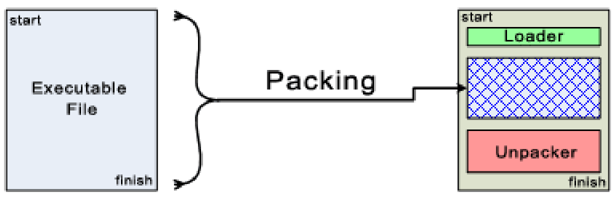
As explained in the previous sections, malware writers are used to contrive new and sophisticated techniques to bypass static analysis and reverse engineering. In the context of malware analysis, *obfuscation* is the technique "to camouflage the telltale signs of malware, undermine antimalware software, and thwart malware analysis" ([32]). The term *obfuscation* involves different practices with the common aim of bypassing AV scanners. The most common examples of *obfuscation* are:

1. packing (more details in the following subsection)
2. polymorphism: the technique through which static binary code is mutated just to evade detection by signature scanners (note that the run time code is kept invariated)
3. metamorphism: the technique through which each time a malware runs it changes the operational code which is loaded in memory

Packing is the most popular form of obfuscation [15]; every day malware writers adopt this technique with the addition of further refinements. Since that, in our research we deal with new interesting trends used by malware with regard to *packing*, we dedicate the following section to this topic which is quite fundamental for the correct comprehension of the core of the work.

2.3.1 Focusing on packing

FIGURE 2.2: A generic schema of packing



A packer is a software through which it is possible to transform the original malicious code by applying compression or encryption algorithms (from here on, we will refer to the terms transformation and encryption in the same way, since compression can be seen as a form of encryption). Let's call 'I', the executable file passed as input to the packer and let's call 'O' the output executable generated by the packer. After the use of a packer, the executable 'O' will contain a stream of bytes representing the encryption of 'I' (the blue area in Fig. 2.2) and a decryption routine 'd' also called 'unpacker' (the red area in Fig. 2.2). When 'O' executes on the infected machine, the routine 'd' will be invoked to restore the original code 'I'. At this point a transition to the so called *Original Entry Point* (OEP) will be triggered, going to execute the original malicious code 'I'.

More in detail, once the packed executable is launched, it starts decrypting the packed code and it writes the result of the decryption routine into a memory area. Such memory area must be both writable and executable (i.e., it will exhibit 'W' and 'X' flags) because it will contain the malicious code which will be invoked at the end of the unpacking procedure. Different strategies can be used to handle and allocate the memory area which will host the unpacked code:

- A possible way is creating an empty section in the PE file produced as output. The section will be writable and executable (and sometimes also readable) and will be used as the region where to write the unpacked code. The section used does not have to be necessarily empty: in some cases, it is possible that the unpacked code overwrites a section which partially contains code, but also in this case the section must show the flags 'W' and 'X'. When we see this strategy we can speak of memory area *statically* allocated.
- A second way consists of taking advantage of a memory area *dynamically* allocated. Windows OS makes available a lot of APIs for the memory allocation and management. A typical schema used by packers is based on the following APIs: 'VirtualAlloc' and 'VirtualProtect'. The first one allows us to allocate virtual memory, while the second one can be used to change the memory protection flags according to need. The typical use of a malware writer is starting by allocating a memory area with 'VirtualAlloc'. Then copy the packed code into the dynamically allocated space. A this point the memory protections will be changed in order to make the new allocated buffer writable and executable and finally the unpacking procedure will overwrite such area with the original malicious code.
- A variant of the second strategy is to allocate a memory area over the heap or the stack of the process. This can be done with the use of appropriate APIs (e.g. 'HeapAlloc') and requires that the memory protections ot the heap (or of the stack) is changed just to make the heap (or the stack) writable and executable.

Regardless of which strategy is used, we can group all the packers in two main categories:

1. Off-the-shelf packers: third-party software distributed in order to protect executable files
2. Custom packers: software created by malware authors exclusively for their purposes

At the beginning, off-the-shelf packers were created to protect benign software or to help the distribution of these ones over the web (because several off-the-shelf packers employ a compression algorithm, reducing the size of the executable to distribute). In a second time, malware authors found that the adoption of them could increase the probability of evasion of their malware and this made diffuse off-the-shelf packers in the arsenal of malware writers. Some typical examples of such packers are: UPX ([2]), Armadillo ([3]), ASPack ([4]). With the aim of enhance the complexity of the analysis, malware writers started to create their own packers (the so called custom packers), which cannot be found on the web. The only way to understand how a custom packer work is to directly study it from a sample packed with it. In general the strategies are those ones presented before for both custom and off-the-shelf packers, but the encryption algorithm can largely vary for each packer.

Some further degrees of complexity can be added: for instance a packer can encrypt the initial code with more than one layer of encryption, or maybe the unpacking code can decrypt only a block of instructions at a time and then executing it ([44]).

Now, we can understand why packing can often be crucial for the evasion of a malware from AV scanners. When an AV software encounters a new file, never seen before, it begins to analyse the features to detect some anomalies. Unfortunately no anomalies can be detected statically because code is encrypted and it does not reveal any malicious intent. In most of cases, if the packer is unknown, the file will be accepted because it does not exhibit any problem, allowing the malware to infect the attacked system. Further checks have been introduced just to detect known packers: in facts the signature strategy can be recycled in this scope (e.g. PEid tool [5]). But what happens if a packer with an unknown signature is used? Different approaches have been developed just to solve this question. They face the problem leaning on machine learning and data mining algorithms. According to these theories, it is possible to collect a set of features which are strong indicators of the fact that a malware is packed or not. If this is the case, a successive step is required in order to perform the unpacking so that the file can be correctly analysed. Note that, also if a packed sample is detected thanks to machine learning, that does not reveal which packer has been used: the unpacking procedure is really complex and time-consuming because it needs to try different strategies for the unpacking without making any assumption of how the packer works.

2.4 Frameworks and tools employed

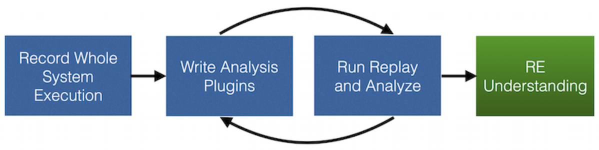
To lead the experiments, we will explain in the central part of this work we decided to lean on some robust frameworks which helped us in our analysis process. The tools we intend to show in this section are mainly two:

- PANDA ([21])
- Volatility ([9])

PANDA

PANDA is the acronym of Platform for Architecture-Neutral Dynamic Analysis. It consists of a system, built on top of QEMU, which allow leading dynamic analysis and instrumentation of code. QEMU ([14]) is an open source machine emulator and virtualizer. In our experiments, we configured QEMU to emulate a Windows 7 environment with 32-bit architecture. Once set up the virtual environment with QEMU, PANDA includes a system for recording and replaying the execution of one or more binaries over such environment. When the record of an execution is captured, it is possible to analyse it through the so-called "plugins". The typical workflow in PANDA is to collect a recording of a piece of execution of interest and then analyze such recording over and over again with the aforesaid plugins.

FIGURE 2.3: Ideal PANDA workflow



Plugins are pieces of code written in C/C++ which implement some specific analysis. By default, the installation of PANDA provides some useful plugins for standard analysis like network monitoring, looking for some particular strings used by the binary, tainting some files or other interesting things. Moreover, you can write your plugin using a dedicated PANDA interface. The interface offered by PANDA involves several callbacks which can be used to instrument some significant moments in the process of execution of a binary. For example, a very useful callback is "`PANDA_CB_INSN_EXEC`" which is invoked for each time that an instruction in the record is executed. The author of the plugin can use this callback to extract the opcode of such instruction or to investigate the process which is currently running. Other very interesting callbacks are the memory callbacks like "`PANDA_CB_VIRT_MEM_AFTER_WRITE`" which is activated after a process performs a write on virtual memory. Of course, the use of a callback instead of another depends from the intent of the author. To sum up, a plugin is a list of registered callbacks, and for each callback, the plugin author has to write what actions to do when the event connected to such callback happens.

Volatility

The Volatility Framework is a collection of tools for the extraction of digital artifacts from volatile memory (RAM) samples. It supports analysis for Linux, Windows, Mac, and Android systems. Memory dumps, crash dumps, VMware dumps (.vmem), virtual box dumps and QEMU dumps can be studied with the help of volatility which is based on python. The idea behind the volatility workflow is similar to the previously described PANDA: it makes available some plugins which can be easily extended. When you write your volatility plugin you can access a lot of interfaces to interact with the dump under observation. For example, it is possible to extract the list of the processes which were active in that dump by merely calling the function "`pslist()`". To correctly run, Volatility needs to set up a 'volatility profile', which will indicate the OS specifics we want to investigate (data structures to handle processes, threads, etc.)

Chapter 3

Low Entropy Packed Malware Schemes and Spread

3.1 Entropy of Executable files

Shannon entropy H of a discrete random event x is given by the formula:

$$\mathrm{H}(x) = -\sum_{i=1}^{n} \mathrm{P}(x_i) \log_2(\mathrm{P}(x_i))$$

where $\mathrm{P}(x_i)$ is the probability of the i^{th} unit of information (such as a number) in event x's series of n symbols. This formula generates entropy scores as real numbers; when there are n = 256 possibilities (e.g. 1 Byte), they are bounded within the range of 0 to 8.

$$\mathrm{n} = 256 \implies 0 \leq \mathrm{H}(x) \leq 8$$

The entropy is a metric for measuring uncertainty in a series of numbers (or bytes), that is, it measures the difficulty level of independently predicting each number in the series. The difficulty in predicting successive numbers can increase or decrease depending on the amount of information the predictor has about the function that generated the numbers, and any information retained about the prior numbers in the series.

If we consider *lossless compression* and *encryption* functions it is trivial to prove that they generates high entropy data. Lossless compression functions, initially, generate a statistical model for the input data, then use such model to map input data to bit sequences in a way that frequently encountered data will produce a shorter output than infrequent data; therefore they remove predictability, which implies high entropy.

Instead, encryption functions are directly designed to generate unpredictable data which, again, has high entropy by definition.

Therefore, despite generating a *packed executable* is an obfuscation process that compresses or encrypts data, it is obvious why entropy became an important metric in this context. As a consequence of this, the first authors who dealt with classification of *packed* or *not packed* executables considered the entropy of the entire file to discriminate between the two classes of executables [29].

The most used file formats for executables divide the file into sections, which is a way to organize data and provide the operating system with permission rules (Read-Write-Execute) on such data. E.g.: in PE files (which are briefly described in 2.1), code instructions are usually placed in the `.text` section that is just readable and executable, while read-only data goes to `.rdata` that is just readable. This way of partitioning an executable file is very important for the entropy. If we consider the `.text` entropy, we expect to find a lot of redundant bytes, therefore a low entropy. The reasons are two: i) independently from the target instruction set architecture, the bytes of the instruction encoding are more frequent ii) certain instructions occur more frequently than others. Therefore, considering the entropy of the whole file does not take into account that packed code could be placed only in one section, leading to an average entropy value that it is not suspicious. For this reason, a more precise approach was proposed in [24], where the entropy was evaluated at a granularity of single sections. Accordingly, the granularity of the entropy computation was increased ([33, 45]), considering n-grams (a contiguous sequence of n items).

Even if increasing the granularity of the entropy computation can help to individuate packed code, attackers started to introduce some countermeasures. This arms race began when malware writers introduced byte appending schemes (described later in Section 3), which allowed them to circumvent all the analysis approaches at the state of the art (i.e., based on the entropy of the whole file, of each section and of n-grams). However, LEPM were not so spread to constitute a problem, and they were not considered for several years. Nowadays, LEPM represents the evolution of the first samples making use of byte appending. As we will show, for this kind of malware, the increasing granularity is not useful because LEPM adopts proper schemes to generate low entropy byte sequences, independently from granularity.

When the first samples belonging to the *Zeus* malware family appeared, they contained trivial countermeasure to avoid entropy checks. They basically made use of byte appending which means inserting the same byte (or subset of bytes) repeated for a certain number of times so that entropy decreases [46]. Different techniques based on encoding were hypothesized in [10] but nobody was able to recognize them in the wild. However new more complex schemes have been introduced in the last years; in this section we give some insights on them and we discuss the structure of the framework we built accordingly for the detection of LEPM. Then we show the schemes which are used by LEPM just to keep low the entropy level and we provide some insights on the heuristics used to detect such schemes. Finally, we focus on how to build statistically meaningful dataset of LEPM and how we carried out our analysis in order to 1) study the spread of LEPM, 2) shed light on their frequency and, 3) point out how important they must be considered when classifying packed and not packed executables for malware detection.

3.1.1 Analysis

In Section 5 we discussed the drawbacks of static analysis for detecting LEPM. To overcome such limitations, we have designed a dynamic analysis tool which allow to recognize the packing scheme used by LEPM and to measure the LEPM spread inside a given dataset. It is important to notice that LEPM authors use the first layer just to keep entropy low and then, in case they want to make the encryption more robust, they can use other layers with stronger encryption schemes. Therefore, it would not make sense if a sample uses a generic encryption scheme as the first layer and a low entropy scheme as the successive layer.

The architecture of our dynamic analysis tool, depicted in Fig. 3.1, is composed by four modules:

Packer Detector is based on PANDA framework [21] and it detects if a malware is packed or not. Firstly it tracks the write accesses on virtual memory thanks to the callback "`PANDA_CB_VIRT_MEM_AFTER_WRITE`". Such callback allows to access the address of the write operation, which is stored into an proper data structure (i.e., a list of addresses). Then it detects when a write address (belonging to the list of the write addresses) is executed. This is carried out by the callback named "`PANDA_CB_INSN_TRANSLATE`" which provides some useful information like the program counter (PC) of the currently executed instruction. If such PC matches with a write address, this means that an unpacking operation has been performed. Further checks are added to exclude those samples writing over a legitimate DLL area (see the following paragraphs).

Packer Verifier checks the results of the Packer Detector with the tool developed in [44] which currently represents the state of art with regard to the detection of unpacking behaviors.

Schema Classifier, always based on PANDA [21], it reconstructs the unpacking scheme and the operations which are performed during the initial unpacking phase.It takes as first input the output of the *Packer Detector* which consists of a file comprising the addresses of the memory regions modified by the unpacking procedure. In addition to this, the second input is a LEPM because of the dataset construction constraints (all the entropy sections minor than 7.0, see 3.2.1) and the double verification that is packed carried out during the previous steps thanks to the *Packer Detector* and the *Packer Verifier*. The *Schema Classifier* is the final component of our analysis framework which allow us to define the rules needed to recognize the schemes used for carrying out unpacking. In facts, it implements several heuristics which are described for each of the detected schemes in the subsection 3.2. Internally, it relies on two PANDA callbacks, "`PANDA_CB_INSN_EXEC`" and "`PANDA_CB_VIRT_MEM_AFTER_READ`". The first one is used to detect and analyse the operations which are executed before the write on virtual memory. The second one is invoked just to analyse the data which are read from memory, transformed according to some encryption scheme, written into memory and finally executed. In a few words, with the second callback it is possible to reconstruct (in some cases), the initial buffer from which encrypted code is taken. As you will see, this is really useful for the precise detection of some schemes.

False Positive Classifier divides the false positives in two categories:

1. samples writing to a module memory area (e.g., when a legitimate DLL is loaded at runtime and then the DLL is executed). As some samples implement custom loaders, this behavior could create false positives. To develop this part, a volatility plugin was implemented to analyse the memory of the running process. Windows handles the memory areas of a process with the use of a balanced tree called VAD (Virtual Address Descriptor). As expressed in [20], VAD can be used in forensic analysis of Windows memory. In our case the volatility plugin we wrote deals with walking the VAD of the sample we are analysing, and for each memory area, it detects if it corresponds to stack ('S'), heap ('H'), or module ('M'). If all the writes which have been collected by the PANDA plugin operate over a module area we can conclude that the sample is not doing real unpacking, but only it is loading a DLL;
2. *droppers*: executables that download code and inject it into memory to be executed. Detecting such samples do not require a further effort: it is sufficient to run them inside the *packer detector* without the internet connection. If the *packer detector* produces the same output in both the two cases (with and without the internet connection) we can affirm that packing operations were indeed detected;

We entirely developed the source code of the tool except for the *Packer Verifier* which has been used to have ground truth about the packer adoption of a sample.

3.2 Schemes Taxonomy

Low entropy schemes we observed in the wild can be divided into two main categories:

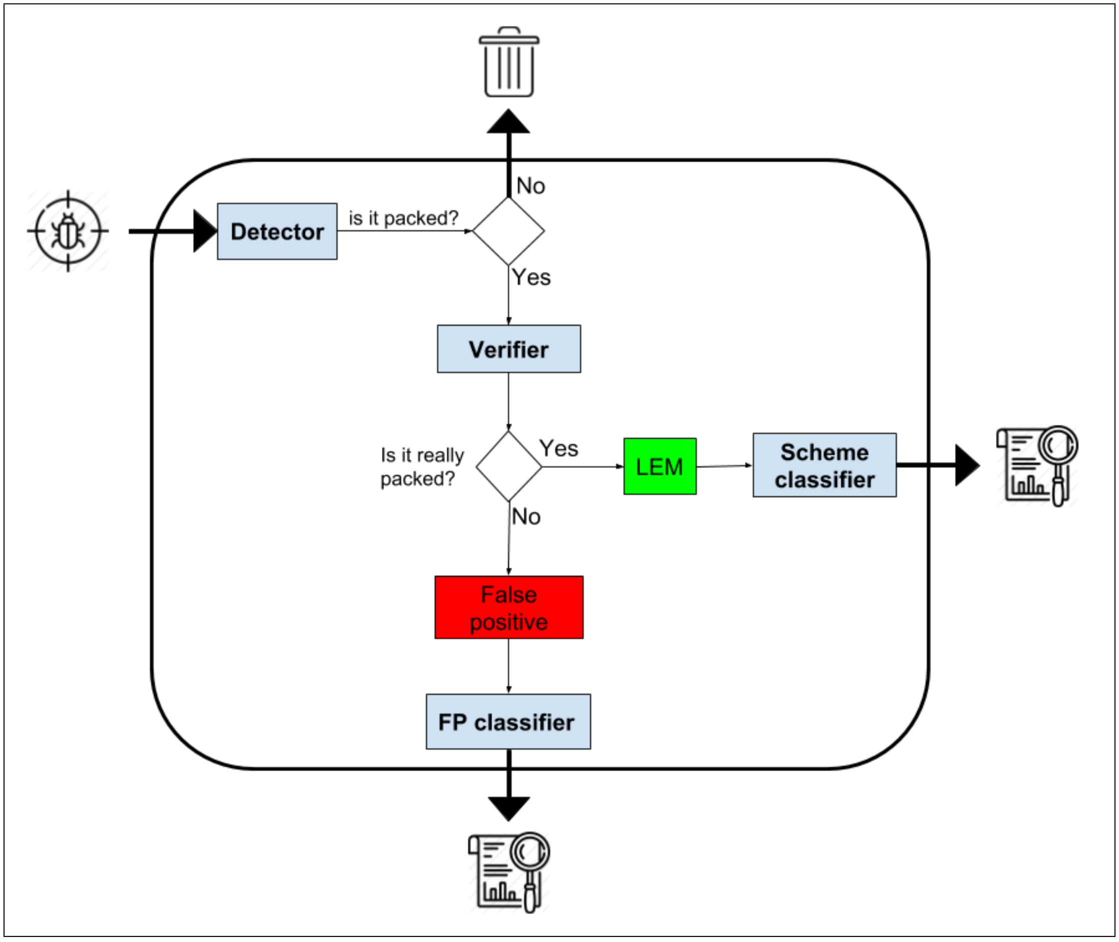


FIGURE 3.1: Dynamic Analysis Tool

- Entropy decreasing schemes (EDS)
- Entropy constant schemes (ECS)

On the one hand, schemes belonging to the EDS category can be used as a first layer when the malware author wants to introduce a successive more robust encryption. On the other hand, the ECS give the possibility to transform the code keeping the entropy at an almost constant level. For both the two categories we detected different custom packers which are based on the following principles:

- byte appending (EDS)
- xor/add encryption (ECS)
- encoding (EDS)
- custom encoding (EDS)
- transposition (ECS)

Byte appending schemes consist of appending the same small subset of bytes with repetitiveness so that they create a pattern making entropy decrease. We found samples using pure byte appending and other ones taking advantage of this idea combined with other schemes. This simple scheme is detected by our tool by analysing all the memory read and observing the distribution of the bytes. If *byte appending* is used we will notice that the frequency of the bytes used to decrease entropy is higher than the other bytes.

Xor/add encryptions are the most popular schemes among LEPM. In those schemes, the idea behind them is using simple operations, like xor or add, that do not modify the byte distribution. This objective can be achieved only if the key respects some constraints that we empirically obtained by carrying out some tests to have evidence about how these transformations affect entropy. We extracted the section *.text*, or more generally, the section containing unencrypted code from a set of 100 benign PE files and we applied different encryptions (with several configurations and modes of operation) to each of the extracted sections. We plotted two graphs in order to understand how the key size influences the entropy: in Fig. 3.2 and 3.3 we represented on the X axis the number of bytes of the key (in range 1, 32) and on the Y axis the entropy level after the encryption with the corresponding key.

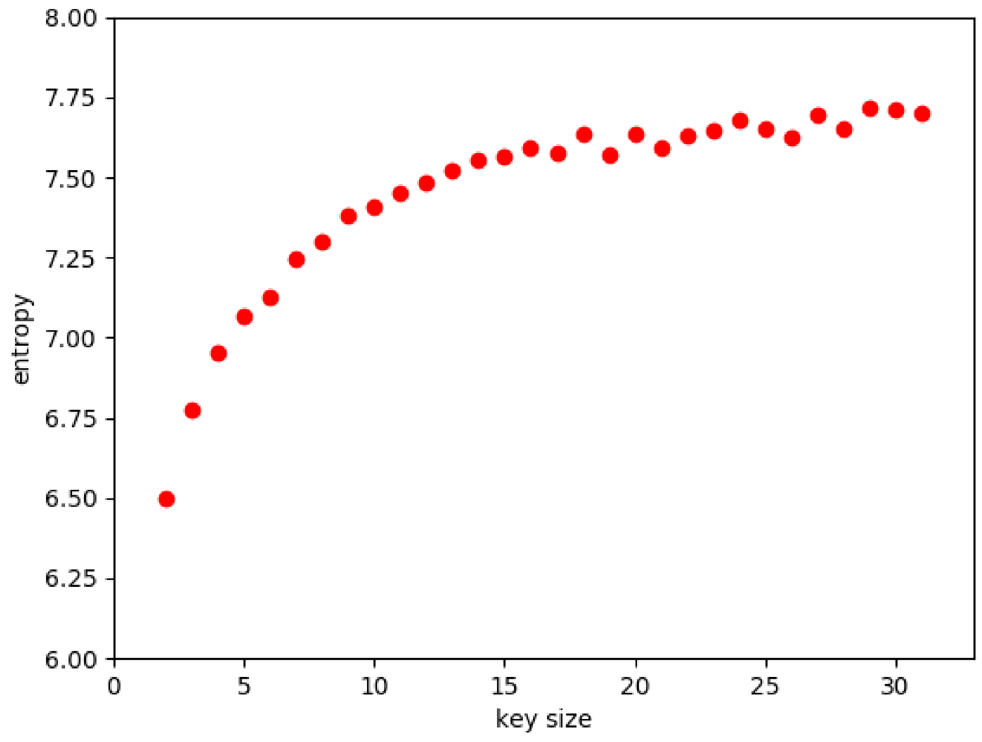


FIGURE 3.2: xor encryption plot

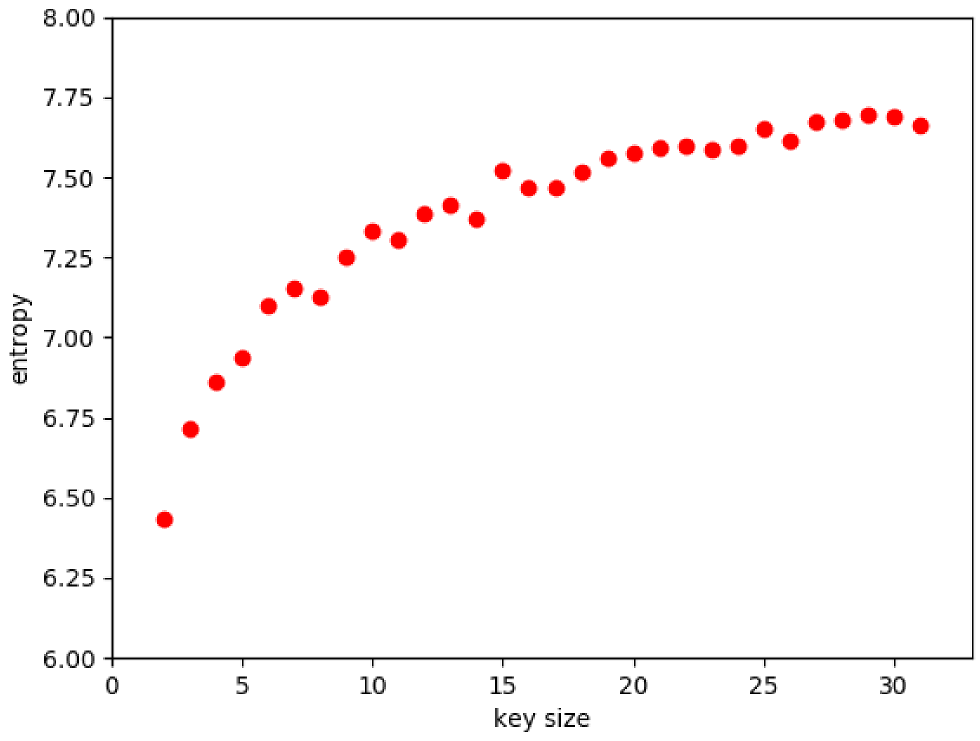


FIGURE 3.3: add encryption plot

After those tests we found that the length of the key influences the entropy level according the following rule:

$$length(key) < 4 \implies H < 7.0$$

From this rule we obtain that the attacker can choose between using a 1-byte key that does not affect entropy of the file, or using a key of higher size which allows to keep the entropy sufficiently low to circumvent the checks. In this case the *schema classifier* works as follows: first of all it disassembles and analyses the operations which are performed before the memory write (we know when the write is executed thanks to the *packer detector*). In this case PANDA allow us to use a famous engine for disassembly named *capstone*. With the help of this library we can disassemble the instructions which are captured by the callback "`PANDA_CB_INSN_EXEC`". If it finds that some interesting operations like ADD, XOR or SUB are executed, it tries to detect whether an encryption key is used. For instance, if before every memory write, the value contained in a register is XORed with a 2-bytes fixed value, this means that the sample is making use of a XOR encryption with a 2-bytes key. Of course, some false positives must be discarded (for instance when XOR is used to set to zero a register's value). In general, malware authors can take advantage of other instructions which can be used as an encryption operation. ADD, and XOR are for sure the most frequent, but we discovered that SUB, ROL, and ROR are also valid alternatives which can be observed in some isolated cases in wild.

Encoding schemes are based on the idea of using a set of symbols to represent information. An encoding schema can be formally defined as a source alphabet S, with cardinality $|S|$, a destination alphabet D with cardinality $|D|$, and a bijective encoding function $enc : S^* \longrightarrow D^*$. An attacker has to find an encoding schema suitable for its S, that is, the entropy of a string $s \in S^*$ must be greater than the entropy of the string s encoded into $d \in D^*$, so, $H(s) > H(d)$. In real cases we mainly observed base64 encodings, but we cannot exclude that new samples could use use different ones such as base32 or base16. Here, the heuristic used by the *schema classifier* adopts a different strategy: it creates an output file containing the memory reads values coming from buffers having size which is suitable to contain a portion of packed code. For each buffer a python script detects if some encodings are used on the basis of the symbols which are present in the considered buffer. A successive degree of complexity can be added using custom encoding schemes, in which the attacker creates the function *enc* and the alphabet D .

Custom encoding mixes traditional encoding with substitution ciphers, i.e., encoded symbol is encrypted using a key. This schema allows malware writers to reach a stronger encryption level and low levels of entropy at the same time. For this reason, this is the most challenging schema to detect. Indeed, while for usual encoding we observe the memory searching for patterns (for instance an high percentage of bytes in a certain range), for custom encoding the use of a key makes the pattern more difficult to detect, because in this case we do not know which kind of pattern to look for. So, in this case we are not able to define exactly the key which is used but we can only use the same output file described in the previous paragraph related to *encoding* and then to search if the buffer respects some rules. In facts, even if a custom encoding is used, it is derived from the application of a traditional encoding with the addition of a substitution key. Anyway, this kind of transformation does not affect entropy of that buffer, so a first rule is obtained by computing entropy of that memory region and comparing it with the typical entropy values of the traditional encodings. A second temptative consists of observing that, also in custom encoded buffers, probably we will have that some bytes (those ones that represent the destination alphabet) will be more frequent that the other ones.

For *custom encoding*, entropy is low because it is used just a reduced set of bytes compared to the usual alphabet, i.e., $|S| > |D|$, making the *enc* function compliant with entropy constraints.

Transposition scheme obtains the ciphertext from a permutation of the plaintext. We found different variants of permutations algorithms; anyway, no transposition scheme influences entropy, as the byte frequency of the plaintext is the same as the ciphertext.

Therefore this scheme is used when the source has already a low entropy but it can be hardened by mixing it with ADD/XOR encryption as explained in Sec. 3.2. To detect this kind of encryption, the *schema classifier* basically compares the read and write buffers: if they contain the the same bytes with the same frequency this probably means that a transposition cipher was used to generate the unpacked memory area.

3.2.1 Dataset

To set up a statistically significant dataset composed by recent malware, we have downloaded from VirusTotal [8] three different sets of Portable Executable [1] (PE from now on) malware, from three different periods of time: [2018-06-06, 2017-12-04], [2018-07-04, 2015-05-02] and [2018-07-23, 2012-10-12]. The groups of malicious samples contains respectively 375, 1121 and 2130 samples, for a total of more than 3500 files.

Our dataset complies with the requirements of:

- sections with $H < 7.0$;
- more than 20 detections on VirusTotal [8]
- overlay [2] data with $H < 7.0$;

Such strict constraints are needed in order to analyse real LEPM.

3.2.2 Distribution and Spread

In Figure 3.4 we report the distribution of LEPM schemes that we observed in our dataset. The following percentages are computed over a dataset made of a total of 3626 samples satisfying the previous constraints.

In Table 3.1, we report the complete output of our tool: it shows that the percentage of LEPM is always remarkable for each set of samples. Respectively, we have 19.73%, 12.57% and 15.49% for the three groups of samples. Moreover, the computed percentages represent a lower bound since that we are not able to distinguish those samples detecting the virtual environment in which they are analysed (in our case QEMU [14]) from the samples discarded as not packed by our tool. Indeed, if a sample can evade the virtual environment, it will not execute correctly, but this does not mean that maybe that sample adopts an unpacking procedure if successfully run.

[1]the Portable Executable format is a file format for executables, object code, and DLLs, used in 32-bit and 64-bit versions of Windows operating system

[2]The overlay is just appended data to the end of the executable file that is ignored when loading an executable into memory because is not covered by the PE header. Opening the executable file for reading will allow access to the entire file including the overlay portion.

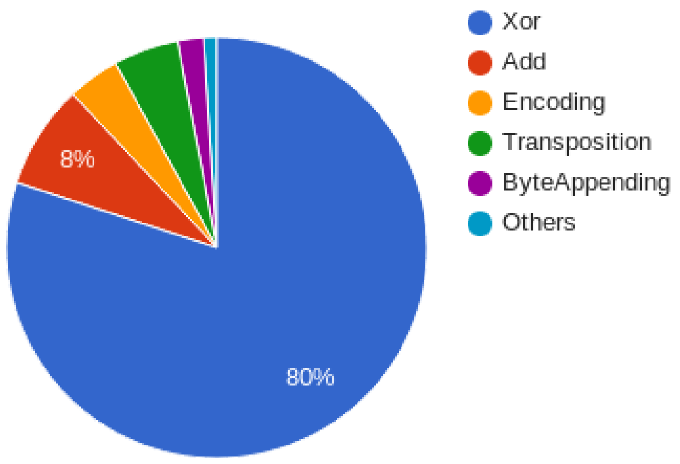


FIGURE 3.4: LEPM schemes distribution

TABLE 3.1: Experiment results

	First Group	Second group	Third group
TOTAL	375	1121	2130
Not packed (Droppers, Dll loaders)	301	980	1800
Official LEPM	74	141	330
XOR based	48	65	262
ADD based:	11	22	26
ENCODING based:	9	29	14
TRANSPOSITION based:	6	13	16
BYTE APPENDING:	0	12	12

Table 3.1 underlines that LEPM are widespread and should be considered while studying packed malware.

Chapter 4

Case studies

In the following sections some remarkable case studies will be discussed, to allow a better understanding on how Low Entropy Packing schemes work. Anyway, it is important to point out that the majority of LEPM we analysed use a *custom packer*. This fact is quite obvious because off-the-shelf packers are often created to protect files, not to make them robust against antivirus detection. Nevertheless, during our analysis, we found a known signature packer, named *Beria* packer that uses a sophisticated scheme. For each case study we show the `sha256` hash code as caption of the table so that the reader can verify the file.

4.1 Case study: the Beria packer

Suppose to call R_b the buffer from which a Beria packed binary reads the packed code and W_b the buffer into which it writes the code after unpacking (i.e., a destination buffer). Analysing the R_b layout, we observe that it is composed by two types of byte: "original" and "metadata". Original bytes are merely copied into W_b, while metadata bytes are evaluated through an algorithm which determines the correct offset (in relation to R_b) where to write the original bytes inside the destination buffer W_b. In this way entropy is kept low for two reasons: i) the original bytes appear with the same randomness of a not packed binary (they are not encrypted, they are just shifted): ii) the metadata bytes are not random bytes, but they have to match some rules according to the evaluation algorithm, so they do not increase randomness of the binary because they follow a specific pattern.

Despite the Beria packer is quite interesting, we did not find any sample using it in the wild so we do not detail the the evaluation algorithm.

4.2 Case study: xor encryption

Table 4.1: sha256=282969984eaca125578a583f5dc15248a9ac305770f0c1f93eb645b3c23922ea

Family	hematite
Section entropy	
.text	6.50
.data	4.97
.rsrc	5.61

This sample executes xor encryption in order to obtain low entropy levels. Let us remember that every sample in our dataset described in Sec. 3.2.1 has all PE sections with $H < 7.0$. We have chosen this sample because it uses three layers of encryption. The unpacking procedure, for each layer, is performed inside the same process. The first layer of encryption maintains entropy low since it is based on XOR encryption. Let's call *Nonce1*, *Nonce2*, *Nonce3* the three constants values used to build the encryption key. The key is generated at run-time using a procedure, named `generateKey()`, having the three nonces as input and producing a one-byte key as result. It is important to notice that the encryption key which is produced by such method is built in a way that it respects all the criteria necessary to maintain a low level of entropy (i.e., basically length minor than 4 bytes). The pseudocode of the first layer is:

```
while addr != final_address_to_unpack do
    b = readByte(addr);
    key = generateKey(Nonce1, Nonce2, Nonce3);
    res = b ⊕ key;
    writeByte(b, addr);
    addr += 1;
end
```

Algorithm 1: Pseudocode of the first layer

Once the first layer of unpacking has been executed the second layer starts using a stronger encryption cipher.

4.3 Case study: custom encoding

We show a sample taking advantage of a custom encoding scheme used as first layer because it allows to use a reduced set of symbols. The second layer will provide a more robust encryption. Two sections are noteworthy: `.rsrc` and `.rdata`. The 75% of the `.rsrc` section is composed of a pattern which is used to generate the encrypted malicious code.

Table 4.3 shows some bytes extracted from the `.rsrc` section showing the pattern. Such pattern is applied for each dword (4 byte). One among the 4 bytes composing the dword

Table 4.2: sha256=320008650befd4d89bae59eb57029064b7695e2ad56278ef583b61d9b8d0438d

Family	emotet
Section entropy	
.text	6.57
.data	2.73
.rdata	5.40
.rsrc:	6.32

Table 4.3: Example of pattern stored inside the `.rsrc` section

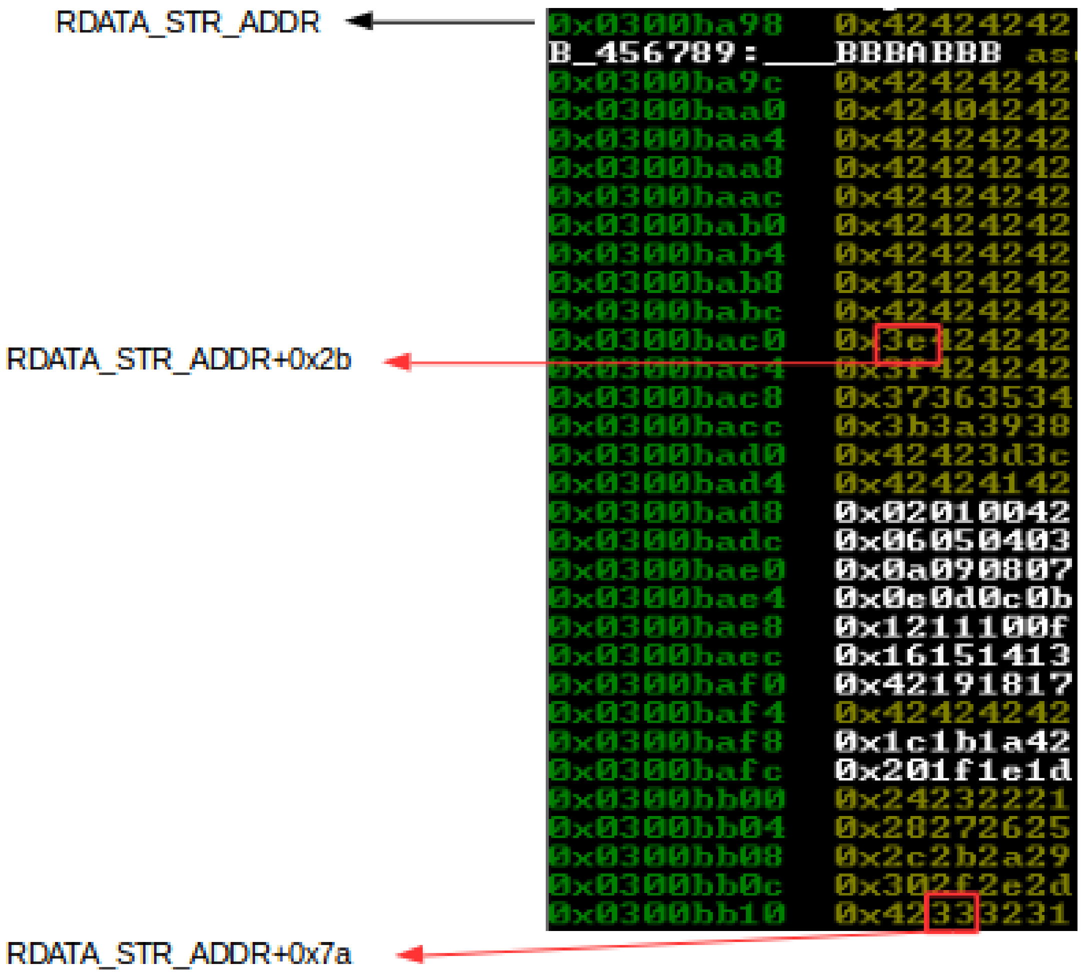
0x	2f	03	35	70
0x	71	03	37	4d
0x	78	02	37	46
0x	69	03	42	45
0x	67	03	53	64
0x	62	03	68	70
0x	4f	02	45	57

(not necessarily the second one, as in the table) is always within the range [0x00, 0x03] (this is an example of byte appending). Each of the remaining 3 bytes contains a value within the range [0x2b, 0x7a]. It is important to remember these last two values because they are fundamental to understand the scheme used by the first layer. It is interesting how this pattern does not increase the entropy level of the `.rsrc` section because of this repetitiveness.

The `.rdata` section contains several strings. Among all the strings stored on this section, one of them is remarkable: it is filled with some characters without a specific meaning (mainly the *B* character), but all the bytes from 0x00 until 0x3f (disposed in random order) appear starting from the offset 0x2b until the offset 0x7a (as Fig. 4.1 points out).

These are the values which appear in the `.rsrc` pattern previously discussed. Fig. 4.1 provide an insight of the string composition. We define `RDATA_STR_ADDR` the starting address of the string (i.e. 0x0300ba98). Pointing to the address `RDATA_STR_ADDR + 0x2b` allow us to discard the non significant bytes (0x42, i.e., the *B* characters) and reach the address 0x0300bac0 where the byte 0x3f is stored. In the same way, `RDATA_STR_ADDR + 0x7a` which contains the byte 0x33 which is the last interesting byte. From here on, a meaningless repetition of bytes is used (the image does not show but from 0x0300bb10 only non significant characters are stored). Now we can understand how the first layer works.

The first layer operates on the previously analysed sections (the `.rsrc` and the `.rdata`). Let's define:



```
0x0300ba98    0x42424242
B_456789:____BBBABBB  as
0x0300ba9c    0x42424242
0x0300baa0    0x42404242
0x0300baa4    0x42424242
0x0300baa8    0x42424242
0x0300baac    0x42424242
0x0300bab0    0x42424242
0x0300bab4    0x42424242
0x0300bab8    0x42424242
0x0300babc    0x42424242
0x0300bac0    0x3e424242
0x0300bac4    0x3f424242
0x0300bac8    0x37363534
0x0300bacc    0x3b3a3938
0x0300bad0    0x42423d3c
0x0300bad4    0x42424142
0x0300bad8    0x02010042
0x0300badc    0x06050403
0x0300bae0    0x0a090807
0x0300bae4    0x0e0d0c0b
0x0300bae8    0x1211100f
0x0300baec    0x16151413
0x0300baf0    0x42191817
0x0300baf4    0x42424242
0x0300baf8    0x1c1b1a42
0x0300bafc    0x201f1e1d
0x0300bb00    0x24232221
0x0300bb04    0x28272625
0x0300bb08    0x2c2b2a29
0x0300bb0c    0x302f2e2d
0x0300bb10    0x42333231
```

FIGURE 4.1: portion of `.rdata` section showing the key string

- `RSRC_START_ADDR` the starting address of the memory area in the `.rsrc` section containing the low entropy pattern;
- `RSRC_END_ADDR` the final address of the memory area in the `.rsrc` section containing the low entropy pattern;
- `RDATA_STR` the string stored in the `.rdata` previously described (we recall that `RDATA_STR_ADDR` is its starting address).

The procedure iterates over all the `.rsrc` area containing the pattern. The values of the pattern map to a certain byte of the string. For instance, if the pattern contains a dword like this: 0x2f033570, this means that the procedure selects the byte at offset 0x2f, 0x03, 0x35, 0x70 w.r.t. the address `RDATA_STR_ADDR`. As we said before, this string contains the character "B" for low values of the offset (for example 0x03) and the values in range [0x00, 0x3f] for offset values between 0x2b and 0x7a. Suppose to represent the string with an array STR[N] where N is dimension. For a certain dword DWORD[4] = BYTE_0, BYTE_1, BYTE_2, BYTE_3 (extracted from the .rsrc area) we will have

```
i = 0;
res = 0;
for addr ← RSRC_START_ADDR to RSRC_END_ADDR do
    offset = readByte(addr);
    byte = readByte(RDATA_STR_ADDR + offset);
    if byte != "B" then
        i = i+ 1;
        res = res ¡¡ 6;
        res = res + byte;
    end
    if i == 4 then
        writeToMemory(res);
        res = 0;
        i = 0;
    end
end
```

Algorithm 2: Pseudocode of the first layer

that the procedure extracts, in order: STR[BYTE_0], STR[BYTE_1], STR[BYTE_2] and finally STR[BYTE_3]. Now, if BYTE_i belongs to range [0x2b; 0x7a], it matches with an *interesting value*, , i.e., a byte in range [0x00; 0x3f]. Otherwise, BYTE_i belongs to the set 0x00, 0x01, 0x02, 0x03 and therefore it matches with a “B”.
Therefore, because of the structure of the pattern, when a byte corresponding to B is read, then it is discarded, while, for other read values the outcome is first shifted by 6 positions and then it is added to the new read value.

In addition to the custom encoding scheme in this case you can see a byte appending scheme used to further reduce the entropy degree. This is due to the fact that the second layer uses a more robust encryption scheme based on a 16 byte key, which translates into an high entropy ciphertext.

4.4 Case study: transposition cipher

TABLE 4.4: sha256=511b622eaee750c5440a59ae588433178e0f44a1764b80ee092da4ee12646189

Family	nitol
Section entropy	
.text	6.42
.data	4.05
.rdata	4.93
.rsrc:	2.81

In this case we focus on a different scheme based on a transposition cipher. The reason why this kind of encryption is applied is that it preserves the byte distribution. The bytes composing the plaintext and the ciphertext are the same, but they are arranged in a different position and this keeps entropy at the same level of the original binary. The offset of the bytes which must be exchanged is chosen through a dedicated procedure described in pseudocode list in Fig. 3.

```
while True do
    offset1 = getOffset(addr1);
    offset2 = getOffset(addr2);
    swap(offset1, offset2)
end
```

Algorithm 3: Pseudocode of the transposition scheme

It is necessary to underline that a transposition scheme cannot decrease entropy, but it only keeps the entropy unchanged, because the bytes before and after the encryption are the same. This means that it cannot be used as the first layer if the second one is a strong encryption algorithm which increases entropy. Anyway, it offers good coverage in combination with a low entropy substitution cipher such as xor/add encryption.

Chapter 5

Static analysis approaches

In this section we offer a survey of the state of the art with regard to the existent approaches and theories about the classification between packed and not packed binaries. The aim is to give an idea of the fact that, even if a lot of efforts has been spent in order to solve this problem, LEPM can easily bypass these techniques, thereby asking for new methodologies of detection.

5.1 Related work on static analysis

Authors of [29] were the first who considered entropy in malware analysis as an interesting metric to perform the classification between packed and not packed malware. The simple idea was that packed/encrypted malware reach higher entropy level w.r.t. not packed malware.

Then, authors in [18] proposed a machine learning-based approach in which they takes into consideration eight features extracted from Portable Executable files. This reduced set of features is related to the sections protection and their names so LEPM are able to easily bypass the analysis.

An improvement of [18] is discussed in [24]: this research tries to minimise the processing time by choosing only few essential features related to the characteristics of sections. In addition to this, entropy analysis is used to perform the classification between packed and not packed malware.

A more consistent set of features is collected in [41] and [34]; where the focus in on basic features related to the PE structure and the entropy levels which can be processed by machine learning techniques. The problem of this approach is that it is not effective against LEPM, which keep low entropy levels and often do not exhibit remarkable changes in the PE file header structure.

Further improvements are suggested in [33] which mixes the heuristic features taken by [34] with *n-gram* analysis of both the code section and the entire PE file.

A more sophisticated approach was proposed in [40]. In this case, features are weighted by an Information Gain (IG) and only features having a relevant IG (i.e., higher than a certain level) are taken into account. Also in this case, the collected features are connected to the PE headers and the entropy levels which satisfy a given IG value. The consequence is that the features extracted by [40] are a superset of those proposed by [34]. Despite the larger set of attributes to process, and a rigorous methodology to measure the importance of them, the approach does not work correctly on LEPM. This is due to the fact that the computation of the IG parameter is based on a dataset containing general packed malware and non-packed malware so it does not consider specifically LEPM. Furthermore, the extracted attributes are always connected to PE headers and entropy levels, while LEPM do not affect these values in an evident way, as from the point of view of the PE structure, the LEPM tend to behave like benign executables.

Further improvement is provided in [45] where structural and heuristic features are used but these are added to the collection of n-grams in order to perform an operational code frequency based analysis.

Recent proposals in [11] and [12] focus on the entropy variations that are due to the unpacking procedure (i.e., when a sample begins to unpack, the written area will exhibit an entropy variation). The main limitation of such approaches is that the experiments which are carried out in such studies tend to consider mainly off-the-shelf packers and that both [11] and [12] are essentially dynamic approaches and consequently they are subject to performance problems of dynamic analysis.

A lot of researches try to distinguish between packed or non-packed binaries, making strong assumptions on how the packer works. In this case it is possible to distinguish between two main kinds of packer: *off-the-shelf* packers (also used by benign binaries) and *custom* packers. For example, in [19] authors have studied an off-the-shelf packer, i.e., the UPX packer,by gathering a set of features which are typical of UPX packer. In the same way, [42] developed a methodology for the creation of a randomness packer signature, but also in this paper only a set of *off-the-shelf* packers is chosen for the experiments. Since LEPM typically use custom packers, such work is not useful for our aim.

To the best of our knowledge, the research proposed in [46] is the most reliable, albeit as it specifically studies malware adopting custom packer dedicated to keep low entropy levels. In detail, Zeus family is chosen as sample to constitute the dataset of the malware. The approach proposed in [46] works properly if the custom packer use the following two methods for decreasing the entropy:

- Random byte insertion
- Reduced source alphabet

The idea is to divide the binary into overlapping regions (with size between 128 and 512 bytes), compute the entropy for each region and produce an entropy surface. Then, authors compare such surface value with a given threshold. The two schemes taken into account are used by some real-world malware samples, but they do not represent all the available schemes for a malware writer. For instance, also a basic xor encryption would lead to decrease the entropy level. Finally, this approach does not represent a definitive solution but a good starting point in the problem of distinguishing packed and non-packed malware. The only other work considering the problem represented by LEPM is [10] which tries to make suppositions about how a malware could evade the entropy checks performed by AV software. Authors of this study actually consider the possibility that a sample could use encoding just to reduce entropy and different encoding algorithms are proposed, but without any demonstration or evidence of the real adoption of these schemes in the wild.

Other proposals try to understand how malware bypass the different checks performed by AV products. Obfuscation involve not only packing but also anti-disassembly tricks such as *opaque predicates*. An approach that deals with different kinds of obfuscation (e.g., packing, encryption and anti-disassembly) is [39]. In this work, entropy analysis and instructions-based detection are put together in order to intercept any type of obfuscation. Instruction-based detection consists of some consideration about the control-flow graph: this means that detection is performed when anomalies in the control-flow graph are detected. Entropy analysis is handled so that only entropy of referenced (i.e. actually executed) instructions is computed. However, this idea could work only if the packed/encrypted code is stored in the same area of the entry point, but this is not always true for LEPM, so [39] does not provide a solution for our problem.

Differently, the authors in [36] base their methodology on entropy computation, but in a different way. They extract the first three fragments of fixed size (1024 byte) from each binary in the dataset and they compute the entropy of each fragment. Further computations based on the three fragments are executed in order to obtain a Similarity Distance Matrix (based on the entropy of the initial fragments and their union). By using the distances in the matrix it is possible to separate packed and not packed malware with an higher granularity level with regard to common entropy-based approaches. The same work also considers the possibility to use XOR encryption (or XOR encoding, as they define this transformation) to preserve the randomness of a file. However they do not provide any evidence of real-world samples using such transformation.

An alternative proposal [16] tries to solve the problem of the classification of packed executables with a methodology inherited from the signal processing world. They offer a visual representation of an executable (i.e., bytes become pixels) and they apply signal processing algorithms to make a classification of the obtained images. The weakness of this work is that it mainly does not take into consideration LEPM, but only off-the-shelf packers.

Differently from all the previous researches, [17] builds a classification system for packed/polymorphic malware taking advantage of both flow graph and dynamic analysis. This method works well with off-the-shelf packer even if for large dataset to analyze, dynamic analysis introduces time constraints.

An interesting proposal deepened in [25] tries to build a packer-agnostic filter aimed to find similarities between packers leaning on the effects of the cryptographic transformations over the byte distribution. Even if byte distribution is directly related to entropy variations, authors of [25] do not consider explicitly the cases of LEPM, thereby keeping the problem still open.

Table 5.1 summarizes all the presented static analysis approaches and, for machine learning approaches, it lists the selected set of features. We also report the way that the different authors used to construct the dataset used in experiments to prove the proposed theory. In the next section we consider the features listed in the table to point out how the existing approaches work with the LEPM. Note that we decided to report in Table 5.1 only the methodologies which are related to the retrieval of static features. Strategies that employ dynamic analysis are not interesting from our point of view.

Table 5.1: a schematic view of the existent approaches

Paper	Proposed approach	Features (only for ML approaches)	Dataset construction
Lyda et al.	Entropy analysis		Not Packed: benign executables Packed: Application of a set of packers to the benign executables
Choi et al.	Machine learning	executable and writable sections, number of the sections which is executable but not a code or which is not-executable but a code, section with no printable name, doExecutableSectionExist, sum of every section size, position of PE signature, position of entry point, protection of entry point section	Not Packed: benign executables and PE files from AV manually analysed Packed: benign executables and PE files from AV manually analysed
Perdisci et al. (1)	Machine learning	standard sections, non standard sections, executable sections,RWX sections, IAT entries, PE header, code, data and file entropy	Not Packed: benign executables Packed: malware from MALFEASE project and application of a set of packers to the benign executables
Perdisci et al. (2)	Machine learning	Feature taken by Perdisci et al. (1) with addition of n-gram analysis of code section and entire PE file	Not Packed: malware from MALFEASE project filtered with state-of-the-art unpackers (Polyunpack [38], Renovo [27]) Packed: malware from MALFEASE project and application of a set of packers to the benign executables
Santos et al.	Machine learning	sections with entropy in range [7.5;8.0], sections with entropy in range [7.0;7.5], mean data section entropy, global file entropy, maximum entropy, mean code section entropy, section of entry point entropy, pointer to raw data, sections greater than raw data, non standard sections, header characteristics field, minor Linker Version, virtual address, standard sections, size of heap commit, sections RWX, major linker version, timeStamp, sections RW, sections RX, address of entry point, size initialized data, IAT size, size of raw data, sections R, sections W, sections X, size of first section raw data, size of heap reverse, header characteristics, standard sections, size code, number of sections, virtual size, size rsrc table, base of data, major OS version	Not packed: benign executables and malware from VxHeavens Packed: Variants of the 'Zeus' family and application of a set of packers to the benign executables
Ugarte-Pedrero et al.	Machine learning and n-grams	Features taken by Santos et al. with addition of header fields (∼200 features). Size of n-grams with n in range [1; 5]	Not packed: benign and malicious executables filtered by PEid, entropy analysis, IAT entries, imported dlls and ratio of standard sections Packed: Application of a set of packers to the benign executables, malware reported by PEid as packed, malware from 'Zeus' family
Nandi et al.	Machine learning	Entropy, size of uninitialized data, size of headers, size of raw data	Not packed: benign executables Packed: Application of UPX packer to the benign executables
Ugarte-Pedrero et al.	Entropy analysis		Not packed: benign executables and malicious samples taken from VxHeavens and checked with PEid Packed: application of a set of packers to the benign executables and malware from 'Zeus' family
Saleh et al.	Control-flow graph		Not packed: benign executables Packed: malicious samples packed with both custom and off-the-shelf packers
Raphel et al.	Entropy analysis		Not packed: benign executables Packed: Application of a set of packers/encoders to the benign executables
Burgess et al.	Steganalysis		Not packed: benign executables Packed: Application of a set of packers to the benign executables
Han et al.	Entropy analysis		Not packed: benign executables Packed: malicious samples from honeypot

Chapter 6

Future work and conclusions

6.1 Conclusions and remarks

We have demonstrated that actual approaches do not take into account threats represented by LEPM. If some years ago LEPM was only an academic theory, nowadays both malware researchers and AV vendors are obliged to consider this problem and to face it in order to improve their detection methods. No existing static analysis technique is able to reliable detect LEPM, but this does not mean that such approach must be abandoned. However, in future works about this topic, a good starting point could be to insist on machine learning techniques to find a set of features which separate the classes of *packed* and *not packed* malware even if LEPM are present in the dataset.

6.2 Future work

In this thesis we tried to explain the danger represented by LEPM through the following steps:

- Theoretical study about entropy and its dependencies concerning cryptographic transformations
- Development of a complex tool able to dynamically analyze samples to decide if they are real LEPM
- Building up of a dataset of malware observed in wild
- Analysis of the schemes adopted by LEPM and proposal of a taxonomy of such schemes
- Demonstration of how much the problem is spread (i.e. demonstration of the LEPM spread)

Existent trends developed just to detect malware with accuracy, precision and high-performance rely on statically computed metrics which can be easily bypassed by LEPM. In this context, countermeasures exhibited by LEPM authors are:

- For each section entropy levels are low
- PE headers of LEPM often appear like a goodware PE (sections named with standard names, entry point correctly located inside .text, etc.)

Our intent for future developments of this work is to demonstrate that state of the art is wrong with a not negligible error. For this purpose, we will follow a series of well-defined steps:

1. Building of a proper dataset without any pollution problem, made of packed samples (both goodware and malware), not packed samples (both goodware and malware) and LEPM. With the expression *pollution* we mean the problem which affects the dataset when it contains samples which are used for training the classifier, but which are not correctly classified. In our case, datasets used in previous work, could include packed samples labeled as not packed and used for the training because maybe they are LEPM or evade the virtual environment.
2. To run different classifiers, using as set of features all the ones listed in table 5.1
3. Elaboration of results and plots

After leading some preliminary tests, we are confident to find that LEPM statical metrics tend to be closer to the ones of not packed files rather than the packed binaries ones.

Unfortunately, building a dataset of proper size (about 100K files), without samples performing virtual environment detection and without LEPM wrongly classified, can be done only by a very time-consuming dynamic analysis experiment and this is the reason for which final results are not present in this thesis. The last contribution of this work will be to release the dataset so that researchers interested in this topic could lead their experiments using a trusted dataset of appropriate size.

Going on with this work is not only an interest for the community, but it will represent also the beginning of my phd research activity at Eurecom, so in next months it will be ended.

Bibliography

[1] https://rada.re/r/.

[2] https://upx.github.io/.

[3] http://www.siliconrealms.com/armadillo.htm.

[4] http://www.aspack.com/aspack.html.

[5] https://www.aldeid.com/wiki/PEiD.

[6] Ida pro. https://www.hex-rays.com/.

[7] Peering inside the pe: A tour of the win32 portable executable file format. https://msdn.microsoft.com/en-us/library/ms809762.aspx.

[8] Virus total. https://www.virustotal.com/. Accessed October 2018.

[9] volatility: a memory analysis framework. https://www.volatilityfoundation.org/.

[10] M. Baig, P. Zavarsky, R. Ruhl, and D. Lindskog. The study of evasion of packed pe from static detection. In *World Congress on Internet Security (WorldCIS-2012)*, pages 99–104, June 2012.

[11] Munkhbayar Bat-Erdene, Taebeom Kim, Hyundo Park, and Heejo Lee. Packer detection for multi-layer executables using entropy analysis. In *Entropy*, volume 19, page 125, 03 2017.

[12] Munkhbayar Bat-Erdene, Hyundo Park, Hongzhe Li, Heejo Lee, and Mahn-Soo Choi. Entropy analysis to classify unknown packing algorithms for malware detection. *Int. J. Inf. Secur.*, 16(3):227–248, June 2017.

[13] Ulrich Bayer, Andreas Moser, Christopher Kruegel, and Engin Kirda. Dynamic analysis of malicious code. *Journal in Computer Virology*, 2(1):67–77, Aug 2006.

[14] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[15] T. Brosch and M. Morgenstern. Runtime packers: The hidden problem? *Black Hat*, 2006.

[16] Colin Burgess, Fatih Kurugollu, Sakir Sezer, and Keiran McLaughlin. Detecting packed executables using steganalysis. In *Visual Information Processing (EUVIP), 2014 5th European Workshop on*, pages 1–5. IEEE, 2014.

[17] Silvio Cesare, Yang Xiang, and Wanlei Zhou. Malwise—an effective and efficient classification system for packed and polymorphic malware. *IEEE Transactions on Computers*, 62(6):1193–1206, 2013.

[18] Yang-Seo Choi, Ik kyun Kim, Jin-Tae Oh, and Jae cheol Ryou. Pe file header analysis-based packed pe file detection technique (phad). *International Symposium on Computer Science and its Applications*, pages 28–31, 2008.

[19] Dhruwajita Devi and Sukumar Nandi. Pe file features in detection of packed executables. *International Journal of Computer Theory and Engineering*, 4(3):476, 2012.

[20] Brendan Dolan-Gavitt. The vad tree: A process-eye view of physical memory. *Digit. Investig.*, 4:62–64, September 2007.

[21] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with panda. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, page 4. ACM, 2015.

[22] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.*, 44(2):6:1–6:42, March 2008.

[23] Robert P Goldberg. Survey of virtual machine research. *Computer*, (6):34–45, 1974.

[24] Seung-Won Han and Sang-Jin Lee. Packed pe file detection for malware forensics. *The KIPS Transactions: PartC*, 16(5):555–562, 2009.

[25] Grégoire Jacob, Paolo Milani Comparetti, Matthias Neugschwandtner, Christopher Kruegel, and Giovanni Vigna. A static, packer-agnostic filter to detect similar malware samples. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 102–122. Springer, 2012.

[26] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-llvm–software protection for the masses. In *Software Protection (SPRO), 2015 IEEE/ACM 1st International Workshop on*, pages 3–9. IEEE, 2015.

[27] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: a hidden code extractor for packed executables. In *In Proc. ACM Workshop Recurring Malcode (WORM*, pages 46–53. ACM, 2007.

[28] Eric Lafortune. Proguard. https://sourceforge.net/projects/proguard/, 2004. Accessed October 2018.

[29] Robert Lyda and James Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE Security & Privacy*, 5(2), 2007.

[30] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 421–430, Dec 2007.

[31] Hausi A Müller, Jens H Jahnke, Dennis B Smith, Margaret-Anne Storey, Scott R Tilley, and Kenny Wong. Reverse engineering: A roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 47–60. ACM, 2000.

[32] P. OKane, S. Sezer, and K. McLaughlin. Obfuscation: The hidden malware. *IEEE Security Privacy*, 9(5):41–47, Sept 2011.

[33] R. Perdisci, A. Lanzi, and W. Lee. Mcboost: Boosting scalability in malware collection and analysis using statistical classification of executables. In *2008 Annual Computer Security Applications Conference (ACSAC)*, pages 301–310, Dec 2008.

[34] Roberto Perdisci, Andrea Lanzi, and Wenke Lee. Classification of packed executables for accurate computer virus detection. *Pattern recognition letters*, 29(14):1941–1946, 2008.

[35] Yudi Prayudi, Imam Riadi, et al. Implementation of malware analysis using static and dynamic analysis method. *International Journal of Computer Applications*, 117(6), 2015.

[36] Jithu Raphel and P. Vinod. Information theoretic method for classification of packed and encoded files. In *Proceedings of the 8th International Conference on Security of Information and Networks*, SIN '15, pages 296–303, New York, NY, USA, 2015. ACM.

[37] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 289–300, Dec 2006.

[38] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 289–300, Dec 2006.

[39] M. Saleh, E. P. Ratazzi, and S. Xu. Instructions-based detection of sophisticated obfuscation and packing. In *2014 IEEE Military Communications Conference*, pages 1–6, Oct 2014.

[40] Igor Santos, Xabier Ugarte-Pedrero, Borja Sanz, Carlos Laorden, and Pablo G Bringas. Collective classification for packed executable identification. In *Proceedings of the 8th Annual Collaboration, Electronic messaging, Anti-Abuse and Spam Conference*, pages 23–30. ACM, 2011.

[41] Muhammad Zubair Shafiq, S. Momina Tabish, and Muddassar Farooq. Pe-probe : Leveraging packer detection and structural information to detect malicious portable executables. 2009.

[42] Li Sun, Steven Versteeg, Serdar Boztaş, and Trevor Yann. Pattern recognition techniques for the classification of malware packers. In *Australasian Conference on Information Security and Privacy*, pages 370–390. Springer, 2010.

[43] Symantec. Trojan.zbot. https://www.symantec.com/security-center/writeup/2010-011016-3514-99, 2010. Accessed October 2018.

[44] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *2015 IEEE Symposium on Security and Privacy (SP)*, pages 659–673. IEEE, 2015.

[45] Xabier Ugarte-Pedrero, Igor Santos, Iván García-Ferreira, Sergio Huerta, Borja Sanz, and Pablo G Bringas. On the adoption of anomaly detection for packed executable filtering. *Computers & Security*, 43:126–144, 2014.

[46] Xabier Ugarte-Pedrero, Igor Santos, Borja Sanz, Carlos Laorden, and Pablo Garcia Bringas. Countering entropy measure attacks on packed software detection. In *Consumer Communications and Networking Conference (CCNC), 2012 IEEE*, pages 164–168. IEEE, 2012.