To someone...

List of Acronyms

IAT Import Address Table

OEP Original Entry Point

API Application Programming Interface

DBI Dynamic Binary Instrumentation

EIP Extended Instruction Pointer

PE Portable Executable

WxorX Write xor eXecution

TEB Thread Environment Block

Summary

Thanks

Chapter 1

Introduction

Chapter 2

Motivation

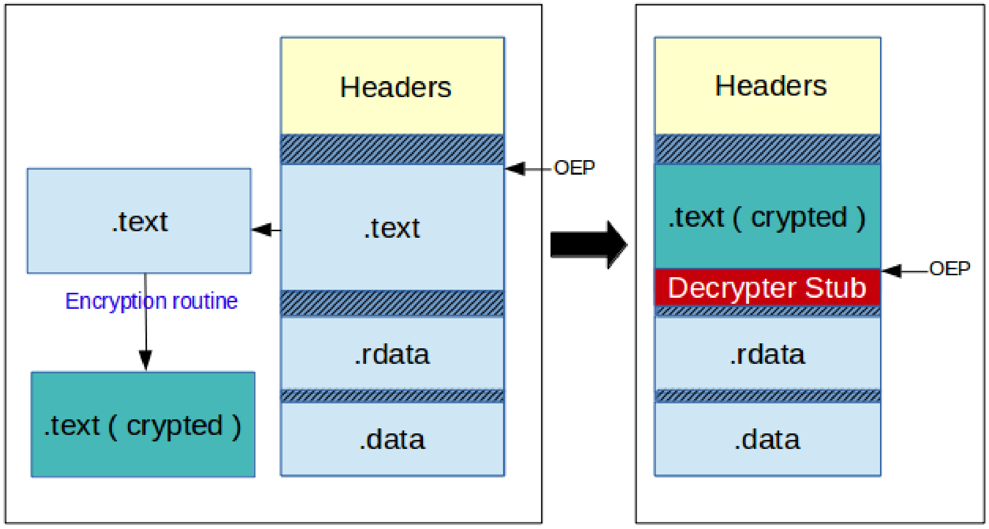
2.1 Problem statement

In order to avoid Anti-Virus (AV) detection and harden the process of reverse engineering usually malware hide and protect their code employing different techniques and tools. This process, generally called *packing*, makes the static analysis of a binary almost completely useless.
There are lots of different run-time tools available in order to compress/encrypt/protect both good and malicious software, the main categories in which this tools fall are:

- *Packer*: a packer usually tries to reduce the size of a binary by using different compression algorithms.
- *Crypter*: the goal of a crypter is to encrypt the executable and hindering the disassembly.
- *Protector*: a protector implements different anti-debugging and generally anti-reversing techniques in order to protect the binary copyright in the case of a good program, or hide the malicious code in case of a malware.
- *Virtualizer*: a virtualizer translates the code of the original program into a new instruction set and interprets it at runtime.

All these tools change deeply the original PE file by modifing the OEP, the Import directory table and many other resources and fields. Usually they are mixed together in order to obtain a more advanced protection tool that in this work we will refer generally as *packer*.

A *packer* is the tool that implements the previously described functions: it receives in input a PE file that represent a Windows program, transforms and obfuscates its code/resources and then appends new codes, called *stub*, that will *unpack* the original one runtime once the program is executed.



Malware writers can decide to use an *off-the-shelf* packer (commercial or free) or to implements their own solution with a *custom packer*.

The complexity of *packers* can be very different: from those which writes and executes directly the original code, to others that employs multiple unpacking routines and obfuscation techniques such as runtime repacking of previously unpacked code.

This process has different consequences both in the AV detection and manual analysis of malicious binary:

- Usually AV employs different static analysis techniques in order to understand if a file is malicious or not, but packing a binary destroys any possibility to understand what the program will do on the system without executing it, and so void any effort from the point of view of a static analysis.

- The process of reverse engineering a packed malware can be very time consuming and a lot of time is wasted trying to extract the original payload instead of focusing on its behaviour .

These problems inspired different works in building an automatic generic unpacker aimed to extract the original code from the packed one. Some of them are more oriented in detection of malicious packed program helping an AV software on end users PCs, others are instead proposed as tools for speed up the work of professional malware analysts.

A comprehensive study and a taxonomy of the levels of complexity of nowadays packers have been presented by Ugarte-Pedrero et al[**?**]. In order to clarify better what a packer is and how it operates we will discuss in the next chapter the main points of the previously mentioned research.

2.1.1 Packer taxonomy

As previously explained a packer can implements different obfuscation techniques in order to harden the analysis of a program. In this capther we are going to analyze the proposed taxonomy in order to better understand the goals of our work and its limitation against the current packing methods.

Before starting to present the techniques we need to define some terms that are essential in order to understand how a packer works:

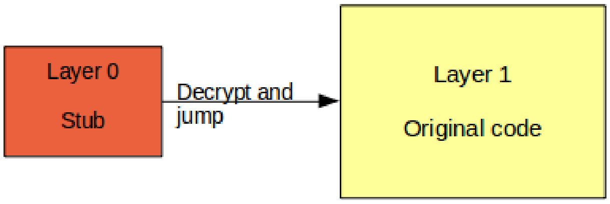
- *Layer*: a layer is a set of contiguous memory addresses that are executed after being written.A layer can unpack another unpacking-layer or layer that contains the original program code.
- *Transition*: a transition is a control transfer from a layer to another layer, it can be a *forward transition* if the execution is going from a previously unpacked layer to a newer layer, or a *backward transition* if the execution is going from a newer layer to an older one. From this definition we can derive the concept of *linear unpacking* in which all the transition are *forward transition*, and cyclic unpacking in which there are some *forward transition* and some *backward transition.*
- *Tailed/Interleaved*: We say that a packer is tailed if there is soon or later a transition from the unpacking layers to the original program code and then the execution never returns to the unpacking layers, contrary we say that the unpacking is *interleaved* if the execution bounce between the unpacking layers and the original program code.
- *Frame*: a frame is a portion of original program code unpacked and executed. If a packer is *mono frame* that means that the unpacking routine will reveal all the original code before jumping and executing it; contrary a *multi frame* packer unpacks only a slice of the original program code, execute it and then unpack another slice of original program code and so on. A *multi frame* behavior can be *incremental* if the slices of original program are revelead in order following the original program execution

and never re-encrypted again, contrary a *shifted frames* policy reveal only one slices at times and re-encrypt the slice once executed.

As proposed by Ugarte-Pedrero et al[?] we are going to identify the complexity of a packer by using a scale from 1 to 6.

Type 1 packer

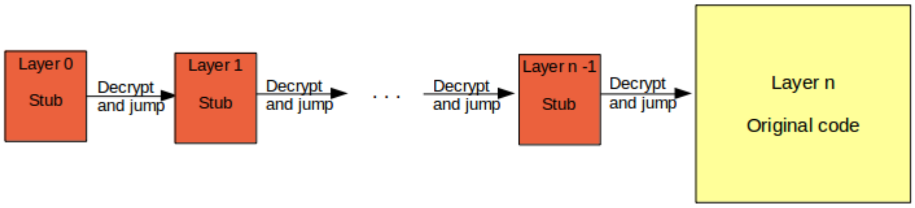
The simplest form of packers are *single layer*, they runtime unpack the original binary using a single stub and then jumps to it with a tail jump.


Layer 0
Stub
Decrypt and jump
Layer 1
Original code

This scheme is typical of packers like UPX,EXEpacker.

Type 2 packer

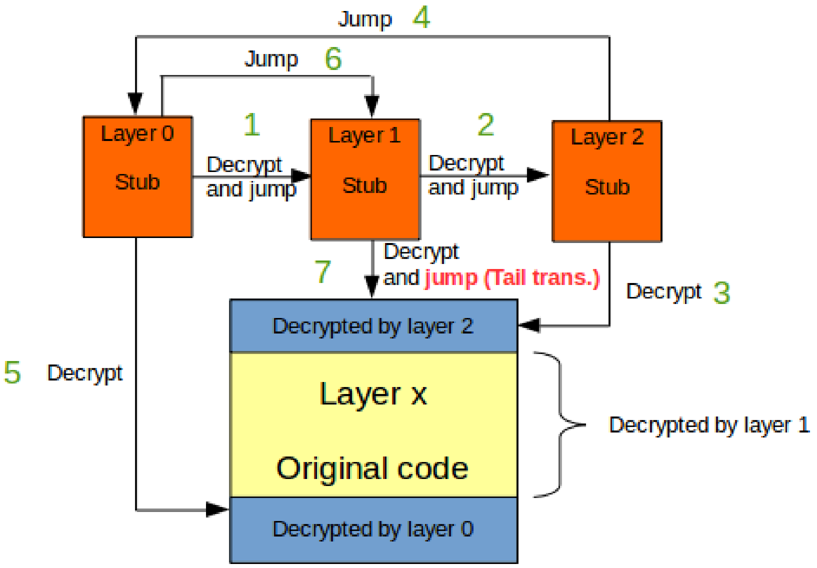
The packer in this category are defined as *multi-layer* and *linear*. This means that they employ different layers and the transitions between them are always from the older to the newer. Also this kind of packers are defined as *tailed* since the last jump redirect the execution at the original program code.


Layer 0
Stub
Decrypt and jump
Layer 1
Stub
Decrypt and jump
. . .
Decrypt and jump
Layer n -1
Stub
Decrypt and jump
Layer n
Original code

We can see from the picture that the layer 0 unpack and jumps to the layer1, then the scheme is repeated untill the layer(n-1) that decrypts and finally jumps to the original code. All the layer from 0 to n-1 are part of the packer stub.

Type 3 packer

These packers are defined as *multi-layer*, *cyclic* and *tailed*. This means that they employ different layers and the transitions are both *forward transition* and *backward transition*.


Jump 4
Jump 6
Layer 0
Stub
1
Decrypt and jump
Layer 1
Stub
2
Decrypt and jump
Layer 2
Stub
Decrypt and jump (Tail trans.)
7
Decrypt 3
Decrypted by layer 2
5 Decrypt
Layer x
Original code
Decrypted by layer 1
Decrypted by layer 0

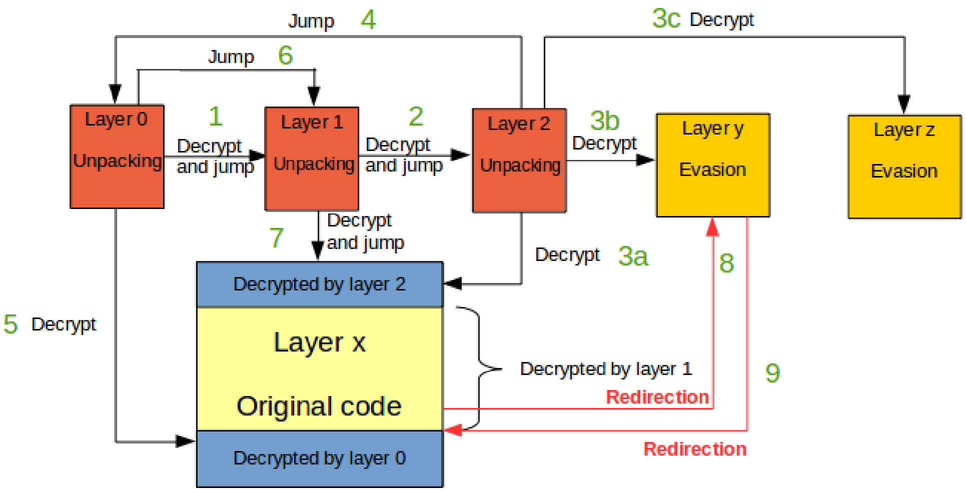
Following the number reported on the picture let's explain briefly how the process of the packer's stub works in this example of packer type 3, keep in mind that the scheme can change but the properties for this kind of packers remains the one described previously.

- (1) The first layer decrypt and finally jump to the layer 1.
- (2) The layer 1 do the same thing, unpacking and jumping to the layer 2.
- (3) Then the layer 2 decrypt a piece of original program code, and then (4)re-jump with a backward transition to the layer 0.
- (5) Now the layer 0 decrypt another piece of the original program code and then (6)re-jump to layer 1.
- (7)Finally layer 1 decrypt the last piece (Layer X) of original code and jump to the OEP of the program.

A scheme of this type is implemented by PE-Compact, Aspack, FSG, AS-protect,NSPack,WinUpack,UPolyX and Obsidium.

Type 4 packer

These packers are *multi-layer*, *cyclic* and *interleaved*. This means that they use different layers with both *forward and backward transitions*, and during the execution of the original program the control is redirected in some way to the packer code.

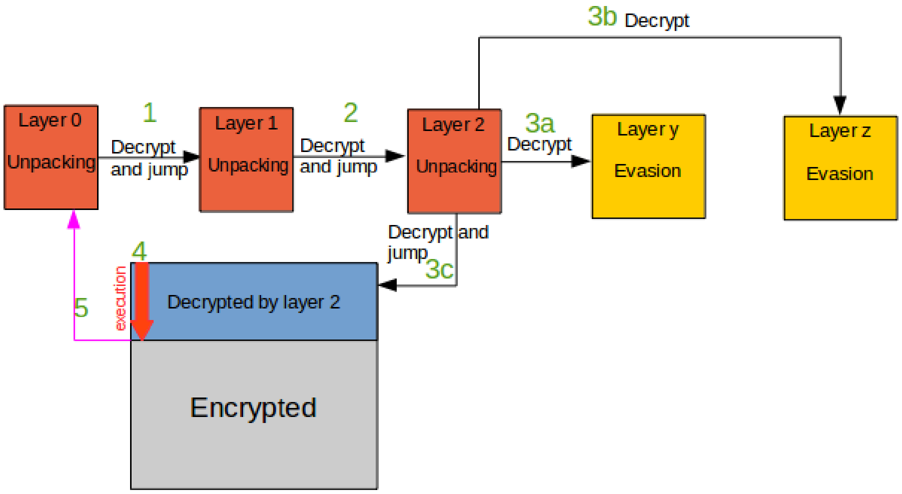

Jump 4
3c Decrypt
Jump 6
Layer 0
Unpacking
1
Decrypt
and jump
Layer 1
Unpacking
2
Decrypt
and jump
Layer 2
Unpacking
3b
Decrypt
Layer y
Evasion
Layer z
Evasion
7
Decrypt
and jump
Decrypt 3a
8
Decrypted by layer 2
5 Decrypt
Layer x
Original code
Decrypted by layer 1
Redirection
9
Decrypted by layer 0
Redirection

- (1) The first layer decrypts and finally jump to the layer 1.
- (2) The layer 1 do the same thing, unpacking and jumping to the layer 2.
- (3a) The layer 2 decrypts a piece of original code and (3b+3c)two additional layers that will implement, for example, some evasion techniques like debugger detection. After this (4) it jumps with a backward transition to layer 0.
- (5) Now layer 0 decrypts another piece of original program and then (6) jumps to layer 1 again.
- (7) Layer 1 finally decrypts and jump to the original program OEP.
- (8) During program execution the packer re-gain control executing the evasion routines; this is usually implements by hijacking the API called by original program.
- (9) After the evasion code has been executed the execution can resume to the original program code or into other location based on the results of the evasion checks implemented.

A scheme of this type is implemented by ACProtect.

Type 5 packer

Packers of this type are *multi-layer*, *cyclic*,*interleaved* and *multi frame* with an *incremental frames* behavior. This means that they use different layers with both *forward and backward transitions*, during the execution of the original program the control is redirected in some way to the packer code and the original program code is revelead slice by slice when need to execute; Like the previous packer's type also here exists a moment in which all the original program code is unpacked in memory, but differently in this case dumping at the OEP doesn't permit to dump all the original program code since it is still encrypted in memory; nonetheless a theoretical dump nearly the end of execution can dump the entire program's code.

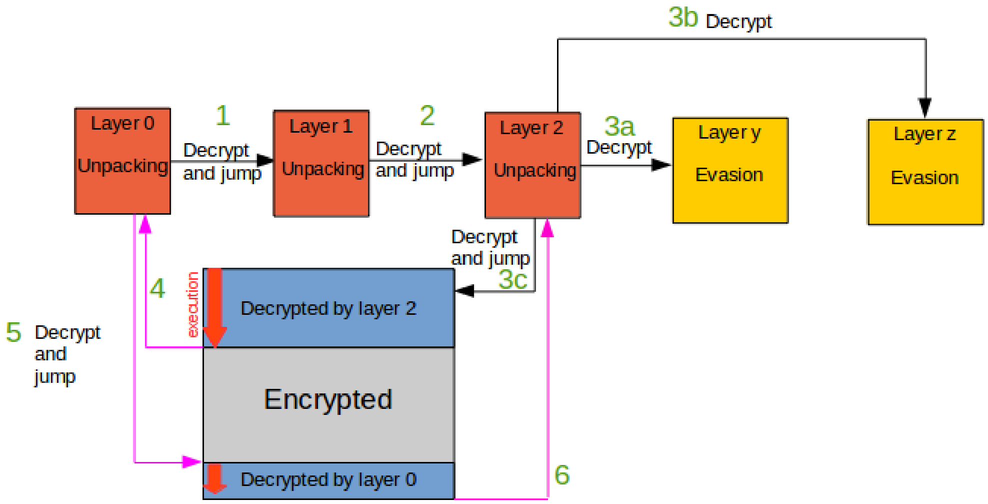

3b Decrypt
Layer 0
Unpacking
1
Decrypt and jump
Layer 1
Unpacking
2
Decrypt and jump
Layer 2
Unpacking
3a
Decrypt
Layer y
Evasion
Layer z
Evasion
Decrypt and jump
3c
4
execution
5
Decrypted by layer 2
Encrypted

- (1) The layer 0 unpacks and jumps to layer 1.

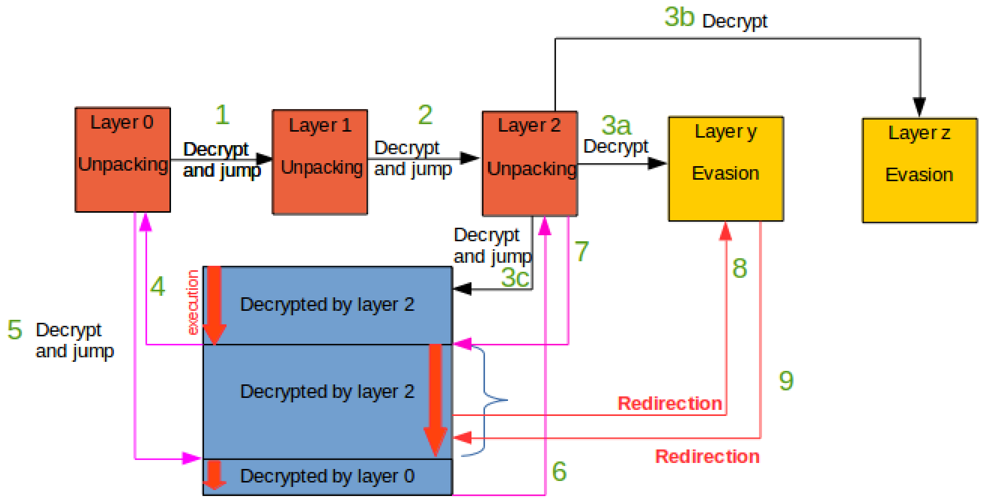
- (2) The layer 1 unpacks and jumps to layer 2.

- (3a+3b) Layer 2 unpacks two additional layers that will implements some kind of evasion techniques. Finally (3c) it unpacks a slice of original program's code at the OEP and redirect the execution there (4)starting to execute the original program.

- (5) The execution reach the end of the unpacked original program's code and supposedly an exception will be launched. The packer catches this exception and redirect the execution to layer 0.


3b Decrypt
Layer 0
Unpacking
1
Decrypt
and jump
Layer 1
Unpacking
2
Decrypt
and jump
Layer 2
Unpacking
3a
Decrypt
Layer y
Evasion
Layer z
Evasion
Decrypt
and jump
3c
4
execution
Decrypted by layer 2
5 Decrypt
and
jump
Encrypted
Decrypted by layer 0
6

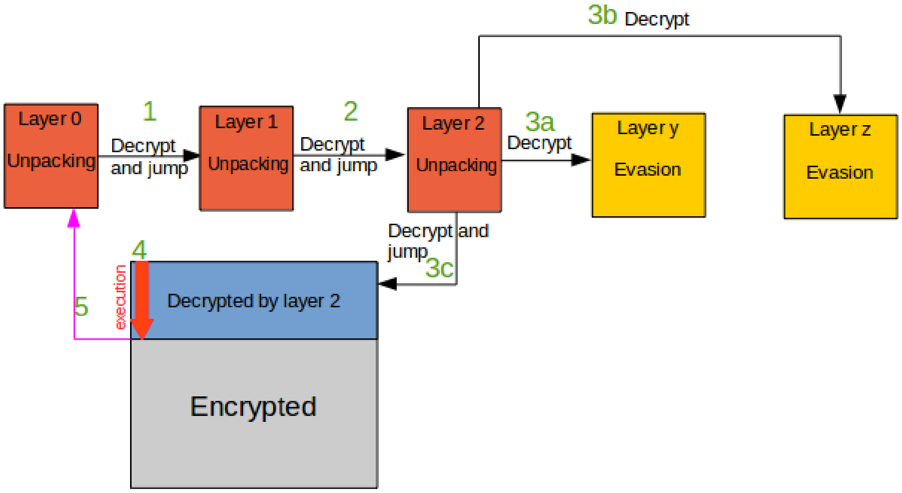
- (5) Layer 0 decrypts on demand a new slice of original program's code and then jumps to it.
- (6) The same things as before happen when execution reaches again the end of the previously unpacked code, and now the layer 2 take control.


3b Decrypt
Layer 0
Unpacking
1
Decrypt
and jump
Layer 1
Unpacking
2
Decrypt
and jump
Layer 2
Unpacking
3a
Decrypt
Layer y
Evasion
Layer z
Evasion
Decrypt
and jump
3c
7
8
4
execution
Decrypted by layer 2
5 Decrypt
and jump
Decrypted by layer 2
Redirection
9
Redirection
Decrypted by layer 0
6

- (7) Layer 2 decrypts the last slice of original program's code and finally jumps to it. As before in (8) there is a redirection to an evasion layer that can decide to (9) resume the execution to the original program's code, or to abort the execution depending on its checks.

Type 6 packer

Packers of this type are *multi-layer*, *cyclic*, *interleaved* and *multi frame* with a *shifted frames* behavior. This means that they use different layers with both *forward and backward transitions*, during the execution of the original program the control is redirected in some way to the packer code and the original program code is revelead slice by slice and re-encrypted once executed; packers with such complexity never let all the program code to live in memory, so a dump that encompass the entire original program's code doesn't exists.

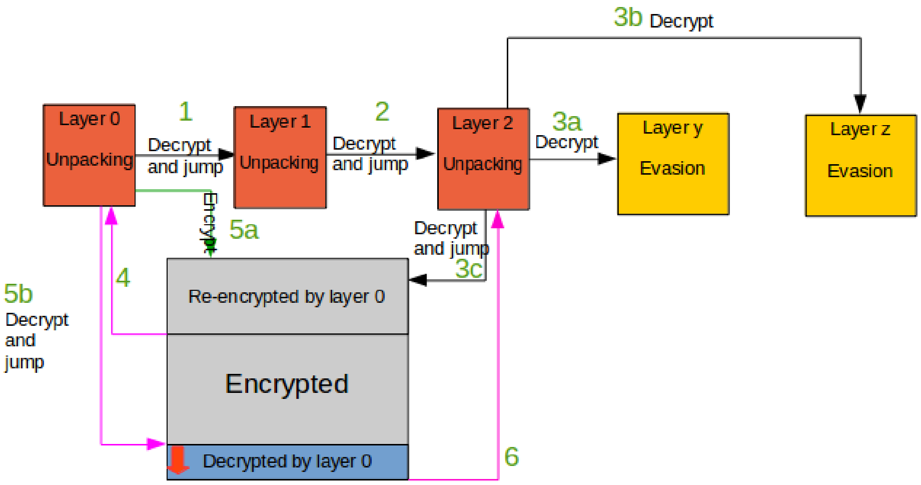

3b Decrypt
Layer 0
Unpacking
1
Decrypt and jump
Layer 1
Unpacking
2
Decrypt and jump
Layer 2
Unpacking
3a
Decrypt
Layer y
Evasion
Layer z
Evasion
Decrypt and jump
3c
4
5
execution
Decrypted by layer 2
Encrypted

- (1) The layer 0 unpacks and jumps to layer 1.

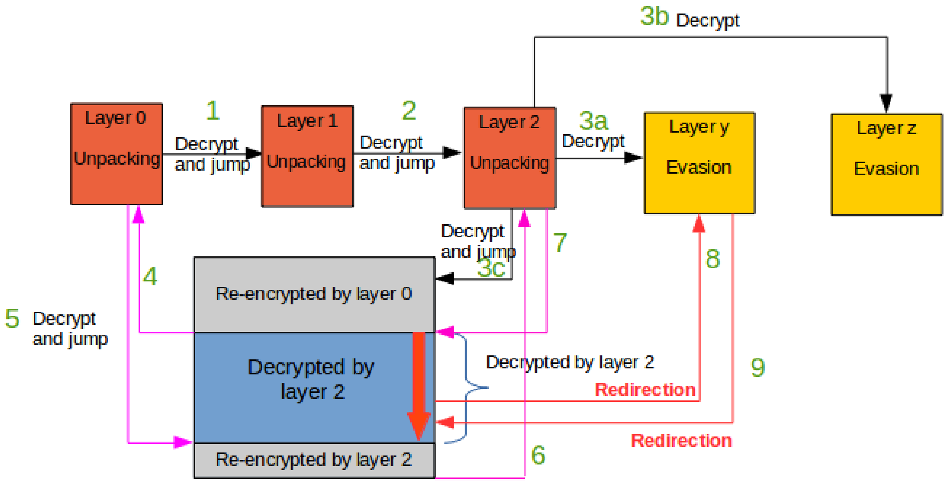
- (2) The layer 1 unpacks and jumps to layer 2.

- (3a+3b) Layer 2 unpacks two additional layers that will implements some kind of evasion techniques. Finally (3c) it unpacks a slice of original program's code at the OEP and redirect the execution there (4)starting to execute the original program.

- (5) The execution reach the end of the unpacked original program's code and supposedly an exception will be launched. The packer catches this exception and redirect the execution to layer 0.


3b Decrypt
Layer 0
Unpacking
1
Decrypt
and jump
Layer 1
Unpacking
2
Decrypt
and jump
Layer 2
Unpacking
3a
Decrypt
Layer y
Evasion
Layer z
Evasion
Encrypt
5a
Decrypt
and jump
3c
4
5b
Decrypt
and
jump
Re-encrypted by layer 0
Encrypted
Decrypted by layer 0
6

- (5a) Layer 0 re-encrypts the previous slice of original program's code, and (5b) decrypts on demand a new slice of original program's code; finally it jumps to the new unpacked slice.
- (6) The same things as before happen when execution reachs again the end of the previously unpacked code, and now the layer 2 take control.


3b Decrypt
Layer 0
Unpacking
1
Decrypt
and jump
Layer 1
Unpacking
2
Decrypt
and jump
Layer 2
Unpacking
3a
Decrypt
Layer y
Evasion
Layer z
Evasion
Decrypt
and jump
3c
7
8
4
5 Decrypt
and jump
Re-encrypted by layer 0
Decrypted by
layer 2
Decrypted by layer 2
Redirection
9
Redirection
Re-encrypted by layer 2
6

- (7) Layer 2 re-encrypts the previous slice of original program's code and decrypts the last slice, jumping to it at the end. As before in (8) there is a redirection to an evasion layer that can decide to (9) resume the execution to the original program's code, or to abort the execution depending on its checks.

A scheme of this type is implemented by Armadillo.

Packer complexity distribution

According to Ugarte-Pedrero et al[**?**], the distribution of packer complexity resulted from the analysis of 6088 packed malware binaries is the following:

Type	Off-the-shelf	Custom packers
Type 1	173 (25.3%)	443 (7.3%)
Type 2	56 (8.2%)	752 (12.4%)
Type 3	352 (51.4%)	3993 (65.6%)
Type 4	86 (12.6%)	843 (13.8%)
Type 5	6 (0.9%)	46 (0.8%)
Type 6	12 (1.8%)	11 (0.2%)

Interpreting the data we can derive that:

- Malware writers seems to predilige using a custom packer rather than rely on an existing one.
- more than 90% of the binary packed with both an off-the-shelf/custom packer use a complexity that range from 1 to a maximum of 4. This suggest that a generic unpacker that can handle such complexities can cover a good number of cases.
- In both cases of an off-the-shelf/custom packer we can see that the preferred complexity seems to be a Type 3 packer. This could be an indication that a complexity of that level is enough to defeat the nowadays AV solutions.
- The use of a complexity greater than IV is rarely used.

As presented, the complexity of a packer can escalate really quickly and building a generic unpacker that can cover all these schemes is a big challenge.

2.2 State of the art

Lots of different tools using different approaches and techniques have been proposed. The approaches for automatic unpacking can be very different:

- *Static unpacking*: this can sound counterintuitive, but some works proposed to identify unpacking routines inside the binary and reconstruct an ad hoc unpacker for the binary starting from these routines. This approach has got numerous limitation but can lead to interesting result in some case as demonstrated by Coogan et al.[**?**]
- *Hybrid unpacking*: this mixes some static heuristics with dynamic analysis.
- *Dynamic unpacking*: these techniques follow the idea to let the unpacker do its work and then try to extract runtime the unpacked code.

Depending on the purpose of the tool there are different requirements that a generic unpacker must respect. If the aim is to help the AV on end users' PCs:

- Safety: try to recognize the malicious behaviour as fast as possible and block the execution.
- Performance: it should not slow down too much the execution of AV scans.

Note that in these cases, the scope is not to reconstruct a binary from a packed one, but rather to stop malicious behaviours when they manifest.
In this area have been done works such as OmniUnpack and JustIn.

On the other side if the aim is to help the analysis in a laboratory:

- Fidelity: the unpacked binary extracted by the tool should be equal to the one that would be unpacked normally.
- Generality: the unpacker can not be focused only against one packer but should unpack different of them with one generic algorithm.

In this case we do not care so much about safety because usually analysis is performed inside a controlled environment and the analysts want to observe the complete execution of malware. Also the performances are not a critical feature here because we are not constrained by user experience needs.
In this category have been developed tools like PolyUnpack, Ether, Eureka, Renovo, Lynx. These tools merely collect dumps of the binary while unpacking and they don't reconstruct a fully runnable binary given a packed one.

2.3 Goals and challenges

Since our work is a component of a bigger malware analysis platform[**?**], our tool is oriented to help malware analyst during the reversing process of a packed binary. Our approach aims not only to unpack the malware, but also to reconstruct a fully working unpacked binary. To do so, we not only have to identify

the original entry point (OEP) and dump the code at that moment, but we have to find the IAT inside the process and reconstruct a correct import directory in the final PE file.
According to the discussion about packer's complexity our aim is to unpack untill Type 4 since, as presented by Ugarte-Pedrero et al[**?**] in their packer's complexity distribution, more than 90% of the binary packed with both an off-the-shelf/custom packer use a complexity that range from 1 to a maximum of 4.

The first thing we have to deal with are the unpacking routines of the packers: every time the execution of the malware comes from a previously written memory area, then it could be a sign that the unpacking stage has finished or that a new unpacking layer has started.
We have also to deal with techniques of IAT obfuscation: some malwares can do this in order to make difficult to statically analyse them to understand what they are doing.

Chapter 3

Approach

3.1 Dynamic Binary Instrumentation

Dynamic Binary Instrumentation is a technique for analysing the behaviour of a binary application through the injection of instrumentation code. The instrumentation code can be developed in an high level programming language and is executed in the context of the analysed binary with a granularity up to the single assembly instruction. The injection of instrumentation code is achieved by implementing a set of callbacks provided by the DBI[1] framework. The most common and useful callbacks are:

- Instruction callback: invoked for each instruction
- Image load callback: invoked each time an image (dll or Main image) is loaded into memory
- Thread start callback: invoked each time a thread is started

Besides the callbacks the DBI framework allows to intercept and modify operative system APIs[2] and system calls and this is very useful to track some behaviours of the binary, like the allocation of dynamic memory areas.

3.2 Approach overview

Our tool exploits the functionalities provided by the Intel PIN DBI framework to track the memory addresses which are written and then executed with an instruction level granularity. More in details for each instruction the following steps are performed:

---

[1] Dynamic Binary Instrumentation

[2] Application Programming Interfaces

1. Instruction Filtering: ignore the effects of a particular set of instructions for performance reasons
2. Written addresses tracking: keep track of each memory address which is written in order to create a list of memory ranges of contiguous writes defined *Write Intervals*
3. *WxorX*[3] instructions tracking: check if the currently executed instruction belongs to a *Write Interval*. This is a typical behaviour in a packer that is executing the unpacked layer and for this reason we trigger a detailed analysis which consists of:
    (a) Dumping the main image of the PE[4] and a memory range on the heap depending on the address of the current instruction
    (b) Reconstructing the IAT[5] and generating the correct Import Directory
    (c) Applying some heuristics to evaluate if the current instruction is the OEP[6]

The result of our tool is a set of memory dumps or reconstructed PEs depending on the result of the IAT fixing phase and a report which includes the values of each heuristic for every dump. Based on these information we can choose the best dump, that is the one that has the greatest chance of work.

3.3 Approach details

In this section we are going to describe in details the steps introduced in the previous section.

During the development we have adapted our approach in order to increase speed and effectiveness of our tool. Following there is a detailed explanation of our improvements on the initial approach:

1. in the first step, we add the option of not to track writes of library instructions on the stack and in the TEB[7]
2. in the second step we filter instructions of known libraries before dumping
    (a) when trying to reconstruct the IAT we added some code in order to deal with obfuscation techniques like *IAT Redirection* and *Stolen API*

---

[3] Write xor eXecution
[4] Portable Executable
[5] Import Address Table
[6] Original Entry Point
[7] Thread Environment Block

(b) we implemented five heuristics:
  - entropy: check if the value of the entropy is above a certain threshold
  - long jump: check if the "distance" between the current EIP and the previous one is above a certain threshold
  - jump outer section: check if the current EIP is a different section from the one of the previous EIP
  - pushad popad: check is a *pushad-popad* pattern has been found in the trace
  - init function calls: check if the imports of the dump are functions commonly used by the malware and not by the unpacking layers

For the instructions that execute from the same write set we adopted the following approach: if the "distance" between the current EIP and the EIP of the previous instruction is above a given threshold then we do the same as if we were in the case 2, otherwise we jump to the next instruction.

Finally, we have noticed that dumping only the main executable in memory is not enough because some packers dump the final payload on the heap. In order to deal with it, we track heap allocations and writes inside an heap interval. If necessary, we dump these intervals too.

3.3.1 Instructions filtering

Since our tool works with an instruction level granularity, limiting our analysis to the relevant instruction of the program is a critical point. For this reason we have introduced some filters, based on the common behaviours showed by the packers, which make our tool ignore the effects of a set of instructions. More in detail two kind of instructions are not tracked by default:

- Write instruction on the TEB and on the Stack
- Instruction executed by known Windows Libraries

The write instructions on the stack are ignored because unpacking code on the stack is not common compared to unpacking it on the heap or inside the main image. Moreover many instructions write on the stack and keeping track of all the written addresses would downgrade the performances gaining little advantages against a very small set of packers.

The same considerations can be applied to the instructions which write on the TEB, since most of these writes are related to the creation of instruction handlers and there is very little chance that a packer uses this addresses to unpack the encrypted payload.

The instructions executed by known Windows Libraries are never considered when checking if the current instruction address is contained inside a *Write Interval* because this would mean that the packer writes the malicious payload in the address space of a know Windows Library. This behaviour has never been identified in the packer analysed; moreover, this could introduce some crashes if the application explicitly use one of the functions which have been overwritten.

3.3.2 Track WxorX instructions

All packers, in order to work correctly, have to present a common behaviour: they must write the original program in memory and then they must execute the written code. We have defined an instruction that is first written by the packer and then executed as a *WxorX instruction.* Our tool tracks each write operation made by the program and build a set of memory ranges which identify contiguously written memory addresses. A memory range is dentified by a starting address and an end address and it is managed in order to take into account the memory operation that overlaps it (The starting address and the end address are decreased or increased respectively when an overlapping writes is detected).

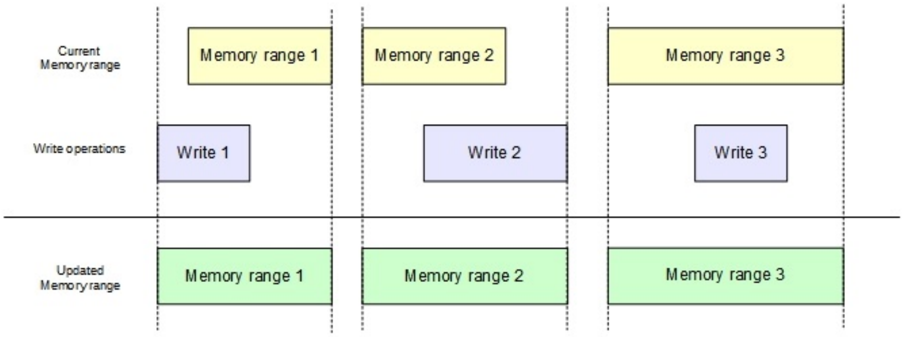


Figure 3.1: Memory range management

The first time we detect that the instruction pointer is inside one of our memory ranges, all the analysis and dumping routines (explained in the next chapters) are triggered and the memory range is marked as *broken.* It is easy to deduct that the definition of a broken memory range is : A memory range in which the execution has been redirected to.

3.3.3 Heuristics description

Our tool makes a dump each time the WxorX law is broken in a new *Write Interval* or in an existing one if *InterWriteSet* analysis is enabled. Consequently, at the end of the execution we may have a lot of dumps and check them manually can be very time-consuming. Heuristics help in automatically identify the dumps that are most likely to work and also provide a "best dump", the one with the greatest chance to be the contain the original unpacked code. All the heuristics are described in [**?**].

Entropy can be considered as a measure of the disorder of a program. For example, random data have the highest possible entropy. Since encrypted data more closely resembles random data, entropy can be used in detecting if an executable has been compressed, encrypted or both. Usually compressed files have less entropy that the original files, because the purpose of compression is to reduce the dimension of the file preserving some patterns used to recover the information. Encryption, on the opposite, has the goal of making unreadable a file, that is ideally changed in a random stream of bits. For example:
FFFFFFFFFFFFFFFFFFFF ——>20'F' (compressed)
FFFFFFFFFFFFFFFFFFFF ——>;LAKSDFJA;WIEFEJ;AEJF (encrypted)
As a result, while decompressing a file, its entropy slightly increase; while decrypting a file, its entropy slightly decrease. Consequently, in order to identify that a decompression or a decryption stage is finished, we have to check the absolute value of the difference between the initial and the current entropy.

It is very uncommon that the unpacking stub is located just under or above the OEP. Instead, once the original code is fully unpacked, there is a long jump from the unpacking stub to the OEP of the unpacked code. Sometimes, this jump has the target address is in a completely different section from the start address. This technique is called *tail jump* and may be a sign of the completion of an unpacking stage. In Figure 3.2, the cases (a) and (b) are extremely rare, while case (c) is the most common.
We base on these considerations the *long jump* and *jump outer section* heuristics.

Usually packers employ a technique called *pushad-popad*. The *pushad* function stores on the stack the values of all the registers and the *popad* function restores them in the registers. After a *pushad*, a packer can execute its unpacking routine and soon before the *tail jump* it can restore all the registers with a *popad*. In this way, the two instructions almost delimit the unpacking stub and help to identify the range of memory addresses in which the OEP is located. As an example, a commercial packer like *ASPack* employs this technique.

When a malware is packed, it usually exhibits very few or false imports. This is a technique of obfuscation used by some packers in order to hide the real behaviour of the malware. Of course, it is a job of the packer to fully reconstruct

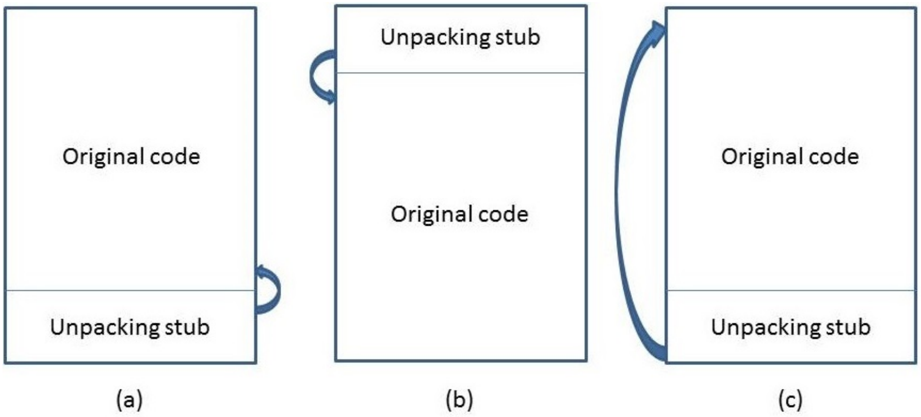


Figure 3.2: Different jumps from the unpacking stub to the original code

the IAT in order to make the binary runnable.

The purpose of the *init function calls* heuristic is to search in the IAT for functions commonly used by the malware and not by the unpacking stub. For example, if a malware has to contact an Internet domain to download some malicious code, it will have in its imports some internet communication APIs like *connect* and *send*. Since the unpacking stub does not need them to perform its job, when packed the binary will not have these APIs in its IAT, but the packer will take care of inserting them among the imports during the unpacking stage. In some extreme cases, the packed binary may have as imports only the *GetProcAddress* and *LoadLibrary* functions, because these two are used to dynamically load libraries and functions.

Consequently, once identified some "interesting" APIs for the malware but not for the packer, we check how many of them a dump has among its imports. Usually, the more of them, the higher the probability of being the correct dump.

3.3.4 Evaluation of *InterWriteSet* analysis

The initial approach was to dump the code and trigger the analysis of if once the WxorX law was broken in a *Write Interval* for the first time and then ignore all the subsequent instructions from the same *Write Interval*. However, when analysing a binary packed with *mpress*, we noticed the behaviour in Figure 3.3 that forced us to slightly modify the initial approach.

When *mpress* enters for the first time the *Write Interval* which contains the original code (a), it jumps in an area of memory that contains a small stub that

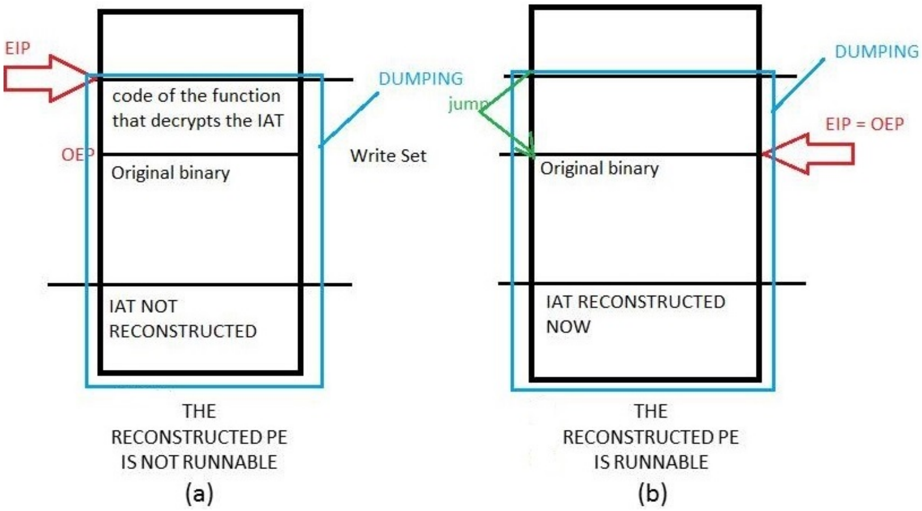

EIP
code of the function that decrypts the IAT
OEP
Original binary
IAT NOT RECONSTRUCTED
DUMPING
Write Set
THE RECONSTRUCTED PE IS NOT RUNNABLE
(a)
jump
Original binary
IAT RECONSTRUCTED NOW
DUMPING
EIP = OEP
THE RECONSTRUCTED PE IS RUNNABLE
(b)

Figure 3.3: Mpress behaviour

reconstructs the IAT and then jumps to the original code (b), that resides in the same *Write Interval*. With the original approach we were able to dump the code in (a), but we ignored the jump in (b) and so we were unable to recreate a running PE.

Consequently, we adapted our approach to deal with this behaviour: we consider jumps inside the same *Write Interval* only if they are longer than a given threshold. If this is the case, we trigger the analysis as if it was the first time the WxorX law is broken. However, this improvement may potentially generate too many dumps, making the analysis very slow and difficult. For this reason there is a maximum number of jumps that are considered inside the same *Write Interval*.

Chapter 4

Implementation details

4.1 System architecture

Our tool is entirely based on PIN, a binary instrumentation framework developed by Intel. It lets us to have the instruction-level granularity useful to track memory writes on a finer grain. In this way we are able, for example, to see where a single assembly write instruction is going to write and consequently create the write sets.
We have integrated Scylla, an external open source program, to dump the code and reconstruct the IAT. Moreover we have extended it in order to deal with *IAT Redirection* and *Stolen API* techniques.
Finally we use the IDA Pro disassembler and an IDAPython script in the *Init function calls* heuristic. The script calls IDA which reads the imports of the dump and compare them to a list of functions commonly used by the malware and not by the packer (registry manipulation, internet communication).

4.2 System details

In this section we are going to explain in detail the implementation of the most important parts of our tool.

4.2.1 WriteSet management

We introduce the concept of *WriteInterval* in order to group together contiguous writes to check if an instruction executes from a previously written memory area. All the *WriteIntervals* are grouped together in a *WritesSet*, a simple C++ vector.

A *WriteInterval* is a C++ structure with the following fields:

- addr_begin: start address in memory of the *WriteInterval*
- addr_end: end address in memory of the *WriteInterval*
- entropy_flag: flag used by the *Entropy Heuristic*
- long_jmp_flag: flag used by the *Long Jump Heuristic*
- jmp_outer_section_flag: flag used by the *Jump Outer Section Heuristic*
- pushad_popad_flag: flag used by the *Pushad Popad Heuristic*
- broken_falg: flag which indicates if the WxorX law has already been broken in this *WriteInterval*
- detectedFunctions: flag used in conjunction with the *Init Function Call Heuristic*
- cur_number_jmp: current jump number, used to properly name the result file (see Section 4.2.3)
- heap_flag: flag that indicates if the write is on the heap

For more information about heuristic see Section 4.2.5.

The following steps explain how *WriteIntervals* are created and updated:

1. for each instruction we check if it is a write
2. if so, we insert a *callback* function before it. A *callback* is a feature of PIN: it allows to instrument the code by inserting some code that will be executed before or after the original instruction. In our case, we intercept the write instruction and before the execution we retrieve its EIP[1], the address where it will write and the size of the memory that will be written. With these information we compute the start and end addresses of the write
3. Now we proceed to the construction or the update of the *WriteInterval*. We have five cases:
    (a) the memory written by the instruction neither is contained nor overlaps with another *WriteInterval*. In this case we create a new one and add it to the *WritesSet* vector

---

[1] Extended Instruction Pointer

(b) the start address of the write is before the start of a *WriteInterval*, but the end address is inside it. In this case we update the *WriteInterval* setting as start address the start of the write, but leaving unaltered the end address

(c) the same as case (b), but this time regarding the end address. Consequently, we only update the end of the *WriteInterval*

(d) the memory written by the instruction completely contains a *WriteInterval*. In this case we update both the start and the end of the *WriteInterval*

(e) the memory written by the instruction is completely contained by an existing *WriteInterval*. In this case we do nothing

We check each instruction, including writes, to see if it executes from one of the *WriteIntervals*. If this is the case, then we proceed with our analysis; in the other case we execute the instruction and go to the next one. In both cases the *Write Intervals* are preserved, the reason will be clear in Section 4.2.3.

4.2.2 Hooks of functions and syscalls

In order to make our tool properly work we have to hook some functions and system calls. We do this by inserting *callback* functions before or after the original instruction and more rarely by completely replacing the original routine. We always try to hook functions at the lowest possible level.

When dumping code, we need to include also the code on the heap, because some packers unpack code in that memory area. In order to do this, we have to track all the heap allocations and deallocations in order to create an *heapzone* in our tool.
An *heapzone* is an abstraction of a heap area and is implemented as a C++ structure with the following fields:

- begin address
- end address
- size of the *heapzone*

In order to manage the *heapzones* we have to hook the following functions:

- *RtlAllocateHeap*: used to allocate a heap region. We insert a *callback* function after the original one so we are able to retrieve the returned address where the heap has been allocated and its size. With these information we are able to create the *heapzone*

- *RtlReAllocateHeap*: used to reallocate a heap region. We use the same callback as the previous case, except that we can insert it before the original function because the address to be reallocated is passed as an input parameter
- *VirtualFree*: used to free a heap region. We insert a *callback* before the original instruction and we retrieve the address of the heap area that will be freed. We check if this address corresponds to one of our *heapzones* and, if this check is positive, we remove the corresponding *heapzone*

4.2.3 Dumping module

The dumping module takes care of creating the dumps and trying to reconstruct the *Import directory*.
When we find out that an instruction executes from one of the *Write Intervals*, we have two options:

- this is the first instruction which executes from this *Write Interval*. In this case we trigger the analysis on the entire block of memory and we mark it as *broken*
- this is not the first instruction which executes from this *Write Interval*, in other words the *broken* flag has already been set. In this case, if the *InterWriteSet* flag is enabled we proceed with the *InterWriteSet* analysis

In the first case the analysis goes through the following steps:

1. we call *Scylla* as an external process. The reason why we do it is that we have noticed that if we call *Scylla* inside the tool, a particular memory configuration may cause it to crash
2. Scylla tries to create a dump with the current EIP as the OEP. The dump eventually created is inside a folder named *NotWorking* because it is not runnable, since the *Import directory* is not still reconstructed
3. Scylla tries to find the IAT inside the current process
4. if Scylla succeeds in finding the IAT then is tries to reconstruct the *Import directory* in the not working dump
5. if it succeeds then the dump is moved in the main folder, otherwise it is left inside the *NotWorking* directory. In this way, even if the dump is not runnable, we can eventually access the code of the malware if we have correctly found the OEP

In the other case we eventually have to proceed with the *InterWriteSet* analysis. This kind of analysis is the same as the previous one except that it considers jumps inside the same *Write Interval*: if the absolute value of the difference between the current EIP and the previous one is greater than a given threshold and if the maximum number of jump in the *Write Interval* has not been reached, we consider the current EIP as a new candidate for the OEP and trigger the same analysis as before. This is also the reason why we do not remove broken *Write Intervals* from the *WriteSet* as soon as they become *broken*.
We did a survey to establish a reasonable threshold and we set it as 20% of the current *Write Interval* length. The maximum number of considered jumps has to be set by command line, as well as the flag that enables the *InterWriteSet* analysis.

In both cases the obtained dump, no matter if it is working or not, is subject to the analysis of our heuristics (see Section 4.2.5). Moreover we keep track of the number of dumps by incrementing a counter every time Scylla tries to dump the code, even if it does not succeed.
During the development of our tool we noticed that some packers unpack code also on the heap. For this reason, along with the main executable image we dump also the heap, adding it as an additional section in the PE.

4.2.4 IAT Fixing Module

4.2.5 Heuristics implementation

We use heuristics in our tool in order to evaluate the obtained dump: each heuristic can set a flag in the final report and all the flags contribute to identify the best dump, as explained later in this Section.
We have implemented five heuristics:

- entropy heuristic: at the beginning of the analysis, when the main module of the binary is loaded, we get its original entropy value. Each time a dump is created we compute again its current entropy and compare it with the initial one. Then we use the following formula to compute the difference:

$$difference = \left| \frac{current\_entropy - initial\_entropy}{initial\_entropy} \right| \tag{4.1}$$

  If this value is above a given threshold we set the correspondent flag in the output report.

In order to estimate the threshold we created a simple program which executes some writes and then we packed it with the most common packers. We then calculated the entropy before and after the packing process and their difference. The histogram in Figure 4.1 shows the result. A threshold of 0.6 is sufficient to cover almost all the packers

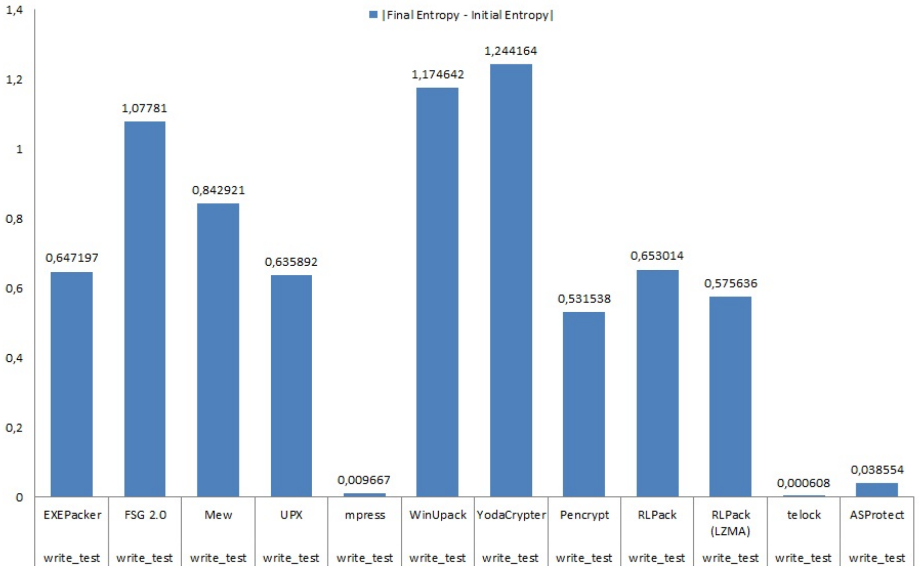


Figure 4.1: Entropy Threshold Survey Results

- jump outer section heuristic: for each executed instruction we keep track of the EIP of the previous one. In this way we can retrieve the section in which the previous instruction was located and compare it to the section of the current one. If these two are not equal we set the *jump outer section* flag
- long jump heuristic: as in the previous case we take advantage of tracking the previous instruction's EIP. In this case we simply compute the difference between the previous and the current EIP:

$$difference = |current\_eip - previous\_eip| \tag{4.2}$$

  If this difference is above a given threshold we set the *long jump* flag in the output report
- pushad popad heuristic: during the execution of the binary we have two flags indicating if we have encountered a *pushad* or a *popad*. For every

instruction we check if it is one of the previous two and if so we set the corresponding flag. Then, when we produce a dump, if both flags are active, we set the *pushad popad* flag in the output report

- init function call heuristic: the aim of this heuristic is to search through the dumped code for calls to functions commonly used in the body of the malware and not in the unpacker stub. We achieve this result by using an IDAPython script: using IDA we are able to read the list of the imports of the dump and to confront it with a list of ”suspicious” functions selected by us. Then we count the number of detected functions and write it in the output report

At the end of the execution of our tool we have a report which contains a line for each dump that lists the results of every heuristic, as well as the dump number, a string that says if the *Import Directory* is probably reconstructed or not, the OEP, the begin and the end addresses of the *Write Interval* considered for the dump. We use these information to choose the best dump as follow:

- first we check if the *Import Directory* is probably reconstructed
- is so, we count the number of the ”suspicious” functions detected and the number of active heuristics’ flags
- if the previous numbers are the best result until this moment, we save this dump number, eventually rewriting another one saved before
- at the end of the procedure we choose the saved dump number as the one that has the greatest chance to work

If no dump has its *Import Directory* reconstructed, we return the value ”-1”.

Chapter 5

Experimental validation

5.1 Goals

With our experiments we want to show the effectiveness of our tool. During the development we did some preliminary tests on normal programs: we packed them with common packers and tried our tool on them. Then results were pretty convincing, our tool correctly unpack sample programs packed with the following packers:

- UPX
- FSG
- Yoda Crypter
- mew
- mpress
- PECompact
- ASProtect
- WinUPack
- PESpin

Moreover, we are able to dump the program at the entry point, but not to reconstruct the IAT, for executables packed with:

- Themida, but a version without the anti-evasion flag activated
- Obsidium, but a version without the anti-debugging flag activated

We are able to produce not working dumps also for executables packed with ASPack.

5.2 Dataset

We built our dataset from the database of VirusTotal. Using a python script we were able to write a specific query to download only packed executables. We put these malwares in a shared folder between the host and the guest systems.

5.3 Experimental setup

In order to automatize we created a .bat script on the host machine that is able to restore, start, wait for 10 minutes and close the VirtualMachine using VBoxManage, the command line tool for VirtualBox.
At the start up of the Virtual Machine, a python script is set to automatically move a malware sample from the shared folder into the Virtual Machine and then trigger the execution of PIN with all the command line arguments to properly analyse the sample. After 5 minutes, it eventually stops the execution of PIN.
The final results are then moved into the shared folder with the host, in order to not lose them at the restoring of the Virtual Machine.
In conclusion, our system goes through the following steps:

1. download samples from VirusTotal and put them in the shared folder between the host and the guest machines
2. run the .bat script from the host machine which manage the Virtual Machine
3. the .bat script restores the Virtual Machine to a clean state and starts it
4. the start of the Virtual Machine triggers the execution of a python script that move the first of the samples in the guest system and starts PIN
5. after 5 minutes if PIN is still running it is stopped
6. the python script moves the results of the analysis in the shared folder
7. after 10 minutes the .bat script running in the host system stops the Virtual Machine
8. we go back to Step 3

5.4 Experiments

Chapter 6

Limitations

The main limitations of our work are:

- we do not handle packers that decrypt part of the code and after the execution re-encrypt it
- we do not handle spawning of new processes, dropping of new malware, downloading and executing of new binaries, DLL injection
- if there is an IAT obfuscation technique that we can not handle we still produce a dump, but it will not be runnable
- the packers do not write and execute directly on the stack
- there are not relative calls in the heap section, because we dump the heap and create a new section with the stuff discovered in it without patching any call to a relative address
- the binary analysed is a 32 bit binary

Chapter 7

Future works

Chapter 8

Conclusions

Appendix A