POLITECNICO DI MILANO
Corso di Laurea MAGISTRALE in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

NECST Lab
Novel, Emerging Computing System Technologies
Laboratory

Relatore:

Correlatore: Tesi di Laurea di:

Anno Accademico 2015-2016

To someone...

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 their code employing different techniques. This process, called *packing*, makes the static analysis of a binary completely useless. A *packer* is the tool that implements the previously described functions: it receives in input a binary, transforms and obfuscates its code/resources and then appends new codes that will *unpack* the original one runtime.

The complexity of *packers* can be very different: from those which write and execute directly the original code, to others that employ 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:

- Packing a binary destroys any possibility to understand what the program will do on the system without executing it. This problem, with the fact that packing a binary with different packers lead to different signature of the malware, voids any static analysis techniques implemented by AV.

- The process of reverse engineering a packed malware can be very time consuming and since lots of malware is pushed every day on internet there is the necessity of fast analysis and fast updating of AV software.

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 a professional malware analyst.

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.

- Hybrid unpacking: this mixes some static heuristics with dynamic analysis.

- Dynamic unpacking: this lets the unpacker do its work and tries 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 this case, the scope is not to reconstruct a binary from a packed one, but rather to stop malicious behaviours when they manifest.
In this area different works have been done, some of them are: OmniUnpack, JustIn.

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

- 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 don not care so much about safety because usually analysis is performed inside a controlled lab and the analysts want to observe the complete execution of malware. Also the performances are not a critical feature here because we are not constricted 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 fail to reconstruct a fully runnable binary given a packed one.

2.3   Goals and challenges

Since our work is born as a component of a bigger malware analysis platform (Jackdaw), our tool is oriented to help malware analyst. 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.
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   Approach overview

Our tool has an instruction-level granularity: each instruction is analysed and then goes trough the following steps:

1. check if it is a write instruction: we track the memory region in which the instruction writes, in order to create a list of memory blocks (we call them "Write sets") of contiguous writes.

2. check if instruction executes from one of the write sets, including write instructions. If this is the case we do the following things:

   (a) dump the code at this point of execution

   (b) reconstruct the IAT of the obtained dump

   (c) apply some heuristics to evaluate our dump

3. jump to the next instruction

In this way, at the end of the execution of the malware we have a series of dumps and a report which includes the results of each heuristic for every dump. With these information we can choose the best dump, that is the one that has the greatest chance of work.

3.2   Approach details

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

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*

   (b) our heuristics are:

       • 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 has been found in the trace

       • init function calls: check if the imports of the dump are function 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.

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

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

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 system.

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

Bibliography

Appendix A