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

Malwares usually pack their code in order to avoid static analysis of their payload. Basically their original code is transformed is some way and only when the malware executes it is restored. There can be more than one layer of packing, which means that a packed code can be packed another time. However, all the packers have one thing in common: when the program executes and the original code must be revealed, then they must first write in the memory area that contains the original code, in order to unpack it, and then execute from there. This sequence of operations is used by almost every unpacker as a metric to evaluate the unpacking process. There are a lot of common packers, UPX, FSG, PECompact, Themida, etc., but malware authors can also write custom packers.
Some packers implement a more complicated technique of unpacking: they unpack only portions of the original code and execute it, then they pack again this part and unpack the next portion. This kind of packers are outside the scope of this paper.

2.2 State of the art

There have been a lot of attempts to build an automatic unpacker. Some of them are standalone tools like PolyUnpack, Ether, Eureka; others work together with an antivirus, like OmniUnpack and JustIn.
There are other approaches: BCR, for example, does not aim to extract the original code, but the unpacking routine, and then use it to unpack the malware.

2.3 Goals and challenges

Our approach aims not only to unpack the malware, but also to reconstruct a working binary of it. To do so, we not only have to identify the original entry point (OEP) and dump the code at that moment of the execution of the malware, but we have to resolve all the imports, reconstructing its import address table (IAT).
The first thing we have to deal with are the unpacking routines of the packers: every time the execution of the malware is 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) in this step there are no improvements
   (b) when trying to reconstruct the IAT we added some code in order to deal with obfuscation techniques like *IAT Redirection* and *Stolen API*
   (c) 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 are 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 PIN. It lets us to have the instruction-level granularity useful to insert callback before and after instructions. In this way we are able, for example, to see where a write instruction is going to write and consequently create the write sets.
We have integratd Scylla, an external plug-in, to dump the code and reconstruct the IAT. However, we have modified 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 (regitry 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 of our tool we did some preliminary tests on normal programs: we packed them with common packers and tried if our tool worked. The results were pretty convincing, our tool correctly unpack sample programs packed with the following packers:

- UPX
- FSG
- Yoda Crypter
- mew
- mpress
- PECompact

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

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 our system we used VBoxManage, the command line tool of VirtualBox. We created a .bat script on the host machine that was able to restore, start, wait for 10 minutes and close the VirtualMachine using VBoxManage commands. At the start up of the Virtual Machine, a python script was 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 stopped the execution of PIN. The final results were then moved into the shared folder to the host, in order to not lose them at the restoring of the Virtual Machine.

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