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

Nowadays malwares employ different techniques to defeat analysis: evasion techniques are commonly used in order to detect analysis environments and to avoid triggering malicious behaviour when this checks succeed.
PIN is a binary instrumentation framework developed by Intel and is the base of our tool. This paper focuses on defeating evasion techniques of PIN.

2.2 State of the art

This is the first contribution to this research area.
Previous works have focused on trying to detect binary instrumentation framework, like the eXait library or the work DynamoRIO.

2.3 Goals and challenges

Our goal is to develop a framework which defeats some of the common evasion techniques against DBIs.
Our work starts with defeating the library of eXait: a list of techniques that are able to detect if a program is instrumented by PIN. These techniques go from fingerprinting the environment, searching for strings or code patterns, to timing attacks.
The final purpose is to produce a library that makes transparent a pintool.

Chapter 3

Approach

3.1 Approach overview

We have classified the DBI-evasion techniques into six main areas:

1. real EIP leak: DBIs usually copy instrumented code in a code cache on the heap. Consequently the real EIP of the program is different from the EIP expected by the application inside the main module. This group of techniques employs a single instruction to leak the real EIP in the code cache and check it against the expected one. We deal with these techniques patching the return value with the expected address inside the main module address space

2. detect JIT-related API calls: identify functions that are commonly used by JIT compilers and write a small routine at the beginning of them in order to check how many times are called. If this count is above a given threshold, then the JIT compiler is detected. We deal with it marking some memory areas as "protected" and tracking the writes: if a write is inside one of this "protected" areas it is redirected into another place

3. detect page permissions: usually JIT compilers need to allocate a lot of pages with read-write-execute privileges. If the number of these allocations is above a certain threshold, then the JIT compiler is detected. We defeat it by hooking all the function that check or retrieve in some way page permissions information

4. memory fingerprinting: scanning the process' memory can discover some DBIs' artefacts or code patterns on the heap. We defeat these techniques creating a whitelist of memory areas in which the process is allowed to read. Then we instrument all the reads and return fake values if the read address is outside the whitelist

5. timing attacks: since DBIs take longer time than normal execution to instrument code, this delay can be used to detect them. We hook every instruction that can read time and divide the results by a configurable divisor in order to lower the results

6. detect by parent process: a DBI is the parent process of the binaries being instrumented. Consequently, if a program discovers that the parent process is different from *cmd.exe* or *explorer.exe* it can detect a DBI. We deal with it faking the list of processes and avoiding to open processes like *csrss.exe* that leak the processes' list

3.2 Approach details

Chapter 4

Implementation details

4.1 System architecture

4.2 System details

In this section we are going to focus in detail about the implementation of each of the six points listed in the previous chapter:

1. The techniques used in order to retrieve the real EIP exploit some assembly instructions which when are executed need to save the execution context, including the instruction pointer value, in order to work correcly. Some of these are the instructions that trigger exceptions (E.g. INT 2E) or the instructions that save the floating point unit state (E.g. FSAVE). In order to deal with these thecniques we performed a single instruction check, and if we found one of these instruction, after its execution, we restore the context with a fake EIP value (The one that the program would have seen if)

2.

3. the eXait library uses the function VirtuaQuery to retrieve information about the page permissions. In order to avoid this we hooked the VirtualQuery and when the program queries pages outside the whitelist we fake the results and return that those pages are not mapped. Moreover, the VirtualQueryEx has the same function of the VirtualQuery but it allows to specify a handle of the process whose pages we want to query. However, if the handle corresponds to the current process, the VirtuaQueryEx does exactly the same things as the VirtualQuery. In order to prevent this, we patched also the VirtualQueryEx.

4.

5.

6.

Chapter 5

Experimental validation

5.1 Goals

5.2 Dataset

5.3 Experimental setup

5.4 Experiments

Chapter 6

Limitations

Chapter 7

Future works

Chapter 8

Conclusions

Appendix A