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 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 those that save the floating point unit state (E.g. FSAVE). In order to deal with these techniques we performed a single instruction analysis, and if we found one of these instructions, after its execution, we restore the context with a fake EIP value (The one that the program would have seen if it was run without PIN).

2. A common method used to detect a Just In Time compiler is to write a small routine at the beginning of ZwAllocateVirtualMemory (frequently used by JIT) which increments a counter every time it is called with the following two parameters :

   - AllocationType MEM_COMMIT and MEM_RESERVEs
   - Protect = PAGE_EXECUTE_READWRITE

   If the counter becomes bigger than a threshold the pin tool has been detected. We avoid this technique analyzing all the write operation made by the instrumented program, and when one of these is attempted in the

protected memory range, identified as the beginning of the API ZwAllocateVirtualMemory, it is redirected into another memory address.

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. To avoid the readings made by the instrumented process in the memory region in which reside dlls, code patterns or resources necessary to the DBI, we have built a whitelist with all the memory area in which a process is allowed to read (stack, sections, the runtime dynamic allocated memor, mapped file). If the process tries to access resources outside those in the whitelist, we return random data to avoid leak of PIN artifacts.

5. A process can retrieve the time from the system in three main different ways and we developed three techniques in order to lower correctly the time value returned :

   - Detect the reads on the KUSER_SHARED_DATA structure where tick count and clock cycles passed are stored and divided it by a configurable divisor.
   - Hook the system call NtQueryPerformanceCounter and fake, in the same way as the point above, the returned result using a different divsor.
   - Intercepted the rtdsc assembly instruction and fake the value returned in the registers EAX and EDX.

6. Based on a survey[1] of the most frequent techniques employed to discover the parent process we implemented some countermeasures in order to fake the list of processes :

   - Hook the system call NtQueryProcessInformationSystem in order to modify the returned process list changing the name pin.exe in cmd.exe
   - Hook the system call NtOpenProcess and, if the process that has to be open is csrss.exe return the status of the operation as NTSTATUS_ACCESS_DENIED

Chapter 5

Experimental validation

Chapter 6

Limitations

Chapter 7

Future works

Chapter 8

Conclusions

Bibliography

[1] j00ru//vx tech blog. Controlling Windows process list; 2009. Available from: http://j00ru.vexillium.org/?p=194.

Appendix A