SAPIENZA
UNIVERSITÀ DI ROMA

BlueTracer:
a Robust API Tracer for Evasive Malware

Faculty of Information Engineering, Informatics and Statistics

Master of Science in Engineering in Computer Science

Candidate

Simone Nicchi
ID number 1705157

Thesis Advisor

Prof. Camil Demetrescu

Co-Advisors

Dr. Daniele Cono D'Elia
Dr. Emilio Coppa

Academic Year 2017/2018

BlueTracer: a Robust API Tracer for Evasive Malware
Master thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: nicchi.1705157@studenti.uniroma1.it

Ai miei genitori, che non hanno mai smesso di supportarmi

Introduction

Short version of Intro chapter

Donec et nisl id sapien blandit mattis. Aenean dictum odio sit amet risus. Morbi purus. Nulla a est sit amet purus venenatis iaculis. Vivamus viverra purus vel magna. Donec in justo sed odio malesuada dapibus. Nunc ultrices aliquam nunc. Vivamus facilisis pellentesque velit. Nulla nunc velit, vulputate dapibus, vulputate id, mattis ac, justo. Nam mattis elit dapibus purus. Quisque enim risus, congue non, elementum ut, mattis quis, sem. Quisque elit.

Contents

1 Introduction **1**

2 Background **4**

2.1 Binary Rewriting 4

2.1.1 Import Address Table Patching 5

2.1.2 Export Address Table Patching 6

2.1.3 Proxy DLL 7

2.1.4 Inline Hooking 7

2.1.5 Debugger Based Hooking 9

2.2 Virtual Machine Introspection 9

2.3 Dynamic Binary Instrumentation 10

2.4 Conclusion 11

3 Architecture and Implementation **12**

3.1 Overview 12

3.2 Thread Management 13

3.2.1 Log Files and Multithreading 15

3.3 Native APIs Tracing 16

3.3.1 Main Image Check 18

3.3.2 Native API Name Resolution 20

3.3.3 Native API Information Representation and Retrieval 21

3.3.4 Native API Logging 24

3.4 APIs Tracing 30

3.4.1 APIs Instrumentation 31

3.4.2 API Analysis before Execution 39
3.4.3 API Analysis after Execution 41
3.5 Context Change Analysis 43
3.5.1 Callbacks Tracing 44
3.5.2 Windows Asynchronous Procedure Calls Tracing 46

4 Experimental Results **49**

5 Conclusions **50**
5.1 Future Directions 50

Chapter 1

Introduction

Malicious software (or malware) is any software specifically designed to bring harm to a computer system. The problem posed by malwares is one which is becoming increasingly important, as new and more sophisticated malwares arise every day and the economical damage for organizations keeps worsening [9]. To face this threat, professionals are typically aided by a range of automatic tools capable of analysing and detecting malicious software.

Malware analysis can be carried out either statically or dynamically. Dynamic analysis encompasses techniques that execute a sample and observe the actions it actually performs, whereas in static analysis the sample is examined without running it. Such techniques have evolved over time to keep track with the increasing complexity and diversity of malwares. However, in recent years, the shift towards automation, caused by the need of dealing with a huge and ever-growing number of samples, together with the rising complexity of obfuscation mechanisms utilized by malwares, has strongly favoured dynamic analysis.

One of the most employed dynamic analysis techniques is function call monitoring. Generally, a function is made up of code which carries out a particular task, like for example creating a file or printing a message. Although the utilization of functions allows for easy re-usability of code and simpler maintenance, the propriety which makes them particularly valuable from a program analysis perspective is that they abstract the implementation details, providing a semantically richer representation of some functionality. For instance, let us consider a sorting function; it might not

be important to know the underlying sorting algorithm as long as it is known that the function sorts the input number set. In the context of dynamic analysis, the abstractions provided by API calls and system calls (or eventually Windows Native APIs) are incredibly helpful since they can be used to grasp the overall behaviour of the sample being analyzed.

The typical technique used for function calls monitoring in dynamic malware analysis is *API hooking*, i.e. the interception of function calls provided by DLLs. The idea is to alter the original sample so that, besides the function of interest, a *hooking* function is also called, which is in charge of performing the wanted analysis, e.g. logging the function invocation on a file or analyzing the function's parameters [16].

A problem that all dynamic analysis techniques have to face, including function call monitoring by means of API hooking, is the widespread of evasive malwares. Such malwares check whether or not they are being executed in an adverse environment and conceal their harmful behaviours accordingly, like for example by carrying out an exit sequence [10]. Unfortunately, such anti-evasion mechanisms are frequently adopted my malicious samples. According to Symantec's Internet Security Report of March 2018, 18% of new malware were virtual-machine-aware [18]. To make matters worse, the API hooking techniques presented in literature are easily detectable and are not coupled by any mechanism to hide their presence from evasive malwares.

Contributions. Throghout this thesis we present **BlueTracer**, a robust library and system call tracer for Windows applications, specialized in the monitoring of evasive malwares. BlueTracer is based on the Intel Pin [23] dynamic binary instrumentation (DBI) framework and is able to counteract malwares' anti-evasion measures thanks to its integration with BluePill, a software toolkit built on top of a DBI layer which allows the simulation of the execution environment a particular malware was designed for and conceals any virtualization artifacts and setup details which might set off evasion [10]. BlueTracer is capable of tracing the input values, the output values and the return values of an extremely wide range of system calls (including Windows Native APIs) and API calls. Moreover, it also supports

the tracing of Windows callbacks functions and Windows asynchronous procedure calls (APC). The tool was tested on a benign application aimed at assessing how good an anti-malware system is against evasion techniques and on actual evasive malwares, proving to be effective in both tracing the samples' activity and remaining undetected.

Structure of the Thesis. The remaining part of this thesis is structured as follows. Chapter 1 describes the major *API hooking* techniques present in literature, outlining their stengths and weaknesses, especially from a detection point of view. Chapter 2 introduces the concept of Dynamic Binary Instrumentation (DBI) and presents Intel Pin, the framework used to develop BlueTracer. Chapter 3 focuses on the implementation of the tool, on its structure and the design choices which were made during its development. Chapter 4 illustrates the experimental results and assesses the tool's effectiveness. Finally, we present concluding remarks in Chapter 5, together with possible future developments.

Chapter 2

Background

In literature there are many different implementations of API hooking. The objective of this chapter is to provide an outline of the various approaches utilized to hook functions in DLLs, outlining the benefits and the limitations of each technique, with a strong focus on their detection by malicious software. In particular, the focus will be on user space API hooking of `Win32` binaries, since this is BlueTracer's current field of application. Obviously, as it is the norm in malware analysis, it also assumed that the program under study is only available in binary form.

Depending on their underlying implementation, API hooking techniques can be divided in three broad categories: **Binary Rewriting** based, **Virtul Machine Introspection (VMI)** based and **Dynamic Binary Instrumentation (DBI)** based.

2.1 Binary Rewriting

Binary rewriting based hooking involves inserting hooks at the API entries, via one of the following two approaches:

1. Redirecting all `call` instructions so that the hook is called instead of the original function.

2. Rewriting the function of interest such that, before its invocation, the hook is executed.

In both cases the hook function gains access to all the arguments present on the stack, thus being able to carry out all the required analysis operations.

The main techniques which use the first approach are *Import Address Table (IAT) Patching*, *Export Address Table (EAT) patching* and *Proxy DLL*. On the other hand, the most significant techniques which use the second approach are *inline hooking* and *debugger based hooking* (*Figure 1.1*).

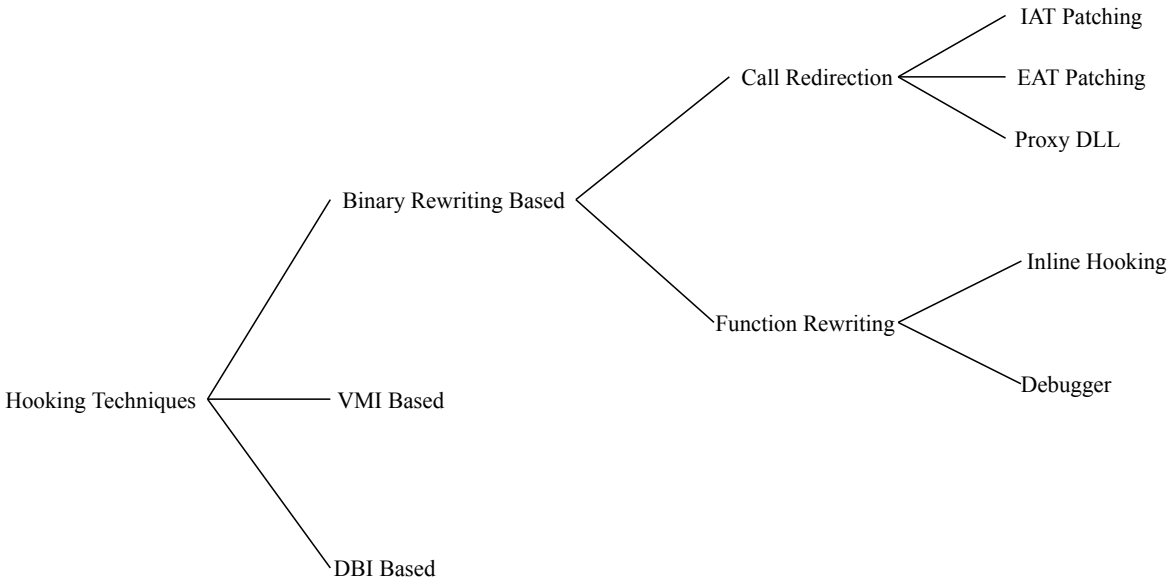

Figure 2.1. *API hooking techniques classification*

2.1.1 Import Address Table Patching

In the header of every Portable Executable (PE) file there is an Import Address Table (IAT) for every dynamic-link library (DLL) that is included by the executable [4] (*Figure 1.3*). This table is utilized to indicate the location of DLL-imported functions in virtual memory and is filled by the Windows loader with the actual function memory addresses after the executable is loaded in memory.

The idea is to overwrite the original pointer to an imported API function so that, instead of pointing to the original API, it will point to a different function.

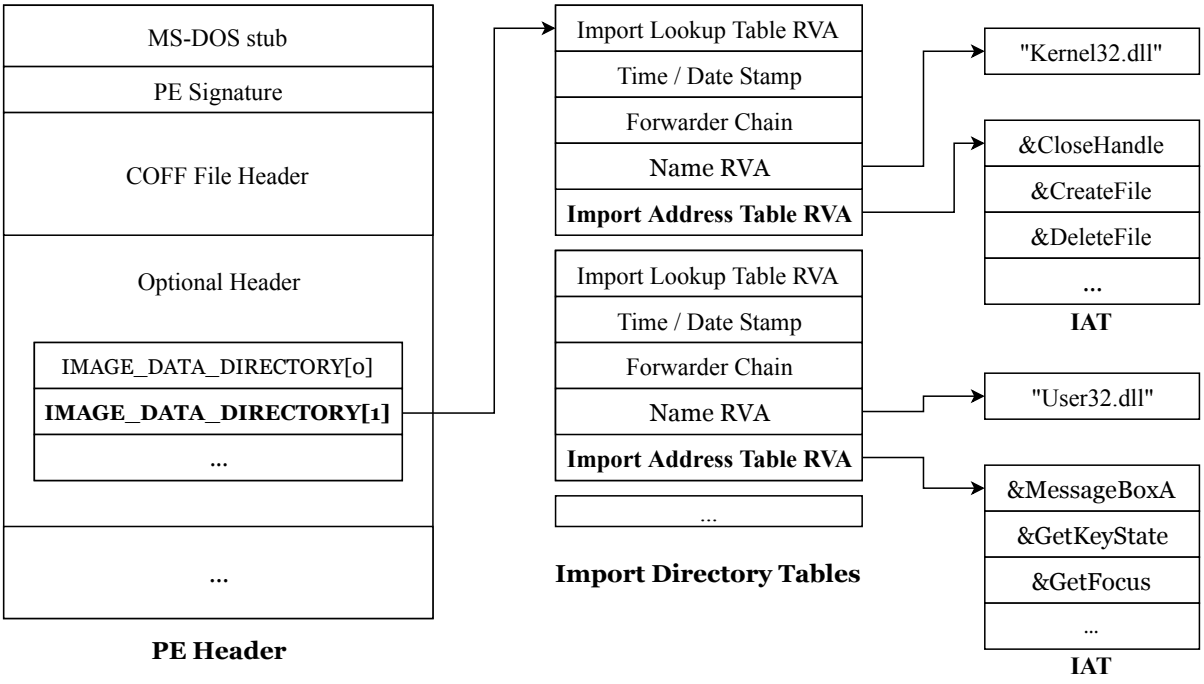

Figure 2.2. *IAT in PE header*

Despite being extremely simple to implement, IAT patching suffers from a couple of disadvantages, which significantly limit its use in practice:

- It is incredibly easy to detect by simply examining the entries of the IAT and checking whether or not each address falls inside the memory range of the DLL that should contain the function [22].
- It is ineffective when function pointers are acquired dynamically, e.g. via `LoadLibrary` and `GetProcAddress` [5].

2.1.2 Export Address Table Patching

Export Address Table (EAT) patching is similar to IAT patching, with the difference that DLL export address tables are patched instead. The export address table (EAT) contains the name of every function exported by the DLL together with the relative virtual address (RVA) where the function can be found, which is relative to the DLL base address when loaded in memory. To hook an API function via EAT patching all that is needed is to overwrite the corresponding address in the table with the address of another function.

EAT patching produces similar results to the ones obtained through IAT patching, but, unlike IAT patching, the created hooks are global, i.e. they affect every program which utilizes the altered DLL [4].

However, in a similar manner to what occurs for IAT patching, it can be easily detected to by simply examining the entries of the EAT and checking whether or not each RVA, when added to the DLL base address, falls within the DLL memory range [29].

2.1.3 Proxy DLL

In the Proxy DLL approach to hooking, also known as Trojan DLL, the DLL containing the functions to be hooked is replaced with another one having an identical name and exporting all the symbols of the original DLL [19]. In addition to calling the original functions so that they can carry out their tasks, the Proxy DLL may also make available different implementations for the hooked functions [4].

Even though a Proxy DLL is trivial to implement, it is also extremely easy to detect since the original DLL is substituted with another file, which is very likely to have a different size. Moreover, checksums could be employed to detect the presence of a Trojan DLL.

2.1.4 Inline Hooking

In *inline hooking* the API to be hooked has its initial instructions (at least the first 5 bytes) overwritten with an unconditional jump to a replacement function. In order to ensure that the API's original functionality is not lost due to the modification of its entry point, a *trampoline function* is created, consisting of a copy of the overwritten instructions and an unconditional jump back to the unaltered portion of the original function. As a result of this, the replacement function can invoke the original function by calling the trampoline, after performing all the desired analysis operations [4]. *Figure 4* illustrates a program's execution flow before and after the use of *inline hooking*.

Inline Hooking, which was made famous by its employment in the Microsoft *Detours* Windows API hooking library, is one of the most used API hooking techniques

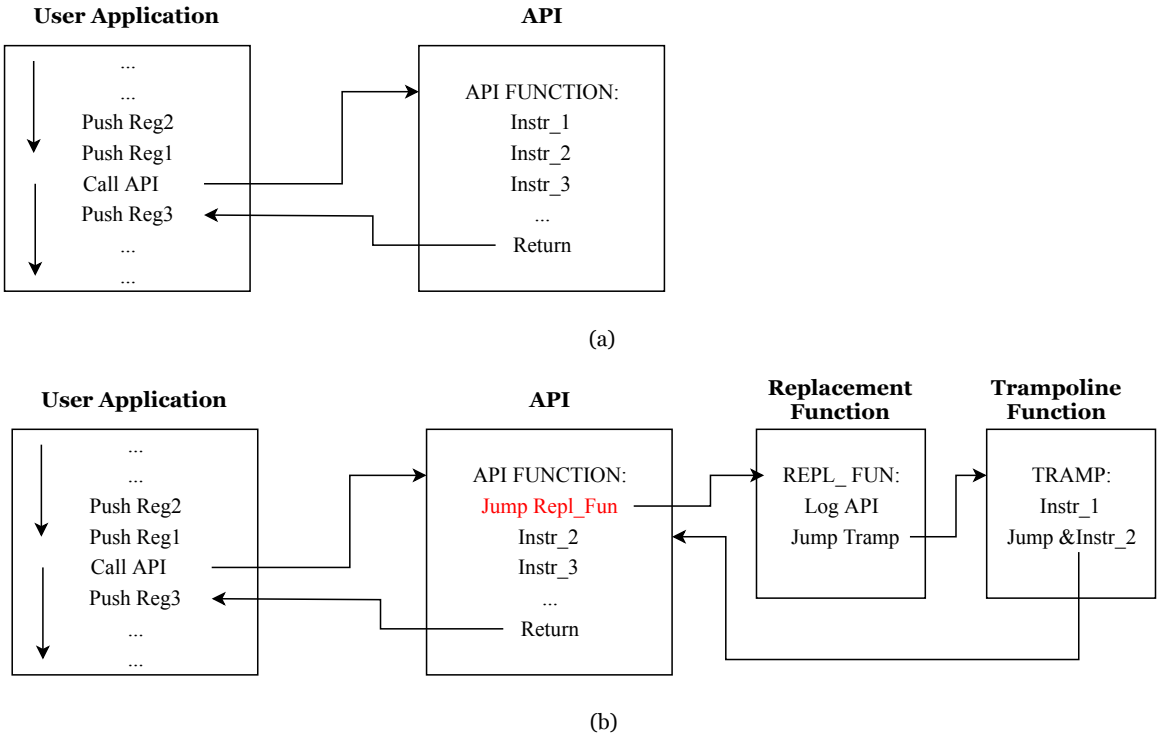

Figure 2.3. *(a) Ordinary API call execution flow*
(b) API call with inline hooking

since it offers a number of advantages:

- It is fast and efficient.
- It can be utilized to hook any code, not just operating systems APIs, but also programmer defined functions [28].
- Unlike IAT patching, the type of command used to call the function does not matter, meaning that the hooking will be effective regardless of the fact that a function is called using the IAT or using `LoadLibrary` together with `GetProcAddress`.

Unfortunately though, *inline hooking* is also affected by some limitations:

- Can be easily detected, for instance by comparing the code section of system libraries in memory with a matching original copy loaded from the file system to detect library modifications [5] or by searching API entry points for specific patterns (e.g. presence of `jmp` instructions) [22].

- It needs additional modifications in the case where the function's entry points includes specific instructions, like ones which contain relative memory addresses. In fact, such instructions cannot be executed from a trampoline as the trampoline is located in a different memory location than the one of the original program code [4].

2.1.5 Debugger Based Hooking

Hooking through the use of a debugger is realized by instructing the debugger to position a breakpoint at the entry point of the target API function. The placement of a breakpoint involves overwriting the initial instructions of the target API functions with CPU specific instructions, like `INT 3` for `IA-32`. These lead the CPU to throw a debug exception in case they are pointed by the current instruction pointer (IP). The exception is then intercepted by the debugger, which is able to deduce the API which is being called by the application from the address at which the exception took place [22]. Moreover, the debugger also has total control over the memory contents and the CPU state of the process being debugged.

Contrarily to inline hooking, a debugger can be used to hook functions whose entry points include instructions containing relative addresses [4].

On the other hand, a debugger is much easier to detect. In fact, there exist specific Windows APIs whose purpose is to find out whether or not the current process is being debugged. For example, `IsDebuggerPresent` allows to determine if the calling process is running under a debugger, while `CheckRemoteDebuggerPresent` checks for the presence of a debugger in a separate process. In addition, the `INT 3` instruction in an API entry point immediately gives away the debugger's presence [22].

2.2 Virtual Machine Introspection

Virtual Machine Introspection (VMI) based hooking relies on the idea of executing the target program in an emulated environment, typically with QEMU being used as virtual machine monitor (VMM). Function calls are monitored by comparing the

virtual processor's instruction pointer with the RVAs of DLLs' exported functions when added to the DLL base address. Function arguments are also monitored and this is done by providing them to callback routines, which perform the appropriate tracking operations.

In theory, a PC emulator allows to have functionalities similar to the ones of a debugger, i.e. the code being monitored can be stopped at any arbitrary point during its execution, allowing its registers and virtual memory to be inspected, with the added advantage of not being subject to the aforementioned issues related to breakpoints. Moreover, VMI based hooking is harder to detect with respect to the previously illustrated hooking techniques, since emulation is utilized to execute an unknown binary with a complete operating system in software, without the sample being never ran directly on the processor [3].

The significant drawback of VMI based hooking is that it incurs in the *semantic gap* problem, i.e. the issue of deducing high-level information from the raw system information by making sense of the CPU state and memory contents [16]. VMI based hooking tools might need an in-depth knowledge of kernel data structures or other details at low-level, which could constitute a complication when dealing with proprietary operating systems. For this reason, as of right now, VMI is not as effective in practice as a traditional debugger when investigating a sample.

2.3 Dynamic Binary Instrumentation

Dynamic Binary Instrumentation (DBI) is an analysis technique in which the behavior of a binary application is inspected at run-time via the injection of instrumentation code. Such code, after being injected, executes as a component of the ordinary instruction flow, allowing to learn information about the behavior and the state of a sample at different points during its execution [13]. We will further elaborate on DBI throughout Chapter **??**. *add ref*

In DBI based hooking, the learnt information refers to which APIs are called and, possibly, with which arguments and return values.

There indeed exist DBI based API tracing tools that rely on the previously illustrated idea, namely *drstrace* and *drltrace*, which are both built on top of the

DynamoRIO [14] DBI framework. In particular, *drstrace* is a system call tracer for Windows, while *drltrace* is an API calls tracer for both Windows and Linux applications. These tools, however suffer from two notable drawbacks:

- They are not equipped with any mechanism aimed at cloaking the execution environment in order to prevent a malicious sample from detecting the DBI.
- They are limited in the amount of information recorded relative to the traced APIs. This applies particularly to *drltace*, which, unlike *drstrace*, does not log return values and output values for arguments, in addition to not providing a mechanism for translating enumerations' constants to the appropriate name. Furthermore, both tools do not take into consideration Windows callbacks and asynchronous procedure calls (APC).

2.4 Conclusion

In this chapter we showed how the state of the art API hooking techniques suffer from a number of remarkable shortcomings, especially when dealing with evasive malware. In fact, binary rewriting based hooking techniques are all easily detectable, while VMI, although harder to uncover, is affected by the *semantic gap problem*. Finally, existing DBI-based API tracing tools are not accompanied by adequate cloaking mechanisms and are limited in the amount of logged information. The aforementioned issues indicate that there is a need for a robust API tracer, specialized in the analysis of evasive malware and with extensive logging capabilities. This is the rationale at the heart of BlueTracer.

Chapter 3

Architecture and Implementation

In this chapter we discuss the implementation of BlueTracer, providing a detailed account on how the tool is organized, on the design choices which were made during its development process, on how encountered challenges were dealt with and on the decisions which were taken to improve performance.

3.1 Overview

This section requires significant extensions

BlueTracer, being a part of Blue Pill, is also implemented using Pin, a dynamic binary instrumentation framework by Intel, which is vastly utilized for program analysis, testing of software and in the security field. The version of Pin used for the development of the tool is 3.5, in order to benefit from the notable improvements, both in terms of execution speed and offered features, which were introduced going from the 2.14 release to the 3.x series. Pin comes with its own OS-agnostic and compiler-agnostic runtime, called PinCRT. PinCRT exposes three layers: a generic operating system interface providing basic OS services (e.g. process and thread control), together with C and C++03 (without RTTI) runtime layers, for writing instrumentation and analysis routines [23].

BlueTracer has been organized primarily taking into account the rich set of APIs offered by Pin, which have led to the decision to split the tool in three parts: the

first aimed at **native APIs tracing**, the second for **APIs tracing** and the last focused on **callbacks and APCs tracing** (*Figure 3.1*). In particular, the tracing of native APIs also employs a different source of API information (*Dr. Memory*'s system call data) than the ones utilized for tracing APIs (*drltrace*'s configuration file or the information extracted from *PyREbox*'s database).

This chapter will begin by describing how multi-threading was addressed in the tool, as this is central to all its components. Then, the implementation of each one of three aforementioned parts will be discussed in detail.

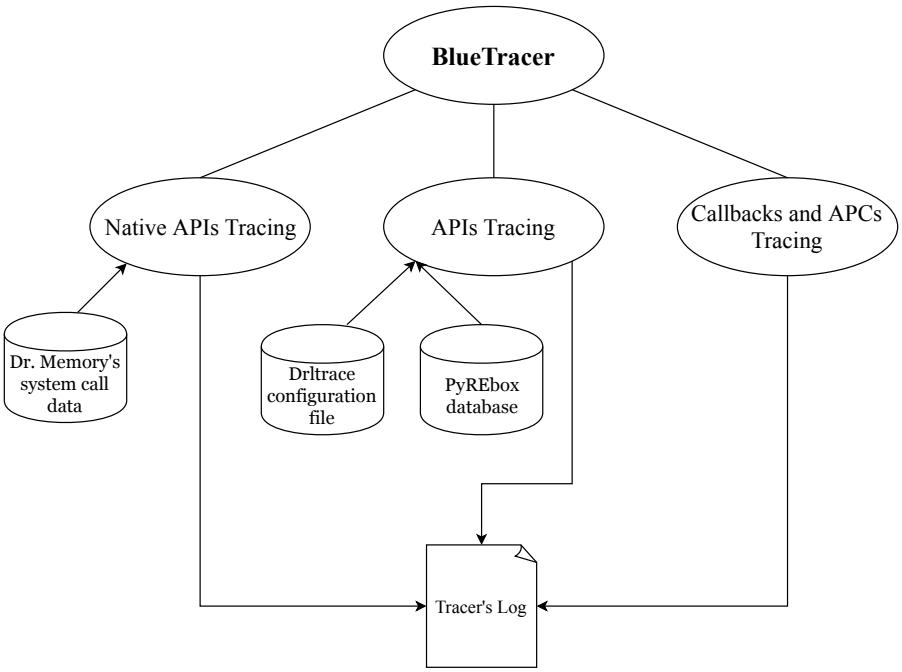

Figure 3.1. *BlueTracer's high level structure*

3.2 Thread Management

Since the samples under analysis are typically multithreaded applications, let us go through the mechanisms exposed by Pin to manage threads and how those were employed in the implementation of the tool.

Pin assigns to each thread an ID, a small number beginning at 0 which is not the same as the operating system thread ID. A way to obtain such ID is by using as analysis routine argument `IARG_THREAD_ID`, which passes the thread ID

assigned by Pin for the calling thread. This ID is typically used as an index of an array of thread data. In fact, the Pin API makes available an efficient thread local storage (TLS). In order to utilize it, it is first required to allocate a new TLS key via `PIN_CreateThreadDataKey`, which can optionally take as input a pointer to a destructor function. After that, any thread of the process can use the TLS key, in addition to its Pin-specific thread ID, to store (`PIN_SetThreadData`) and retrieve (`PIN_GetThreadData`) values in its own slot. The starting value relative to the key in every thread is `NULL`. Pin makes also available call-backs when each thread starts (registered with `PIN_AddThreadStartFunction`) and ends (registered with `PIN_AddThreadFiniFunction`). This is typically where thread local data is allocated, manipulated and stored in a thread's local storage[23].

In BlueTracer, each TLS slot stores a `struct` of the type `bluepill_tls` (*Listing 3.1*) for every thread. Such `struct` is dynamically allocated every time a thread starts in the `OnThreadStart` callback function and is consequently deallocated in the `onThreadFini` function when the thread ends.

```
1  typedef struct {
2   ...
3   syscall_tracer* syscallEntry;     // Pointer to NTAPI entry
4   vector<stackEntry>* shadowStack;  // Shadow stack
5   uint call_number;                 // Calls counter
6
7   buf_info_t* buffer;               // Buffer for writing to file
8   FILE* OutFile;                    // Output file pointer
9
10  // Pointer to function for opening file/writing to file
11  void(*file_write)(THREADID, buf_info_t*, FILE*, const char*,   ...);
12
13  ...
14 } bluepill_tls;
```

Listing 3.1. Thread Local Data

Since the first three fields of the above `struct` (lines 3-5) are employed when tracing native APIs and APIs, they will be discussed in detail later in the chapter.

Now let us focus on the remaining fields, which are used by BlueTracer to write the traced information in the appropriate log files.

3.2.1 Log Files and Multithreading

In BlueTracer, the traced data is written to a binary file, one for every thread. The default naming convention used for the tracer's log files is `Traced.[OS Process ID].[Pin-specific Thread ID]`, similarly to the one Blue Pill employs in its own log files, with the user being able to change `Traced` with a name of its choice in the configuration file.

When writing data to file, each thread invokes the `file_write` function, whose pointer is located in the instance of the `bluepill_tls` struct associated to the thread (line 11 of *Listing 3.1*). However, such data, which follows the same format of strings used by `fprintf`, is not directly written to file. Instead, an intermediate 8 kB buffer is used (line 7 of *Listing 3.1*): only when the buffer is full (or when the amount of data to be written does not fit the buffer) file writing actually occurs. The choice of using a buffer was made as an attempt to improve performance, as it allows the aggregation of small write operations into a block size that is more efficient for the disk subsystem.

A problem which was encountered when trying to conjugate file management and multithreading is that there exists a known isolation issue affecting Pin on Windows. Specifically, it is possible for a deadlock to take place if a file is opened in a callback in the context of multithreaded applications. As a result of this issue, it is not possible to open the tracer's log file in the `OnThreadStart` callback. Pin's manual proposes to circumvent the problem by opening the file in the `main` and tagging the data with the thread ID [23]. However, this conflicts with the idea of having one file for each thread.

In order to bypass this limitation of the Pin's framework, the following strategy was employed:

1. When initializing the thread local data in `OnThreadStart`, `file_write` is set to point to a function named `file_open`.

2. The first time a thread attempts to write data to file `file_open` is invoked.

3. `file_open` carries out the following actions:
   (a) Opens the tracer's log file (this is safe since the file is not opened in a callback)
   (b) Sets the obtained file pointer in the thread local data (line 8 of *Listing 3.1*)
   (c) Adds the data to be written in the buffer (which is eventually written to file if the buffer is too small to hold it)
   (d) Sets `file_write` to point to `buf_write`, a function which is in charge of just writing data to the buffer and to file.
4. As a result of this, when the thread attempts to write to file again, `buf_write` is invoked, thus allowing the thread to just write to file without going through opening the file again.

3.3 Native APIs Tracing

Windows Native APIs are employed to call operating system services in kernel mode in a controlled way. In fact, all core Windows components, which possess direct access to hardware and services in charge of handling the computer's resources (e.g. memory), operate in kernel mode. This means that, every time a user mode application desires to carry out certain actions, like for instance starting a thread or allocating virtual memory, they must rely on kernel mode services. The Windows Native API corresponds to the system call interface of standard monolithic operating systems, such as the majority of UNIX-like systems, with the difference that in the latter case the system call interface is documented and can be utilized directly by applications. Instead, due to Windows' architecture, Windows Native APIs are concealed to the programmer by the higher level Windows (`Win32`) APIs [25]. User mode Windows Native APIs, which are identified by their `Nt` prefix and are exported by `ntdll`, have caught the attention of malware writers since they are seen as a way of bypassing the documented APIs with the objective of performing a series of actions without being discovered [6]. For this reason it is a good idea to also trace them, in addition to the ordinary Windows APIs.

Pin provides a set of APIs aimed at assisting the extraction of information relative to the system calls made by the pinned application, also including information relative to Windows Native APIs. In particular, in Blue Pill (and consequently in BlueTracer) `PIN_AddSyscallEntryFunction` and `PIN_AddSyscallExitFunction` were used for this purpose. In fact, these allow to register notification functions which are called immediately before and after the execution of a system call [23]. In BlueTracer, the function in charge of gathering Native API information before execution is `TraceSysCallEntry` while the one responsible for collecting Native API information after execution is `TraceSysCallExit`. They have been both been embedded in Blue Pill's notification functions and their overall structure is detailed in *Figure 3.2*.

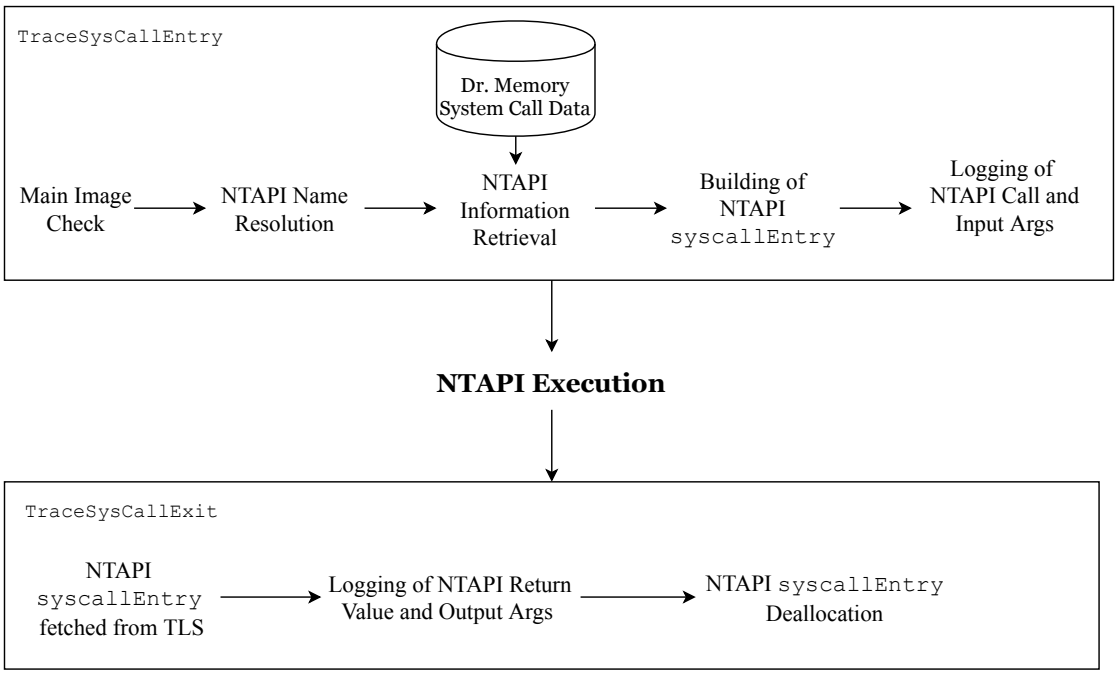

Figure 3.2. *Native APIs Tracing Workflow*

Following *Figure 3.2*, let us now thoroughly analyze the main steps which take place when Native APIs are traced.

3.3.1 Main Image Check

In order to filter the logged information relative to Native APIs, BlueTracer allows the analyst to decide, through the use of a boolean parameter (`MainImage`) in the configuration file, whether or not only Native APIs called invoked directly from the main executable of the pinned application should be traced. Therefore, as it can be seen in *Figure 3.2*, the first thing which is done in `TraceSysCallEntry`, assuming that the `MainImage` configuration parameter has been set to `true`, is to determine if the Native API call is taking place directly from the main executable.

To this end, Pin's `IMG` APIs are employed, where in Pin an `IMG` represents all the data structures relative to binaries and shared libraries [23]. Specifically, in Blue Pill, `IMG_AddInstrumentFunction` is utilized to register a callback which is invoked each time an image is loaded. Inside such callback, which takes as one of the input parameters the `IMG` object representing the image being loaded, the following steps are taken:

1. `IMG_IsMainExecutable` is employed to determine if the image being loaded is the main executable of the pinned application.

2. If this is the case, `IMG_HighAddress` and `IMG_LowAddress` are invoked. These two APIs return the highest address and the lowest address respectively of any code or data loaded by the image corresponding the the `IMG` object they take as input. By employing these APIs is therefore possible to determine the address range relative to the main executable.

Having obtained the main executable address range in this way, it is then employed in `TraceSysCallEntry` to learn if the Native API call is occuring directly from the main image. Internally, before entering kernel mode every Native API executes some common code, i.e. each Native API stores its ordinal in the `eax` register and invokes `KiFastSystemCall`, where the `sysenter` instruction is used to actually enter kernel mode. When `sysenter` is executed, the kernel obtains the ordinal number from `eax` and utilizes it to call the corresponding function, prior to going back to user mode [17].

In Pin, the Native API is intercepted right before `sysenter` is executed. This

could be inferred by the fact that, in `TraceSysCallEntry`, the instruction pointer (obtainable via the `PIN_GetContextReg` API) contains `sysenter`'s address. In light of this, in addition to the fact that each Native API calls `KiFastSystemCall` without setting a stack frame, the application's stack during the execution of `TraceSysCallEntry` is in the following state (*Figure 3.3*).

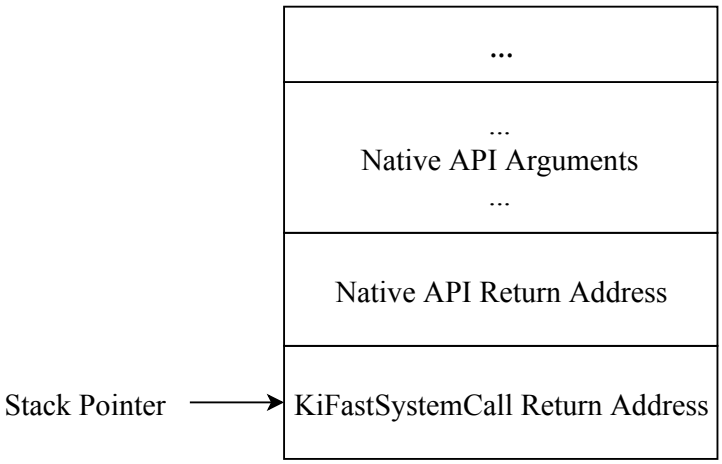

Figure 3.3. *Stack before* `sysenter` *execution*

Having outlined the general situation, it is finally possible to describe how the main image check is carried out:

1. The first check involves discovering whether or not `sysenter` is called directly from the main executable. As Pin intercepts Native APIs right before the execution of `sysenter`, this can be trivially done by checking if the instruction pointer value obtained in `TraceSysCallEntry` falls within the main executable memory address range.

2. Secondly, it is required to examine if the Native API return address belongs to the main executable. As a result of what shown in *Figure 3.3* such return address is obtained by adding 4 to the stack pointer and retrieving the pointed value, where the stack pointer can be obtained in Pin through the use of `PIN_GetContextReg`. The resulting value is then, once more, compared with the main executable memory address range.

3.3.2 Native API Name Resolution

Previously, it was stated how it is possible to register a callback function to be executed before Native APIs through the use of `PIN_AddSyscallEntryFunction`. Such callback functions, named `SYSCALL_ENTRY_CALLBACK`s, receive a set of parameters, including `ctx`, the application's register state immediately before the system call execution, and `std`, the system call standard. In Blue Pill, these two parameters are used to invoke `PIN_GetSyscallNumber` inside its `SYSCALL_ENTRY_CALLBACK`. Such API returns the number (ID) of the Native API to be executed in the provided context [23].

Blue Pill employs a mechanism which allows it to obtain the Native API name from its number, since the hooking functions it employs are indexed by name and not by ordinal. This choice was made because the identifiers vary depending on the Windows version, even among different Service Pack versions. For the same reason, the information needed by BlueTracer to correctly log Native APIs is also indexed by Native API name and, consequently, BlueTracer also utilizes the aforementioned Native API name resolution mechanism.

The idea is to create an array, named `syscallIDs`, where Native API names are indexed by their IDs, by parsing `ntdll`'s export information. The export data of a PE is stored in the `IMAGE_EXPORT_DIRECTORY` structure located in the header. In our case, the most relevant fields of this structure are:

- The `AddressOfFunctions` array, which contains RVAs [1] pointing to the actual exported functions and is indexed by an export ordinal.
- The `AddressOfNames` array, which is an array of 32-bit RVAs pointing to symbol strings.
- The `AddressOfNameOrdinals` array, which is an array of 16-bit ordinals existing in parallel with `AddressOfNames`, i.e. they possess the same number of elements and there is a direct relation between equivalent indices [26].

[1] A RVA (Relative Virtual Address) is essentially an offset within the PE image in memory

With this in mind, `syscallIDs` is built by carrying out the following actions for every element `iName` in `AddressOfNames`:

1. The corresponding `AddressOfNameOrdinals` element is retrieved, i.e. the one with the same index. Let us call this value `iOrdinal`.

2. `iOrdinal` is used to index in `AddressOfFunctions`. This time the obtained value is the RVA pointing to the exported function.

3. The pointed function is expected to begin `mov eax, syscall_number`. As a result of this, the first byte should be `B8h` and the syscall number can be obtained by considering the next four bytes.

By following the above procedure, every element in `AddressOfNames` can be therefore mapped to the corresponding identifier.

3.3.3 Native API Information Representation and Retrieval

When tracing Native APIs, it is wanted to record as much data as possible as well as correctly formatting the arguments' values. To do this, it is required to have access to some kind of source of Native API related information, which can assist the tracer in the logging activity by, for example, providing the number of arguments a Native API takes as input, listing the arguments' types and differentiating between input and output arguments. In BlueTracer, this information was adapted from the Native API data provided by Dr.Memory [12], a memory monitoring tool based on the DynamoRIO DBI framework.

For each Native API, the information related to it is contained in a `struct` of type `syscall_info_t` (*Listing 3.2*). Most fields of such `struct` are self-explanatory. In particular, `num` is a `struct` storing two values indicating the system call number; these are filled in dynamically by Dr.Memory but are not needed by BlueTracer. Furthermore, the `flags` field is utilized to notify whether or not all the details of the Native API are known, as most Native APIs are undocumented. Undoubtedly though, a major role is played by the `arg` array, which is made up by `sysinfo_arg_t` `structs` (*Listing 3.2*) containing the data of the Native API arguments. This array

is initialized with size `MAX_ARGS_IN_ENTRY` (i.e. 18), since a Native API can have at most 18 arguments.

```
1   typedef struct _syscall_info_t {
2
3       drsys_sysnum_t num;          // Native API ID
4       const char *name;            // Native API Name
5       uint flags;                  // SYSINFO_ flags
6       uint return_type;            // Return type
7       int arg_count;               // Number of arguments
8
9       // Array of arguments
10      sysinfo_arg_t arg[MAX_ARGS_IN_ENTRY];
11
12      ...
13  } syscall_info_t;
```

Listing 3.2. `struct` containing Native API-related information

```
1   typedef struct _sysinfo_arg_t {
2       int param;                   // Parameter Ordinal
3       int size;                    // Size
4       uint flags;                  // SYSARG_ flags
5       int type;                    // Type
6       const char *type_name;       // Symbolic Name of the arg Type
7   } sysinfo_arg_t;
```

Listing 3.3. `struct` containing information associated to a Native API argument

In `sysinfo_arg_t`, other than `param` and `size`, that are straightforward, the following fields are also present:

- `flags`, which stores an `OR` of flags describing the argument's characteristics. The most important ones are:
  - `SYSARG_READ (R)` : input argument.
  - `SYSARG_WRITE (W)` : output argument.

  - `SYSARG_INLINED` : non-memory argument, i.e the whole value is in parameter slot.
  - `SYSARG_HAS_TYPE (HT)` : argument with a type specifier. In fact, a non `SYSARG_INLINED` argument is by default of type `struct` (`DRSYS_TYPE_STRUCT`), unless specified otherwise by `SYSARG_HAS_TYPE`.

- `type`, which is an `enum` value indicating the data type of the parameter
- `type_name`, a string indicating the symbolic name of the `arg` type, which is typically filled dynamically based on the `type` value. In case the argument value is a named constant, the `type_name` field contains the name of the enumeration the constant belongs to. In fact, for each one of these enumerations, BlueTracer has access to a `struct` specific for that enumeration containing the constant values and their corresponding name. This information was also provided by Dr.Memory and is employed to translate constants to the appropriate name, as it will be explained in the next section.

To make things clearer, an instance of `syscall_info_t` for the Native API `NtAllocateVirtualMemory` was provided in the listing below (*Listing 3.4*).

```
1    { { 0,0 },"NtAllocateVirtualMemory", OK, RNTST, 6,
2    {
3        { 0, WIN_SIZE(HANDLE), SYSARG_INLINED, DRSYS_TYPE_HANDLE },
4        { 1, WIN_SIZE(PVOID), R | WR | HT, DRSYS_TYPE_POINTER },
5        { 2, WIN_SIZE(ULONG), SYSARG_INLINED, DRSYS_TYPE_UNSIGNED_INT },
6        { 3, WIN_SIZE(ULONG), R | WR | HT, DRSYS_TYPE_UNSIGNED_INT },
7        { 4, WIN_SIZE(ULONG), SYSARG_INLINED, DRSYS_TYPE_UNSIGNED_INT, "MEM_COMMIT" },
8        { 5, WIN_SIZE(ULONG), SYSARG_INLINED, DRSYS_TYPE_UNSIGNED_INT, "PAGE_NOACCESS" },
9    }
```

Listing 3.4. Instance of `syscall_info_t` relative to `NtAllocateVirtualMemory`

In *3.2.2* it was explained how BlueTracer obtains a Native API name from its ID. After that, as shown in *Figure 3.2*, there must be a way to, given a specific Native API name, retrieve the corresponding `syscall_info_t struct`. This is done by building an hash map during the tool's initialization which maps the Native API

name to the appropriate `struct`. Such idea is also adopted for named constants enumerations since a hash map is also constructed with the objective of associating the name of a named constant enumeration to the corresponding `struct`.

3.3.4 Native API Logging

With reference to *Figure 3.2*, let us consider the point in `TraceSysCallEntry` where the `syscall_info_t struct` relative to the Native API being traced was successfully retrieved. Before describing how the logging of "known" Native APIs is performed, it is worth mentioning that, in case a `struct` describing a certain Native API is not found, BlueTracer just logs the name of the Native API and the input value of a user-specified number of arguments (4 by default).

The first step towards logging "known" Native APIs is allocating and initializing a `syscall_tracer struct` (*Listing 3.5*) representing the specific Native API being traced. This `struct` is built starting from the data present in the previously retrieved `syscall_info_t struct` and it is at the heart of the logging of the Native API.

```
1   typedef struct _syscall_tracer {
2       ADDRINT syscall_number;         // Native API ID
3       const char * syscall_name;      // Native API Name
4       int argcount;                   // Number of Arguments
5
6       // Array of arguments
7       drsys_arg_t arguments[MAX_ARGS_IN_ENTRY];
8
9       drsys_arg_t retval;             // Return value
10      int syscall_counter;            // Native API Counter
11  } syscall_tracer;
```

Listing 3.5. `struct` representing the Native API being traced

Even for `syscall_tracer`, each field is self-explanatory. The only one which was not encountered before is the `syscall_counter` field. The rationale behind this field is that, in the log, each traced call has a unique integer associated to it, utilized to group together the information relative to the call in post-processing.

The next identifier to be assigned is stored in the `call_number` field of the thread local storage (*Listing 3.1*). Therefore, `syscall_counter` simply contains the unique identifier associated to the Native API in the log. The arguments of a specific Native API are represented via the `drsys_arg_t` struct (*Listing 3.6*).

```
1  typedef struct _drsys_arg_t {
2
3      // Whether operating pre-call (if true) or post-system call (if false)
4      bool pre;
5
6      int ordinal;                // Parameter Ordinal
7      int size;                   // Size
8      uint flags;                 // SYSARG_ flags
9      int type;                   // Type
10     const char *type_name;      // Symbolic Name of the arg Type
11
12     // String describing the symbolic name of a named constant's enum
13     const char *enum_name;
14
15     // Argument value
16     ADDRINT value;
17
18 } drsys_arg_t
```

Listing 3.6. `struct` representing the an argument of the Native API being traced

`drsys_arg_t`'s fields are mostly the same of `sysinfo_arg_t`, with three additions:

- `pre`, which is a boolean flag set to `true` when the logging is occurring before the Native API is executed (i.e. in `TraceSysCallEntry` and set to false when the logging is occurring after the Native API is executed (i.e. in `TraceSysCallExit`).
- `enum_name`, a string containing the symbolic name of the enumeration a named constant belongs to. In most cases, this is the symbolic name associated to the first constant of the enumeration. Such field was introduced to decouple

type name and enumeration name.

- `value`, an `ADDRINT`[2] containing the value of the argument. This is obtained through the `PIN_GetSyscallArgument` API, which takes as input the context before the execution of the system call, the system call standard and the ordinal number of the argument whose value is requested.

As previously mentioned in section 3.1.1, each thread possesses its own log file, whose pointer resides in the thread local storage, and writes on it by means of the `file_write` function, also located in the thread local storage (*Listing 3.1*). The first Native API component being logged is its name and this is done with the following format:

`~~[System ID of thread]~~ [Call Counter] [Image Name]![Native API Name]`

In the context of Native APIs, the name of the image is always `ntdll`, but this is not the case when tracing APIs.

Argument logging is done via the `print_arg` function (*Listing 3.7*), which is structured as follows:

- Firstly, it is determined if the argument being logged is a named constant, i.e. it is checked if `enum_name` is different from `NULL`. In that case, the `get_arg_syname` function is invoked, which is in charge of, given the input named constant value, finding the corresponding name and recording it to the log. The named constant resolution is carried out by initially fetching the `struct` containing all the members of the enumeration the named constant belongs to, through the use of the aforementioned hash map. Then, it is determined if there is a perfect match between the argument value and one of the values of the `struct`'s entries. If this occurs, then the associated symbolic name is simply retrieved from the `struct`'s entry. Otherwise, a linear search is performed to unveil possible composite named constants (e.g. `FILE_SHARED_READ | FILE_SHARED_WRITE`).

[2]The ADDRINT type is defined by Pin and represents a memory address

```
1  static void print_arg(drsys_arg_t* curr_arg, bluepill_tls* tdata, uint syscall_counter) {
2
3      ...
4
5      // Constant Resolution
6      if (curr_arg->enum_name != NULL) {
7          if (get_arg_symname(curr_arg, tdata))
8              return;
9      }
10
11     switch (curr_arg->type) {
12         case DRSYS_TYPE_VOID:         print_simple_value(curr_arg, true, tdata); break;
13         case DRSYS_TYPE_POINTER:      print_simple_value(curr_arg, true, tdata); break;
14         case DRSYS_TYPE_BOOL:         print_simple_value(curr_arg, false, tdata); break;
15         case DRSYS_TYPE_INT:          print_simple_value(curr_arg, false, tdata); break;
16         case DRSYS_TYPE_SIGNED_INT:   print_simple_value(curr_arg, false, tdata); break;
17         case DRSYS_TYPE_UNSIGNED_INT: print_simple_value(curr_arg, false, tdata); break;
18         case DRSYS_TYPE_HANDLE:       print_simple_value(curr_arg, false, tdata); break;
19         case DRSYS_TYPE_NTSTATUS:     print_simple_value(curr_arg, false, tdata); break;
20         case DRSYS_TYPE_ATOM:         print_simple_value(curr_arg, false, tdata); break;
21     default: {
22         if (curr_arg->value == 0) {
23             (tdata->file_write)(tdata->threadid, tdata->buffer, tdata->OutFile, "<null>");
24         }
25         else if (curr_arg->pre && !TEST(SYSARG_READ, curr_arg->flags)) {
26             (tdata->file_write)(tdata->threadid, tdata->buffer,
27             tdata->OutFile, PFX, curr_arg->value);
28         }
29         else {
30             if (!print_known_compound_type(curr_arg, tdata))
31                 (tdata->file_write)(tdata->threadid, tdata->buffer, tdata->OutFile, "<NYI>");
32         }
33     }
34     ...
35 }
```

Listing 3.7. `print_arg` function

- Then, if the argument's type is primitive (e.g. `int`, `bool`, etc.), `print_simple_value` is invoked to record its value. In particular the second parameter of this function is a boolean which is set to `true` if it is wanted to print the argument's value with leading zeros (e.g. if the argument is of pointer type and the address it stores is being logged) and `false` otherwise. The way `print_simple_value` operates is quite straightforward:

  1. At the beginning the argument's value is simply logged, with leading zeroes or not depending on the previously mentioned parameter
  2. Then, if the argument is a pointer, it is determined if the value it points to needs to be logged. This occurs if the argument is being processed before the Native API execution (i.e. `pre` is `true`) and is an input parameter (i.e. SYSARG_READ is contained in `flags`) or if the argument is being processed after the Native API execution (i.e. `pre` is `false`) and is an output parameter (i.e. SYSARG_WRITE is contained in `flags`). The pointed value is obtained through the use of `PIN_SafeCopy`, which copies the specified number of bytes from a source memory region to a destination memory region, guaranteeing safe return to the caller even if such regions are inaccessible.

- Finally, if the argument's type is not a primitive one, it is checked if it is `null` and if it is a complex output parameter being traced before the Native API execution, situation in which only the address stored in the pointer is logged. If none of these two scenarios are true, then the tracer finds itself in the situation where it is required to log the value of a complex type. This is performed through the invocation of `print_known_compound`. As of right now, such function differentiates between four complex types, namely `UNICODE_STRING`, `OBJECT_ATTRIBUTES`, `IO_STATUS_BLOCK` and `LARGE_INTEGER`, and logs the argument's value accordingly. If the argument's type is a complex type that is not supported, the symbolic value `<NYI>` (not yet implemented) is recorded.

Once the arguments' input values are logged, then the reference to the `syscall_tracer struct` representing the Native API under analysis is stored in thread local data (*List-*

ing 3.1). This is done so that, after the Native API's execution, the corresponding `syscall_tracer` data structure can be accessed from `TraceSysCallExit`. In fact, as mentioned in section 3.2, the `bluepill_tls struct` associated to the running thread can be retrieved in any analysis function by means of `PIN_GetThreadData`.

In `TraceSysCallExit`, after the value of `syscallEntry` has been fetched from the thread local storage, then it is determined whether the Native API has succeeded by checking if the output of `PIN_GetSyscallErrno` is equal to zero, where `PIN_GetSyscallErrno` is a Pin API returning the error code of the system call which has just returned with the provided context. Afterwards, `PIN_GetSyscallReturn` is utilized to get the return value of the Native API and, with this information, the corresponding `drsys_arg_t` data structure is built. Successively, the Native API return value is logged, together with the output arguments, using once again `print_arg` (*Listing 3.7*). Finally, the `syscall_tracer struct` representing the Native API being traced is deallocated.

To conclude this section, let us show how a traced Native API looks like in the log file (*Figure 3.4*). As it can be observed, the adopted log format is quite intuitive. In fact, the idea was to have a log which is both easy to read by the analyst but also easy to parse and post-process.

```
~~2868~~ 1072 ntdll.dll!NtOpenFile
1072    arg 0: 0x000fe810 (type=HANDLE*, size=0x4)
1072    arg 1: 0x100020 (type=unsigned int, size=0x4)
1072    arg 2: len=0x18, root=0x0, name=210/538 "\??\C:\Windows\WinSxS\
        x86_microsoft.windows.gdiplus_6595b64144ccf1df_1.1.7601.23894_none_5c0be957a009922e",
        att=0x40, sd=0x00000000, sqos=0x00000000 (type=OBJECT_ATTRIBUTES*, size=0x4)
1072    arg 3: 0x000fe7c0 (type=IO_STATUS_BLOCK*, size=0x4)
1072    arg 4: FILE_SHARE_READ|FILE_SHARE_WRITE (type=named constant, value=0x3, size=0x4)
1072    arg 5: FILE_DIRECTORY_FILE|FILE_SYNCHRONOUS_IO_NONALERT
        (type=named constant, value=0x21, size=0x4)
1072    succeeded =>
1072    arg 0: 0x000fe810 => 0x58 (type=HANDLE*, size=0x4)
1072    arg 3: status=0x0, info=0x1 (type=IO_STATUS_BLOCK*, size=0x4)
```

Listing 3.8. Log entry relative to a `NtOpenFile` call

3.4 APIs Tracing

In Microsoft Windows, APIs are implemented as functions provided by a set of dynamic-link libraries. These, also known as DLLs, constitute Microsoft's way of implementing the concept of shared libraries in the Windows operating system. Therefore, the functions they make available can be employed by different applications [15]. Windows APIs offer a plethora of functionalities which belong to many different categories, ranging from base services (e.g. file management) to user interface functions and network operations [8]. In light of this, a Windows API has rich semantic information associated to it and tracing the sequence of APIs a malware calls provides analysts with a very high-level view of the sample's behavior. This is why BlueTracer's API tracer component is undoubtedly the most important one.

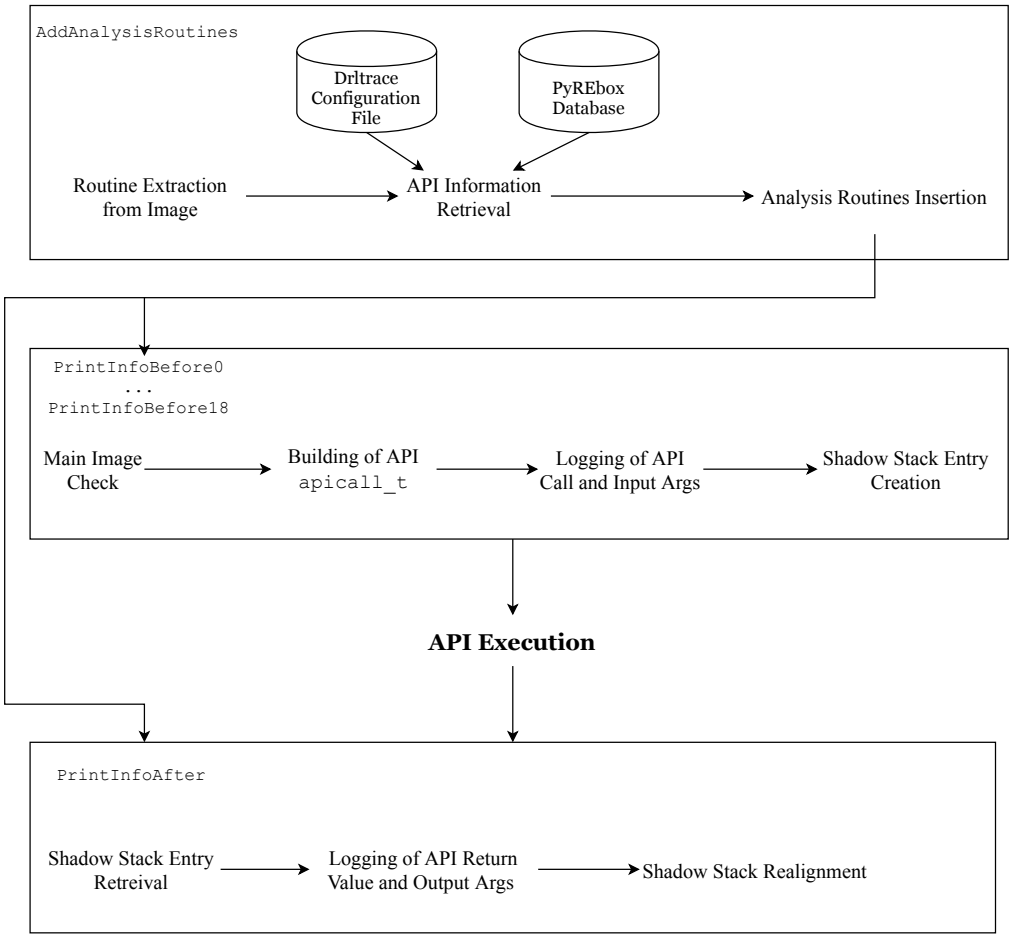

Figure 3.4. *APIs Tracing Workflow*

When tracing APIs, most of the concepts seen in Native API tracing also apply, but there are also significant differences, both in the steps that were carried out and in the Pin APIs which were employed, as it can be observed from the diagram in *Figure 3.4*.

Now let us illustrate the actions being performed during API tracing by analyzing in detail each one of the three main blocks of *Figure 3.4*.

3.4.1 APIs Instrumentation

The first step taken towards APIs tracing is APIs instrumentation, i.e. placing analysis routines before and after the APIs' execution. To do this, a specific function was embedded in Blue Pill's image callback (the notion of image callback was previously introduced in Section 3.3.1). Such function, named `AddAnalysisRoutines`, performs the following actions for every DLL being loaded:

1. It inspects the symbol table, i.e. the DLL's exports. To this end, Pin symbol objects (`SYM`s) are employed, which provide information about function symbols in the applications [23].

2. For every symbol object, it retrieves the corresponding routine (`RTN`) object. In fact, in Pin a `RTN` represents a function usually generated by a compiler for a procedural programming language, e.g C [23]. To retrieve the aforementioned `RTN` object, the `RTN_FindByAddress` API was utilized, which takes as input a `RTN` memory address and returns an handle to the found `RTN`. Such memory address was determined by adding the symbol's RVA (obtained via `SYM_Value`) to the lowest image address (fetched through `IMG_LowAddress`).

3. For every routine object, it retrieves the corresponding name via `RTN_name` and checks whether or not the API being represented is "known", i.e. a source of information associated to the API exists.

    - If the API is not "known", it adds an analysis routine just before the API's execution, which simply logs the value of a user-specified number of args (four by default), exactly like what happens in NTAPIs tracing.

- If the API is "known", it adds analysis routines before and after the API's execution, which log detailed information relative to the API, including the arguments' input and output values.

Having given a general outline on how `AddAnalysisRoutine` works, let us now describe its underlying details.

Similarly to what occurs for NTAPI tracing, in order to log as much information as possible and to correctly record the arguments' values, it is necessary to have a source of API information, providing some kind of assistance during the tracing activity. When dealing with APIs, BlueTracer allows the user to choose between two possible sources of API information: *drltrace*'s configuration file or the data extracted from *PyREbox*'s database. By default, the latter is employed as it embodies a greater amount of information, including variable names and named constants' enumerations.

drltrace (already mentioned in section 2.3) comes with its own external configuration file, in which each line contains information relative to an API, as it can be observed from the sample configuration file entry below (*Listing 3.9*).

```
bool|GetUILanguageInfo|DWORD|wchar*|__out wchar*|__inout DWORD*|__out DWORD*
```

Listing 3.9. Sample entry in *drltrace* configuration file

Each piece of information in an entry is separated by `|`. The first bit of data is the API's return type, the second is the API's name and the rest of the entry is made up by the arguments' types. In particular, the `__out` token is used to mark output arguments, while `__inout` is utilized to identify input and output arguments. In case there is no token, it means that the argument is just an input one.

Before API instrumentation takes place, BlueTracer parses the aforementioned configuration file and, for every entry, builds a `sysinfo_arg_t struct` (*Listing 3.2*) for each argument in the entry. It was decided to reuse `sysinfo_arg_t struct` since its fields were fitting with respect to the arguments' information which had to be stored. Therefore, after this parsing process, each configuration file entry is represented by an <API name, `vector` of `sysinfo_arg_t`s> pair. Such pairs are

then utilized to build an hash map, which maps the API name to the corresponding arguments. This hash map is used in step 3 of the APIs instrumentation process to, given the API name, fetch the appropriate information. In particular, it is important to note that, from a performance point of view, it is much better to include the information retrieval operations in the image instrumentation function, rather than in the analysis ones, as the former are executed just once for every image, while the latter are invoked each time an API is called by the pinned application.

The other source of API information employed by BlueTracer is *PyREbox*'s database. *PyREbox* is a Python scriptable reverse engineering sandbox based on QEMU and VMI techniques, the goal of which is to provide support for reverse engineering through its dynamic analysis capabilities [24]. *PyREbox* is equipped with its own API tracer, which relies on the information contained in a sqlite database to correctly log the APIs' parameters. Such database was, in turn, generated by utilizing as its core the data from the *Deviare*[3] project and then employing a crawler for parsing the Microsoft online documentation with the goal of marking which parameters are input parameters and which are output parameters. The resulting database is incredibly richer in API information than *drltrace*'s configuration file. In fact, the number of APIs for which data is stored is much higher, argument names are also included, as well as the contents of named constants' enumerations, information which *drltrace*'s configuration file does not provide at all.

In order to extract information relative to the APIs from PyREbox's database, a Python script was employed, which, for every API, instantiates a `libcall_info_t struct` (*Listing 3.10*) in a `.cpp` file.

```
1 typedef struct _libcall_info_t {
2     const char* func_name;              // API name
3     int argnum;                         // Number of Args
4     libcall_arg_info_t lib_args[19];    // Array of arguments
5 } libcall_info_t
```

Listing 3.10. `struct` containing API-related information from PyREbox's database

[3]Deviare is an open-source hooking engine for instrumenting arbitrary `Win32` functions [11]

The most important field of such `struct` is surely the arguments' array, which is made up of `libcall_arg_info_t` data structures (*Listing 3.11*) containing the information relative each argument of an API, including an entry specific for the return value data. The array size was set to 19, since it was observed from the database data that the API with the most parameters had 18 arguments. This time, it was decided not to reuse NTAPI's `sysinfo_arg_t struct` (*Listing 3.2*) to store the API's arguments information due to its lack of the field storing the argument name.

```
1  typedef struct _libcall_arg_info_t {
2
3      int arg_num;                // Arg Ordinal
4      const char* arg_name;       // Arg Name
5      int arg_type;               // Arg Type
6      char* arg_type_name;        // Arg Type Name
7      int size;                   // Size
8      bool in_out_flag;           // Input/Output Flag
9
10 } libcall_arg_info_t;
```

Listing 3.11. `struct` containing information associated to an API argument

Although `libcall_arg_info_t`'s fields seem to be pretty self-explanatory, let us quickly go through them to reveal some details which are not so obvious:

- `arg_num` is an `int` which contains the argument's ordinal. In case the return value is being represented, it is set to -1.
- `arg_name` is a string containing the argument name as reported by Microsoft's official documentation.
- `arg_type` is an `int` from an enumeration representing the argument's type. In particular, differently from the strategy adopted from Native APIs, in case of a pointer the `NKT_DBOBJCLASS_Pointer` enumeration member is ORed with an `int` representing the pointed type. The same thing happens in case of a pointer to a pointer but with the `NKT_DBOBJCLASS_PointerPointer` flag

being used instead. This was done to reflect how API information is stored inside the database.

- `arg_type_name` is a string which is used in the three specific scenarios below, according the type of the argument, and is `null` otherwise:

  1. To store the symbolic name of an enumeration.
  2. To store the `struct` type name.
  3. To store the `union` type name.

- `size` is an `int` storing the argument's size. In case of a pointer (or a pointer to a pointer) the size of the pointed type is stored.

- `in_out_flag` is a `bool` which is set to `INOUT` (i.e `true`) if the represented argument is an output argument and `IN` (i.e. `false`) otherwise. This means that, by default, all the arguments are considered as input arguments and are therefore traced before the API execution, while only the arguments for which `in_out_flag` is `true` are traced after the API's execution.

To give a better idea on how API data is represented, an instance of `libcall_arg_info_t` containing the information associated to `WriteFileEx` can be observed below (*Listing 3.12*).

```
1    { "WriteFileEx",5,
2    {
3        {-1, "Return value", NKT_DBFUNDTYPE_SignedDoubleWord, 0, 4, INOUT },
4        {0, "hFile", NKT_DBFUNDTYPE_UnsignedDoubleWord, 0, 4, IN },
5        {1, "lpBuffer", NKT_DBFUNDTYPE_Void | NKT_DBOBJCLASS_Pointer, 0, 0, IN },
6        {2, "nNumberOfBytesToWrite", NKT_DBFUNDTYPE_UnsignedDoubleWord, 0, 4, IN },
7        {3, "lpOverlapped", NKT_DBOBJCLASS_Struct | NKT_DBOBJCLASS_Pointer,
8         "_OVERLAPPED", 160, INOUT },
9        {4, "lpCompletionRoutine", NKT_DBOBJCLASS_Typedef, 0, 0, IN },
10   }
11   }
```

Listing 3.12. Instance of `libcall_arg_info_t` relative to `WriteFileEx`

Due to the high amount of APIs for which information was stored in the `PyREbox` database, it was decided to group API data by DLL. Specifically, for every DLL a corresponding `.cpp` file was created to store the API-related information under the form of `libcall_arg_info_t` instances, one for each "known" API exported by that DLL. Furthermore, in order to make API data retrieval more efficient, it was chosen to employ a two-level hash map approach for this purpose. Given the DLL name (easily obtainable in the image callback via the use of `IMG_Name`), the first level hash map is utilized to retrieve the DLL-specific second level hash map, which, in turn, maps the API name to the appropriate `struct` instance (*Figure 3.5*). This is how API information retrieval is carried out in case *PyREbox* database is used as API data source.

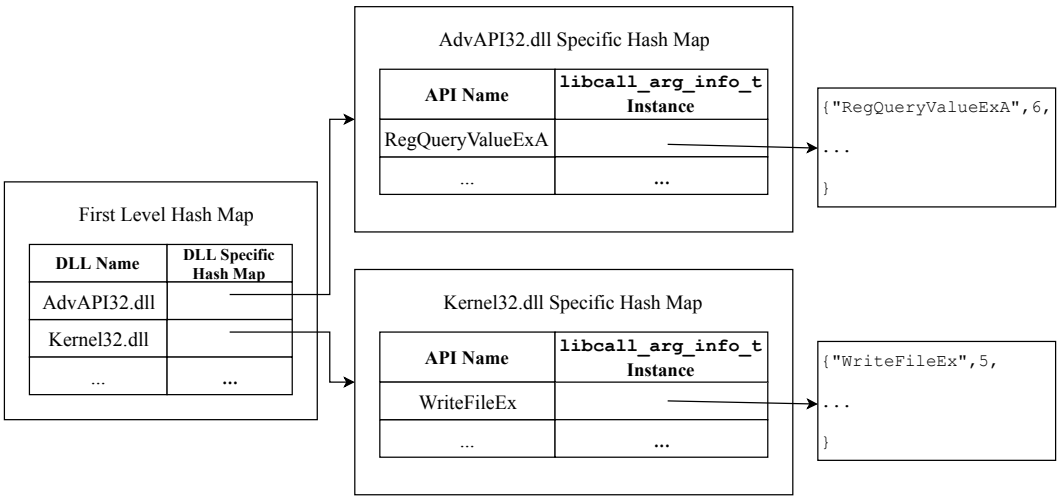

Figure 3.5. *PyREbox API Information Retrieval*

Let us conclude this section by explaining how analysis routines are inserted before and after the API execution during the API instrumentation procedure outlined in page 31. In BlueTracer such operation heavily relies on the use of the `RTN_InsertCall`, which allows to insert analysis functions relative to a `RTN` object. This API takes as input the following parameters:

- The routine object (`RTN`) representing the function under analysis.
- A member of the `IPOINT` enumeration which determines where the analysis

function is inserted in relation to the instrumented object. It can be either `IPOINT_BEFORE`, in which case the analysis code is placed before the instrumented function, or `IPOINT_AFTER`, in which case the analysis code is placed after the instrumented function.

- The analysis function to be called.
- The arguments to pass to the analysis function, specified via members of the `IARG_TYPE` enumeration. If the analysis function is placed before the instrumented object, then such arguments may also include the parameters' values of the function being instrumented. However, for this to occur the pair <`IARG_FUNCARG_ENTRYPOINT_VALUE`, argument number> needs to be included for each of the analyzed function's parameters. Conversely, to obtain the instrumented function's result, `IARG_FUNCRET_EXITPOINT_VALUE` is required. Obviously, this is possible only if the analysis function is placed after the instrumented object.

In BlueTracer, the number of arguments of the API being traced is known only after the API-related information has been retrieved. Because of this and given Pin's way of passing the arguments to the analysis functions, a `switch` had to be employed when adding analysis functions before the API execution. Such `switch`, given the API arguments' number, places the appropriate analysis function, i.e. the one with the number of <`IARG_FUNCARG_ENTRYPOINT_VALUE`, argument number> pairs equal to the number of arguments the API takes. This is done in order to ensure that the analysis function receives all the values of the API's arguments before the API's execution. As a result, in BlueTracer there are 19 (one for zero arguments as well) different API `IPOINT_BEFORE` analysis functions, named `PrintInfoBefore[Arguments' Number]`, which vary only in the number of APIs' argument values they receive in input. Obviously this is true just for "known" APIs. In case the API under analysis is not "known", then the number of API argument values is always the same and the `switch` is not required.

Let us now dissect an invocation of `RTN_InsertCall` to further clarify how BlueTracer adds analysis code before the execution of an API (*Listing 3.13*).

```
1 RTN_InsertCall(rtn, IPOINT_BEFORE, (AFUNPTR)PrintInfoBefore[Number of API Args],
2     IARG_FAST_ANALYSIS_CALL,
3     IARG_ADDRINT, argNumber,
4     IARG_ADDRINT, libPointer, IARG_ADDRINT, rtn_name, IARG_ADDRINT, img_name,
5     IARG_REG_VALUE, REG_STACK_PTR,
6     IARG_THREAD_ID,
7     IARG_RETURN_IP,
8     [Number of API Args]_ARGS
9     IARG_END);
```

Listing 3.13. `RTN_InsertCall` invocation to add analysis code before an API's execution

In particular, the arguments which are passed to the analysis routine are:

- `IARG_FAST_ANALYSIS_CALL`. This is not a real argument but rather a way to enable faster linkage for calls to analysis functions with the objective of improving performance [23].
- `argNumber`, a constant value containing the number of the analyzed API's arguments.
- `libPointer`, a reference to the previously retrieved data structure containing API-related data.
- `rtn_name`, the API name.
- `img_name`, the DLL name.
- The stack pointer value before the API's execution, given by the `<IARG_REG_VALUE, REG_STACK_PTR>` pair. This is necessary for the shadow stack management, which will be detailed in the next section.
- The thread ID assigned by Pin for the calling thread, passed via `IARG_THREAD_ID`. This is necessary in order to access the thread local storage from the analysis function.
- The API return address, provided by `IARG_RETURN_IP`. This is employed in the main image check, similarly to what explained in Section 3.3.1.

- The API argument values. In order to make the code more readable a macro was used in this case.

For what concerns the addition of analysis code after the API's execution, the aforementioned `switch` is not necessary since `IARG_FUNCRET_EXITPOINT_VALUE` is valid only at the entry point of an analysis routine and, therefore, no API argument value is passed at this stage. The analysis function which is placed after the API's execution, named `PrintInfoAfter` essentially receives the same parameters as the ones from *Listing 3.13*, with the major difference that the API's return value is passed via `IARG_FUNCARG_EXITPOINT_VALUE` instead of the API arguments' values.

3.4.2 API Analysis before Execution

In this section we will describe how APIs are analyzed before execution, i.e. the main steps that are carried out in the `PrintInfoBefore` functions, an overview of which is given in *Figure 3.4*. The workflow in this case is mostly the same as the one for NTAPIs' analysis, with one major difference, that is the introduction of a shadow stack used to store API entries in the thread local storage, which will be motivated later in the section.

The first thing which is done is determining whether or not the API under analysis has been invoked directly from the main executable of the Pin application. The concepts involved here are essentially the ones applying to the main image check done for Native APIs, which was detailed in *Section 3.3.1*. In short, it is checked if the API's return address, which is passed to the analysis function through the use of `IARG_RETURN_IP`, falls into the main executable memory address range, which was previously determined via `IMG_HighAddress` and `IMG_LowAddress`.

Afterwards, similarly to what occurs when logging "known" Native APIs, a `apicall_t struct` (*Listing 3.14*) representing the specific API being traced is allocated and initialized. The fields of such `struct` are all trivial and, as what done for Native API tracing, the API arguments are represented via a `drsys_arg_t` (*Listing 3.6*) array with size `MAX_ARGS_CONFIG` (i.e. 18) as this is the maximum number of observed arguments for any "known" API. In particular, the `drsys_arg_t struct` could be reused in the context of APIs since, unlike `sysinfo_arg_t`, its fields could

hold all API arguments' data, both from the *drltrace* configuration file and the *PyREBox* database, including the argument name.

```
typedef struct _apicall_t {;

    const char* img_name;                          // DLL Name
    const char* rtn_name;                          // API Name

    drsys_arg_t arguments[MAX_ARGS_CONFIG];        // Array of Arguments
    drsys_arg_t retval;                            // Return Value
    int count;                                     // Arg Count

} apicall_t;
```

Listing 3.14. `struct` representing the API being traced

The `apicall_t struct` is initialized utilizing the API-related data, which was fetched beforehand during instrumentation and was passed to the analysis function via `libPointer`, as well as the values of the API arguments, which were provided to the analysis function through `IARG_FUNCARG_ENTRYPOINT_VALUE` as already discussed in *Section 3.4.1*.

Then, the logging of the API proceeds exactly the same as for Native APIs: the API call is recorded employing an identical log format in order to maintain uniformity and the input values of the arguments are logged through the use of a `print_arg` function operating exactly in the same way as the one in *Listing 3.7*.

As previously hinted at, the major element which distinguishes API tracing from Native API tracing is that the former employs a shadow stack to store in the thread local storage references to the data structures representing the APIs being traced. Let us explain in detail the motivation behind such choice. It is typical for an API to call another API during its execution, forming a sort of nesting. For this reason, just storing in the thread locale storage the reference relative to the next API to be executed is not enough as this would lead to the analysis of just the most "inner" API, completely ignoring the "outer" APIs which invoked the "inner" one. Therefore, it is necessary to store not just the reference corresponding to the next API in line

to be executed but rather all the references relative to the APIs for which execution has started and not yet finished. A fitting data structure to solve this issue is a shadow stack, where, for each API, the corresponding data structure is placed on top of the stack before its execution and popped from the stack after its execution has terminated, allowing both "inner" and "outer" APIs to be traced. This is why each thread has access to its own shadow stack via the `shadowStack` field in the thread local data (*Listing 3.1*).

Therefore, in light of what just stated, the last operations which are done when tracing an API before its execution are allocating a shadow stack entry (*Listing 3.15*), initializing it and placing it on top of the shadow stack in the thread local data, so that it can be retrieved after the API's execution.

```
1 typedef struct _stackEntry {
2     apicall_t* apiInfo;    // API Call Reference
3     ADDRINT  currentSP;    // Stack Pointer Register Content
4     THREADID threadID;     // Thread ID
5     uint api_counter;      // Call Counter
6 } stackEntry;
```

Listing 3.15. `struct` representing a shadow stack entry

`stackEntry`'s fields are all, once again, self-explanatory. A big role in the shadow stack management is taken by the `currentSP` field, as it will be explained in the next section. Such field stores the stack pointer's value before the API's execution, which was passed to the analysis function with the <`IARG_REG_VALUE`, `REG_STACK_PTR`> pair.

3.4.3 API Analysis after Execution

Lastly, let us detail the most important operations performed towards tracing API-related information after the API execution. These are carried out in the `PrintInfoAfter` function and can be observed from *Figure 3.4*.

At the beginning of `PrintInfoAfter`, the first thing which needs to be done in order to proceed with the logging is the retrieval from the shadow stack of the stack

entry representing the API under analysis. A reference to the shadow stack can be easily acquired by gaining access to the thread local storage via `PIN_GetThreadData`. However, it is not enough to pop the top of the shadow stack to obtain the wanted stack entry. This is due to a limitation of Pin in the instrumentation of routine objects.

In fact, the addition of analysis routines after a function's execution with `IPOINT_AFTER` is implemented by instrumenting each return instruction. This does not guarantee the success of the instrumentation, meaning that, for some routines, the associated `IPOINT_AFTER` analysis function is not executed. Therefore, in our case, it is possible that the stack entry on top of the shadow stack refers to API for which `PrintInfoAfter` has not been invoked and, therefore, the corresponding stack entry has not been popped.

As a result of this, the correct stack entry has to be fetched by iterating through the shadow stack, looking for the entry for which the values of a set of fields coincide with the values of the corresponding parameters being passed as arguments to the analysis function. Such fields are:

- The DLL name.
- The API name.
- The stack pointer value. In fact, for a given API, the stack pointer value before its execution has to be equal to the stack pointer value after its execution.

Once the appropriate shadow stack entry has been retrieved, the API return value, which is passed to the analysis function via `IARG_FUNCARG_EXITPOINT_VALUE`, is set in the `apicall_t struct` so that it can be logged, together with the output arguments, once again utilizing the ideas behind `print_arg` (*Listing 3.7*).

The last operation carried out by `PrintInfoAfter`, after the API under analysis has been traced, is the re-alignment of the shadow stack, i.e. the deallocation of all the stack entries belonging to APIs which have terminated their execution. In fact, due to the previously mentioned issue involving Pin's instrumentation of routines, it is necessary to manually remove from the shadow stack all the entries relative to the

APIs for which the corresponding `IPOINT_AFTER` analysis function is not executed. Such shadow stack re-alignment procedure works as follows:

1. The previously fetched shadow stack entry is removed.
2. All the shadow stack entries on top of the previously retrieved entry are removed. Due to their position in the stack, these entries are relative to APIs which have been called after the traced API. Such APIs, given the fact that the traced API has terminated its execution, have also finished to execute, but their relative entries have not been deallocated accordingly due to Pin's issue with routine instrumentation.
3. All the entries with a lower stack pointer value than the stack pointer value stored in the retrieved entry are removed. In fact, such entries refer, once again, to APIs which have been invoked before the traced APIs but their corresponding entries have not been deallocated accordingly.

3.5 Context Change Analysis

Pin has to intercept the delivery of asynchronous events from the kernel in order to keep control of the application. Microsoft Windows employs two mechanisms aimed at dispatching asynchronous events to user mode, i.e. callbacks and asynchronous procedure calls (APCs), both of which lead to a context change in the application [27].

In order to have a complete picture of the malware sample's activity, it is clearly relevant to also log the occurrences of such asynchronous events and to record as much information as possible related to them. Luckily, Pin provides an API which perfectly meets these needs, named `PIN_AddContextChangeFunction`.

This API allows to register a notification function to execute just before the pinned application changes context due to the reception of an asynchronous event, like for instance a callback or a Windows APC. In BlueTracer, the idea is to use such registered notification function to extract and log information relative to the context change. The notification function in question is of type `CONTEXT_CHANGE_CALLBACK` and the most important arguments it takes as input are:

- A member of the `CONTEXT_CHANGE_REASON` enumeration indicating the reason for the context change. This is the parameter allowing BlueTracer to distinguish between callbacks and APCs. In fact, in the case of callbacks it is set to `CONTEXT_CHANGE_REASON_CALLBACK`, while in the case of APCs it is set to `CONTEXT_CHANGE_REASON_APC`.

- The application's register state before the context change. It is `null` if the context change is caused by a callback.

- The application's register state after the context change.

Having outlined the tools made available by Pin to deal with the application's context changes, let us now explain in detail how callbacks tracing and Windows APCs tracing is carried out in BlueTracer.

3.5.1 Callbacks Tracing

The Windows kernel frequently needs to make callbacks in user mode with the goal of carrying out specific tasks, such as the invocation of hooks defined by the application, the provision of event notifications and the exchange of data with user mode. These calls are commonly referred to as user mode callbacks [20].

To make calls from user mode to kernel mode, the user mode callback dispatcher (`KiUserCallbackDispatcher`) is employed. This function takes two arguments, a number indicating the callback to be invoked and a structure pointer which the callback receives as input, containing several parameters packed in a contiguous memory block [2]. `KiUserCallbackDispatcher` operates in the following way:

1. It accesses the `KernelCallbackTable` in the process environment block (PEB) [4]. The `KernelCallbackTable` is an array of function pointers, where each entry contains a reference to a callback routine [7].

2. It uses the callback number to locate the right routine in the array and invokes it, providing it with the aforementioned input data structure.

[4]The process environment block (PEB) is a data structure employed in Windows operating systems to represent a user mode process [7]

In case of a user mode callback, in Pin the registered `CONTEXT_CALLBACK_NOTIFICATION` function is invoked right before `KiUserCallbackDispatcher`'s execution, obviously with the reason paramater set to `CONTEXT_CHANGE_REASON_CALLBACK`.

In light of what just stated, it is natural for BlueTracer's logging activity, taking place in the registered `CONTEXT_CALLBACK_NOTIFICATION` function, to involve the record of a callback occurrence, together with the name of the callback that is about to be invoked. Unfortunately, tracing the callback's arguments would require further work since they are packed in a data structure, as previously explained.

Right before the execution of `KiUserCallbackDispatcher`, i.e. when the registered `CONTEXT_CALLBACK_NOTIFICATION` function is called, the stack looks as follows (*Figure 3.6*).

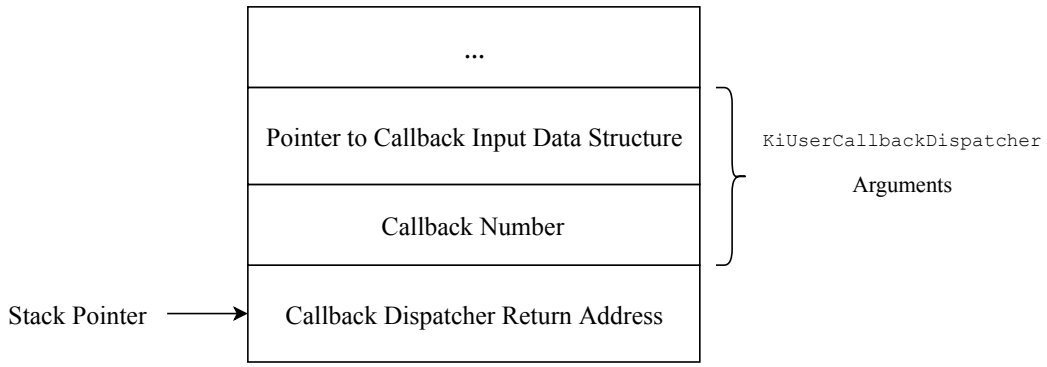

Figure 3.6. *Stack before `KiUserCallbackDispatcher` execution*

Having this in mind, it is now possible to explain step-by-step how the tracing of callbacks takes place in BlueTracer:

1. The stack pointer value is retrieved. This is achieved through the use of the `PIN_GetContextReg` API, that allows to get the value of a register in the context provided as input. Such context, in our specific case, is the application's register state after the context change, which, as previously mentioned, is passed as argument to the `CONTEXT_CHANGE_CALLBACK` notification function.

2. The callback number is obtained by adding 4 to the stack pointer and fetching the pointed value, in accordance to *Figure 3.6*.

3. The `KernelCallbackTable` is retrieved from the PEB.

4. The function pointer stored in the `KernelCallbackTable` entry indicated by the callback number is acquired.

5. It is checked if the address of the function falls within `User32.dll` memory address range. In fact, in `Win32` user mode callbacks are employed exclusively by `User32.dll` for windowing related aspects [2], meaning that the retrieved function address must belong to `User32.dll`'s memory address range. Such range is determined by retrieving an handle for `User32.dll` through `GetModuleHandle`, building an `IMG` object from it utilizing Pin's `IMG_FindByAddress` and invoking the usual `IMG_LowAddress` and `IMG_HighAddress` functions.

6. The name of the callback is determined by creating a `RTN` object from the function address via `RTN_FindByAddress` and consequently invoking `RTN_Name` on such object.

7. The callback occurrence and the callback name is recorded in the log.

The result of the aforementioned callback tracing procedure is a log entry having the following format (*Listing 3.16*).

```
Context change caused by callback -> USER32.dll!LoadMenuW
```

Listing 3.16. Log entry relative to a user mode callback

3.5.2 Windows Asynchronous Procedure Calls Tracing

Windows Asynchronous Procedure Calls (APCs) are employed to alter a thread's ordinary execution path and reroute it to execute some other code. An important concept related to APCs is that, every time an APC is scheduled, it is intended for a specific thread.

APCs are utilized in many situations. Some examples are:

- The I\O Manager employs an APC to terminate an I\O operation inside the thread which started it.

- There exists a particular APC which is used to forcibly enter in a process' execution when it has to terminate.
- Some APIs, like `ReadFileEx` and `WriteFileEx`, utilize APCs when performing asynchronous I\O operations.

APCs are of different types, one of which is user mode APCs. These usually call user mode code and are normally dispatched when the thread willingly enters into an alertable state. This means that, typically, user mode APCs do not asynchronously force themselves into the targeted thread, but they can be seen more like a set of work items in a queue, which the thread processes when he decides to do so [21].

Just like `KiUserCallbackDispatcher` for user mode callbacks, APCs are channelled by means of a single dispatcher function inside `ntdll`, named `KiUserApcDispatcher`. Such function receives as input a set of paramaters, including the address of the APC routine to be invoked and the references to two data structures in which the APC's arguments are stored [1].

If a user mode APC takes place, in Pin the registered `CONTEXT_CALLBACK_NOTIFICATION` function is called immediately before the execution of `KiUserApcDispatcher`, with the reason parameter, this time, set to `CONTEXT_CHANGE_REASON_APC`. Therefore, the course of action taken by BlueTracer to record user mode APCs is exactly the same as the ones for user mode callbacks, i.e. tracing the APC occurrence as well as the name of the APC that is about to be executed. Once again, however, BlueTracer does not yet support arguments' tracing in this scenario since, also in this case, the arguments are packed in data structures.

Let us now go through the steps employed by BlueTracer to trace user mode APCs:

1. The stack pointer value is retrieved, exactly in the same way as explained for user mode callbacks in section 3.5.1
2. The address of the APC to be invoked is obtained by adding 4 to the stack pointer and fetching the pointed value. In fact, in the stack, the APC address is located right on top of the APC's dispatcher return address, which is pointed by the stack pointer.

3. Given the address of the APC to be invoked, the corresponding `RTN` object is created via `RTN_FindByAddress`. Similarly, from the APC address, the `IMG` object representing the DLL exporting the APC is also built, this time through the use of `IMG_FindByAddress`.

4. The APC name and the DLL name are determined by invoking `RTN_Name` and `IMG_Name` respectively on the objects created in the previous step.

5. The APC-related data is recorded in the log. This includes the APC occurrence, the name of the DLL exporting the APC and the APC name.

The resulting log format is exactly the same as the one adopted for user mode callbacks.

Chapter 4

Experimental Results

Donec et nisl id sapien blandit mattis. Aenean dictum odio sit amet risus. Morbi purus. Nulla a est sit amet purus venenatis iaculis. Vivamus viverra purus vel magna. Donec in justo sed odio malesuada dapibus. Nunc ultrices aliquam nunc. Vivamus facilisis pellentesque velit. Nulla nunc velit, vulputate dapibus, vulputate id, mattis ac, justo. Nam mattis elit dapibus purus. Quisque enim risus, congue non, elementum ut, mattis quis, sem. Quisque elit.

Chapter 5

Conclusions

Donec et nisl id sapien blandit mattis. Aenean dictum odio sit amet risus. Morbi purus. Nulla a est sit amet purus venenatis iaculis. Vivamus viverra purus vel magna. Donec in justo sed odio malesuada dapibus. Nunc ultrices aliquam nunc. Vivamus facilisis pellentesque velit. Nulla nunc velit, vulputate dapibus, vulputate id, mattis ac, justo. Nam mattis elit dapibus purus. Quisque enim risus, congue non, elementum ut, mattis quis, sem. Quisque elit.

5.1 Future Directions

Phasellus id magna. Duis malesuada interdum arcu. Integer metus. Morbi pulvinar pellentesque mi. Suspendisse sed est eu magna molestie egestas. Quisque mi lorem, pulvinar eget, egestas quis, luctus at, ante. Proin auctor vehicula purus. Fusce ac nisl aliquam ante hendrerit pellentesque. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Morbi wisi. Etiam arcu mauris, facilisis sed, eleifend non, nonummy ut, pede. Cras ut lacus tempor metus mollis placerat. Vivamus eu tortor vel metus interdum malesuada.

Bibliography

[1] *A catalog of NTDLL kernel mode to user mode callbacks, part 3: KiUserApcDispatcher*. 2007. URL: http://www.nynaeve.net/?p=202.

[2] *A catalog of NTDLL kernel mode to user mode callbacks, part 5: KiUserCallbackDispatcher*. 2007. URL: http://www.nynaeve.net/?p=204.

[3] Ulrich Bayer and Christopher Krügel. "TTAnalyze : A Tool for Analyzing Malware". In: 2005.

[4] J. Berdajs and Z. Bosnić. "Extending Applications Using an Advanced Approach to DLL Injection and API Hooking". In: *Softw. Pract. Exper.* 40.7 (June 2010), pp. 567–584. ISSN: 0038-0644. DOI: 10.1002/spe.v40:7. URL: http://dx.doi.org/10.1002/spe.v40:7.

[5] Armin Buescher, Felix Leder, and Thomas Siebert. "Banksafe Information Stealer Detection Inside the Web Browser". In: *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*. RAID'11. Menlo Park, CA: Springer-Verlag, 2011, pp. 262–280. ISBN: 978-3-642-23643-3. DOI: 10.1007/978-3-642-23644-0_14. URL: http://dx.doi.org/10.1007/978-3-642-23644-0_14.

[6] Geoff Chappell. *Native API Functions*. 2017. URL: https://www.geoffchappell.com/studies/windows/win32/ntdll/api/native.htm.

[7] Geoff Chappell. *PEB*. 2016. URL: https://www.geoffchappell.com/studies/windows/win32/ntdll/structs/peb/index.htm.

[8] *Chapter 1: Introduction to Win32/Win64*. 2018. URL: https://technet.microsoft.com/en-us/library/bb496995.aspx.

[9] *Cisco 2018 Annual Cybersecurity Report*. 2018. URL: https://www.cisco.com/c/dam/m/hu_hu/campaigns/security-hub/pdf/acr-2018.pdf.

[10] Daniele Cono D'Elia, Emilio Coppa, and Camil Demetrescu. *The DBI Blue Pill: Practical Analysis of Evasive Malware*.

[11] *Deviare API Hook Overview*. 2012. URL: https://www.nektra.com/products/deviare-api-hook-windows/.

[12] *Dr.Memory*. URL: http://drmemory.org/.

[13] *Dynamic Binary Instrumentation*. 2007. URL: http://uninformed.org/index.cgi?v=7&a=1&p=3.

[14] *Dynamic Instrumentation Tool Platform*. URL: http://www.dynamorio.org/.

[15] *Dynamic-Link Libraries*. 2018. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/ms682589(v=vs.85).aspx.

[16] Manuel Egele et al. "A Survey on Automated Dynamic Malware-analysis Techniques and Tools". In: *ACM Comput. Surv.* 44.2 (Mar. 2008), 6:1–6:42. ISSN: 0360-0300. DOI: 10.1145/2089125.2089126.

[17] *Intercepting all System Calls by Hooking KiFastSystemCall*. 2015. URL: https://www.malwaretech.com/2015/04/intercepting-all-system-calls-by.html.

[18] *Internet Security Threat Report, vol. 23*. 2018. URL: https://www.symantec.com/content/dam/symantec/docs/reports/istr-23-2018-en.pdf.

[19] Ivo Ivanov. *API hooking revealed*. 2002. URL: https://www.codeproject.com/Articles/2082/API-hooking-revealed.

[20] Tarjei Mandt. *Kernel Attacks through User-Mode Callbacks*. URL: https://media.blackhat.com/bh-us-11/Mandt/BH_US_11_Mandt_win32k_WP.pdf.

[21] Enrico Martignetti. *Windows Vista APC Internals*. 2009. URL: http://www.opening-windows.com/download/apcinternals/2009-05/windows_vista_apc_internals.pdf.

[22] Syed Zainudeen Mohd Shaid and Mohd Maarof. "In memory detection of Windows API call hooking technique". In: (Aug. 2015), pp. 294–298.

[23] *Pin - A Dynamic Binary Instrumentation Tool*. 2012. URL: https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[24] *PyREbox*. 2018. URL: https://talosintelligence.com/pyrebox.

[25] Mark Russinovich. *Inside the Native API*. 1998. URL: http://persephone.cps.unizar.es/~spd/pub/windows/ntdll.htm.

[26] Michael Sikorski and Andrew Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. 1st. San Francisco, CA, USA: No Starch Press, 2012. ISBN: 1593272901, 9781593272906.

[27] A. Skaletsky et al. "Dynamic program analysis of Microsoft Windows applications". In: *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*. 2010, pp. 2–12. DOI: 10.1109/ISPASS.2010.5452079.

[28] Sherri Sparks, Shawn Embleton, and Cliff C. Zou. "WINDOWS ROOTKITS A GAME OF HIDE AND SEEK". In: *Handbook of Security and Networks*. 2011, pp. 345–368. DOI: 10.1142/9789814273046_0014. eprint: https://www.worldscientific.com/doi/pdf/10.1142/9789814273046_0014. URL: https://www.worldscientific.com/doi/abs/10.1142/9789814273046_0014.

[29] Dafydd Stuttard et al. *Attack and Defend Computer Security Set*. 1st. Wiley Publishing, 2014. ISBN: 111890673X, 9781118906736.