UNIVERSITÄT MANNHEIM

PyBox - A Python Sandbox

Diploma Thesis by **Christian Schönbein**

May 03, 2011

First Examiner:	Prof. Dr. Ing. Felix C. Freiling
Second Examiner:	Prof. Dr. Ing. Wolfgang Effelsberg
Advisor:	Jan Göbel, Markus Engelberth

Hiermit versichere ich, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Mannheim, den 03. Mai 2011 Christian Schönbein

Abstract

Considering the challenges resulting from the complexity and variety of malware, the application of dynamic malware analysis in order to automate the monitoring of malware behavior has become increasingly important. For this purpose, so-called *sandboxes* are used which provide the functionality to execute malware in a secure, controlled environment and observe its activities during runtime. The result of this analysis process is a comprehensible report that shows the behavior of the observed malware sample.

While a variety of sandbox software, such as the GFI Sandbox (formerly CWSandbox) or the Joe Sandbox, is available, all solutions are closed-source. Therefore, the task of this thesis is to describe the implementation of an open-source sandbox which uses the techniques *API hooking* as well as *DLL injection*. API hooking is applied to intercept and to log function calls to the APIs provided by the Windows operating system. For this purpose, customized code in the form of a dynamic link library (DLL) is injected into the malware sample's process.

Considering the growing number of attacks on mobile devices, such as smartphones running Google's Android operating system, the thesis also provides an outlook as to how the implemented solution can be ported to a Linux operating system.

Zusammenfassung

In Anbetracht der Herausforderungen, die sich durch die wachsende Komplexität und Vielfalt von Schadsoftware ergeben, werden Methoden der dynamischen Malwareanalyse eingesetzt, um automatisiert das Verhalten und die Auswirkungen einer Schadsoftware bestimmen zu können. In diesem Rahmen werden sogenannte "Sandboxen" eingesetzt. Hierbei wird die Schadsoftware in einer kontrollierten Umgebung ausgeführt und zur Laufzeit beobachtet, welche Aktionen diese ausführt. Das Resultat ist ein für Menschen verständlicher Bericht, der die Verhaltensweise der Schadsoftware aufzeigt.

Es existieren mit GFI Sandbox (ehemals CWSandbox) oder Joe Sandbox bereits Beispiele solcher Sandboxen. Jedoch sind all diese closed-source. Daher besteht die Aufgabe meiner Diplomarbeit darin, eine open-source Sandbox zu implementieren. Diese wird die Techniken *API Hooking* und *DLL Injection* verwenden. API Hooking wird verwendet, um die Aufrufe von APIs in Windows zu unterbrechen und zu dokumentieren. Hierzu wird eigener Programmcode in Form einer DLL mittels DLL Injection in dem Prozess der Schadsoftware eingeschleust.

Aufgrund der größer werdenden Bedrohung von mobilen Geräten, wie z.B. Smartphones, mit dem Betriebssystem Android von Google, durch Schadsoftware, soll im Rahmen der Diplomarbeit weiterhin ein Ausblick auf eine mögliche Portierung der PyBox nach Linux dargestellt werden.

Contents

Listings

List of Figures

List of Tables

Chapter 1.

Introduction

In the early 1990s, the Internet started its triumph with the development of the *World Wide Web* (WWW). This has caused tremendous worldwide changes and has influenced nearly every aspect of modern life. Based on the created infrastructure many new technologies such as Voice-Over-IP, Peer-to-Peer, Online Gaming, Social Networks and Smart Phones, have been developed. In particular, new means and ways to distribute data have emerged. However, despite the large number of possibilities and advantages that the new technologies offer, they have also caused another less desirable development.

Developers of malware have used the new communication channels to spread their viruses, worms, and Trojan horses causing major damages and data losses, resulting in financial losses every year. At first, these attacks were the attention-driven actions of individuals who wanted to show their skills, but with the growing economic significance of the Internet and its platforms, the motives of malware creation and utilization have changed. Soon thereafter, organized crime discovered the potential of this field.

As a consequence, the authors of malware became profit-oriented and professionally trained. According to Ollman [Oll08, p. 1], whole conglomerates have evolved offering highly complex malware products in a very competitive market. However, these attacks are not exclusively financially driven; also, political motives must not be underestimated. This became increasingly obvious last year during the events concerning Wikileaks when various companies such as MasterCard and Visa terminated their cooperation with the whistle-blowing website. Due to their association, they fell victim to so-called *Distributed Denial of Service* (DDoS) attacks by a collective of hackers who call themselves "Anonymous" [Vij10]. Another example are the attacks against the Egyptian government websites during the revolution in late January [Som11]. Thus, hackers do not only use their abilities for profit-related reasons, but also for protest. This form of protest is also referred to as *Hacktivism* (cf. Francie Coulter [CE10, p. 3]). In July 2010, according to Falliere et al. [FMC10, p. 2], the virus *Stuxnet* was discovered, which had affected some of Iran's nuclear program computers causing a huge media impact. Stuxnet can target industrial control systems, and therefore demontrates the high level of malware quality and complexity present these days. In particular, Sophos [Whi11, p. 11] refers to Stuxnet as "military-grade malware" indicating that countries also utilize malware techniques for military reasons. With these incidents in mind, politicians are increasingly concerned with topics like cyberwar and malware in general. This also recently became apparent, at the *Munich Security Conference* (MSC)[1] in February 2011, the world's

[1] http://www.securityconference.de

most important conference concerning security policy, when several politicians acknowledged the major importance of these issues. For example, according to the Frankfurter Allgemeine Zeitung [FAZ11], the German chancellor Angela Merkel mentioned that a cyberwar would be as much of a threat as a "regular" war.

In any case, the amount, complexity, and quality of malware are higher than ever. The McAffee Security Labs [BDG11, p. 2] reports increases in targeted attacks, in sophistication, and in the number of attacks on new devices. And this trend is not slowing down. Thus, for the foreseeable future, it will not get easier to keep one's IT environment clean from malware infections.

1.1. Motivation

The growing amount, variety and complexity of malware poses major challenges for today's IT security specialists. Key factors in this battle are understanding the functionality of malware as well as the ability to predict threats to the security of information systems. Therefore, it is necessary to analyze the different types of malware in order to protect critical systems. However, since manual methods, such as *disassembly* and *reverse engineering*, can be time consuming, automization of the process of analyzing malware behavior is key to this problem. For this purpose, the means of *Dynamic Malware Analysis* are utilized. In this context, so-called *sandboxes*, such as *GFI Sandbox*[2] and *Joebox*[3] are offered to execute malware samples in a controlled environment, observe their actions during runtime, and create machine-readable reports. However, to date all existing solutions are closed-source.

1.2. Task

The task of this thesis is to describe the creation of an open-source sandbox. This entails providing a secure environment in which malware samples can be safely executed by preventing the propagation of critical processes through network devices. During runtime, the processes of malware samples are monitored, logged, and documented. The result of this process is a generated machine-readable report displaying all events.

1.3. Results

In this thesis, we describe the development of the PyBox analysis environment. PyBox is short for *Python Sandbox*. It is developed and executed in a virtual machine running the operating system *Microsoft Windows XP*[4] in order to provide a secure environment. The Python-based application provides the functionality to run a malware process and monitor its behavior by controlling its usage of *Application Programming Interfaces* (API).

[2] http://www.sunbeltsoftware.com/Malware-Research-Analysis-Tools/Sunbelt-CWSandbox/
[3] http://www.joebox.ch/
[4] http://www.microsoft.com/windows/windows-xp/default.aspx

Therefore, the techniques of *dynamic malware analysis*, *API hooking* and *DLL injection* are combined. PyBox injects a *dynamic-link library* (DLL) file into the created process, which installs its functionality and redirects all API calls to its own code. Whenever a hooked API is called, the original execution flow is intercepted and the corresponding information is sent to PyBox. Subsequently, the execution flow can either be resumed or aborted depending on the provided settings. The latter are defined in a configuration file of PyBox. Finally, an XML-based report is created containing the information logged during runtime.

1.4. Outline

This thesis is structured as follows: In Chapter 2, we introduce the prerequisites required for the implementation of PyBox. After providing some Windows fundamentals, we discuss the concepts of hooking and injection methods. In Chapter 3, we describe the implementation of PyBox. More specifically, we present the three components, which are used to monitor the malware behavior: the PyBox virtual machine, the hook library and the analysis tool. In Chapter 4, we provide an outlook as to how the PyBox functionality could be extended to work on Linux operating systems. Finally, Chapter 5 concludes this thesis by summarizing its contents and presenting major findings.

1.5. Acknowledgement

First of all, I want to express my deepest gratitude to Prof. Dr. Felix C. Freiling for giving me the opportunity to work on such an interesting topic for my thesis. Furthermore, I wish to thank Prof. Dr. Wolfgang Effelsberg for being my second examiner. I also want to thank my advisors Jan Göbel and Markus Engelberth for their incessant support and very valuable feedback. I also would like to thank my cousin, my friends, and my cohabitants for their support, the interesting discussions and the great time we had. Last but not least, I want to thank my family for giving me constant backup and support throughout my thesis and studies.

Chapter 2.

Prerequisites

Throughout this thesis, we use analysis techniques to study the behavior of malware. Like any other application, malware needs an infrastructure in which it can be executed. Furthermore, every program is implemented to interact with its environment receiving input and sending output. Malware also uses and manipulates its environment exploiting provided system functionality for its malicious purposes. Therefore, these two topics, malware and operating system, are closely related to each other and it is essential to be familiar with both the characteristics of malware and the infrastructure in which the malware is run. As a result of this information, the appropriate analysis techniques can be derived.

As malware is the center of attention in this thesis, this chapter starts with an introduction to basic information about malware. Afterwards, fundamental concepts concerning the *Windows NT*-based operating systems are described providing a basic understanding of how they use and provide their resources as well as how applications are run. Finally, appropriate techniques that are used by PyBox to analyze malware samples are introduced.

In this chapter, we focus on the Windows operating system because it is still the target system for most existing malware. However, with the increasing usage of mobile devices, there are more and more occurrences of attacks on their predominant operating systems such as the Linux-based Android. Therefore, we take a look at possible ways to port the functionality of PyBox to Linux in Chapter 4 as well.

2.1. Malware

The term *malware* is an abbreviation for *malicious software*. It includes all kinds of software that perform or add unwanted functionality to a system or program without the user's consent. The purpose as well as the caused damage varies from malware sample to malware sample. While some programs do not cause much damage but instead aim for annoyance by for example permanently launching pop-ups, others can be very harmful. For example, there are various malware samples which spy on secret data such as passwords or delete files thus potentially causing a much greater extent of damage.

Although malware is just one aspect concerning IT Security that implies all means to protect a system from external influences, ranging from unauthorized access to natural

Malware Type	Self-replication	Population Growth	Parasitism
Computer Virus	yes	positive	yes
Computer Worm	yes	positive	no
Trojan Horse	no	zero	yes
Spyware	no	zero	no
Adware	no	zero	no
Back Door	no	zero	possibly
Logic Bomb	no	zero	possibly

Table 2.1.: Malware characteristics according to Aycock [Ayc06]

disasters, it is nonetheless an increasingly important topic with new variants emerging each day.

Since we develop PyBox with the intention to analyze these threats, it is helpful to know about the characteristics of malware. Therefore, in what follows we briefly introduce different malware types. Subsequently, we present possible mechanisms to detect malware.

2.1.1. Malware Types

It is becoming increasingly difficult to divide malware into certain categories or malware types since nowadays most malware has a modular structure combining the functionality of various types. Furthermore, there are also samples which include an update functionality even offering the possibility to extent their capabilities. As an example, a Trojan horse can use backdoor functionality to load malicious code, spyware functionality to log a user's behavior and can connect to a bot network in order to be remotely controlled.

According to Aycock [Ayc06], malware can be characterized using three attributes: self-replication, population growth, parasitism. In Table 2.1 we provide an overview of some basic malware types regarding these attributes. In the following, we explain these and other basic malware types in more detail.

Computer Virus

A biological virus is only able to replicate itself inside living cells of an organism. It cannot exist without a host. *Computer viruses* are named after biological viruses because they show an analogous behavioral pattern. They are parasitic programs which infect computers by attaching themselves to a host program and changing its behavior via modification of its code. More specifically, it injects itself into the host object, infects its code which, if executed, replicates itself again and infects further objects. Potential host objects include files or boot sectors. Thus, the key-defining characteristic of computer viruses is that they are self-replicating.

In order to fulfill its purpose, a virus usually consists of three parts: the *infection mechanism*, a *trigger*, and the *payload*. The infection mechanism describes the exact

means used by the virus to modify the target's code in order to infect it. The trigger represents the event which has to occur in order to cause the execution of the malicious code residing in the payload. As the primary characteristic of computer viruses is self-replication, the trigger and the payload are optional components.

A computer virus usually spreads to other objects that are located on a single computer. It does not propagate through computer networks which instead is the main characteristic of a computer worm. However, they often propagate to other computers, too. This is usually the case if human-transportable media are used to transfer data from one computer to another one.

Computer Worm

A *computer worm* is quite similar to a computer virus. In particular, its key-defining characteristic also is self-replication. However, there are also some differences: A computer worm is not parasitic, it does not require a host program and therefore does not modify other code in order to replicate itself. Instead, it is a standalone program that exists independently and replicates itself. It uses computer networks for propagation. Thus, it spreads from one computer to another by infecting them through network devices.

Trojan Horse

The term *Trojan horse* is based on Greek mythology. More precisely, this type of malware is named after an incident of the Trojan War narrated in Homer's epos Ilias in which the Greek defeat the Trojans by a ploy. Having besieged the Trojan city for a long time without the prospect of victory, the Greek construct a huge wooden horse in which their men are hidden. The Trojans – thinking the horse is a present of the gods – transport the horse into their city. In the following night, the Greek are finally able to open the gates and defeat the Trojans.

Akin to this tale, Trojan horses in computing are programs which incorporate secret and unwanted functionality. After installation, the program executes everything as desired while at the same time secret tasks are performed without the consent of the user. A classic example is the situation in which the user executes a program which requires a username as well as a password as user input. The Trojan horse secretly captures the user input and obtains the login information. In contrast to computer viruses and computer worms, they do not replicate themselves, but only pursue their own tasks without infecting other programs.

Spyware

Spyware is the catchall term for software designed to spy on a user. Its purpose is to observe the user's activities and gather information. The obtained information can be of various types. For instance, it may contain user input, screen shots, or the content of certain directories in the file system. Finally, the spyware either stores the information

somewhere or transmits it to a third party using a network connection. All of this happens without the consent or knowledge of the user.

Adware

Adware can be seen as a special case of Spyware: Both are used to capture information about a user and his or her activities. In general, adware is a piece of software which is used for advertising purposes. It downloads special advertisement features and displays them on the local computer. Usually, the adware tries to match the advertisement to the context of the user's activities. For example, if a user is searching for a certain product the advertisement may display similar goods which might appeal to the user. Adware is promoted as free software which offers a special purpose, but secretly changes the behavior of software installed on the computer.

Salomon [Sal10, p. 248] mentions the software *PurityScan* as an example for adware. PurityScan is a free software that claims to provide the functionality to scan one's computer for content which is not desired by the user. However, at the same time, it provides undesired content by placing advertisement in pop-ups, window bars, text links, and even manipulates search results for advertising purposes.

Back Door

Back doors refer to code that is used to bypass security mechanisms. They can either infiltrate into other code or run as independent programs. Back doors are often used in order to avoid authentication processes. For example, *Remote Administration Tools* (RTA) that allow users to access their computers remotely. If a malware succeeds in installing a RTA access, it has successfully created a back door.

Logic Bombs

A *logic bomb* is code that can either be parasitic, i.e. that it can be placed in existing, legitimate code, or it can also be found as standalone software. The code usually consists of two parts: a *trigger* and the *payload*. These elements are analogous to those of the computer virus. The trigger is a condition. It decides whether or not to activate the payload. The payload is the actual code which is executed in the event of the trigger being activated. There are no restrictions regarding the payload's code in order to be classified as a logic bomb. During its non-active presence, a logic bomb is placed such that it is difficult to detect.

Rootkits

In contrast to the malware types presented so far, *rootkits* do not entail one piece of software only. Instead, they consist of a number of useful tools, and the entirety of these tools is referred to as a *kit*. The tools can be either solely code or entire programs enabling access to a computer with the maximal access rights of a system adminstrator.

This type of access is also called *root* access. This is why this collection of tools is called a rootkit.

The goal of a rootkit is its enduring and non-detectable presence in the system. For this reason, its main functionality is to hide its presence by preventing all of its components from being detected. In addition, rootkits can provide other functionalities. Common examples of these enhanced tools are functions that enable remote access to the system as well as spying functionality such as the sniffing of network traffic.

In spite of the potential threat rootkits pose, they are not necessarily a technology which is exclusively applied for malicious reasons. There are also several legitimate fields of application such as the application for spying purposes by the police for public interest.

Bots

Bot is an abbreviation for the word *robot*. The name is derived from its functionality. A bot is a program which is able to pursue certain tasks autonomously. Bots are not necessarily applied for malicious reasons. An example of a legitimate field for the application of bots is the usage of so-called *web crawlers* such as the *Googlebot* [Goob]. Web crawlers are sent in order to scan websites for keywords. Thus, they enable their search engines to list the scanned websites appropriately when a corresponding keyword has been entered as a search term. However, bots also offer much potential to be employed for malicious purposes. A bot can for example incorporate functions such as sending emails, spying data, communicating with other bots, or even infecting other computers and turning them into bots. They can also be remotely controlled. Thus, large crowds of bots can be controlled by a single person. This collective of bots is called a *botnet* which is often used for DDoS attacks or sending spam.

2.1.2. Detection

In the previous section, we have described various malware types. They usually exploit certain software vulnerabilities to execute their payload and infect information systems and networks. In recent years, the number of cyberattacks that make use of more or less complex manifestations of these types has grown rapidly. In order to address this problem, we first must be able to detect such attacks. One possibility to detect them that has proved effective is the utilisation of *Intrusion Detection Systems* (IDS). In the following, we take a closer look at the characteristics of such systems.

Intrusion Detection Systems

An IDS is no complete defense system preventing all attacks but rather represents an element of the process chain to achieve this. However, it plays an important role when it comes to identifying and defending attacks on an information system or network. An IDS determines all possible attack manifestations in order to detect all potential attacks. At the same time, it tries to avoid false alarms.

An IDS can be applied for various reasons. Sometimes, it is used to obtain forensic information. This information can for example enable the police to locate and prosecute attackers attempting to intrude into a certain system. Another reason for their usage is their usage in combination with certain defence mechanisms that aim to prevent or destroy the malicious activity. In such a scenario, the IDS triggers a certain mechanism based on the activity which has been monitored. Furthermore, an IDS can be used to monitor all activities to reveal insights on the vulnerability of a system. Thus, the new information provided by the IDS allows administrators to fix these vulnerabilities and make the system or network more robust to attacks.

According to McHugh et al. [MCA00, p. 44], an IDS generally consists of computers which are set up as sensors collecting information of system or network resources. The monitored information is subsequently sent to a network-specific analysis station. In case an intrusion is detected, a warning signal is triggered. As for the analysis, there are different techniques to decide whether the current activity is an attack or not. On the one hand, *signature-based detectors* can be used which match the current activity to known malicious behavior. On the other hand, *anomaly-based detectors* can be used which observe the current activity and check if it conforms to the specified rules. Based on these techniques, an IDS can monitor user and system activity, protect the integrity of critical system and data files, match activity patterns to known attacks, and check for abnormal activities on a system (cf. Rozenblum [Roz03]).

IDSs are usually categorized according to their fields of application:

- **Network Intrusion Detection System (NIDS)**
  A NIDS is used to monitor a certain network subnet. It scans all traffic and matches all activities to known attack patterns. If a suspicious behavior is detected, an alert is triggered.
- **Network Node Intrusion Detection System (NNIDS)**
  In contrast to a NIDS, a NNIDS does not monitor a whole subnet, but the traffic between a network and a single specific host which is connected to the network.
- **Host Intrusion Detection System (HIDS)**
  A HIDS only monitors activity on a single host in order to detect intrusive activity. It checks critical system and data files by observing various system snapshots or logs. Furthermore, it can audit the interaction between applications and the operating system in order to look for suspicious behavior.

The various IDS types are often combined to increase resistance to attacks.

2.2. Windows Fundamentals

The ability to reverse engineer and to reveal the behavior of a program such as a malware sample requires knowledge of the respective operating system's characteristics. This is due to the fundamental functionalities of software: Programs are designed to interact with the system, whether they write to or read from files, request user input, or display something on the screen. Since thus far PyBox is implemented in Windows, this sec-

tion provides some general information about the 32-Bit Windows NT-based operating systems. In particular, we detail how processes are created and managed, and how they interact with the operating system.

2.2.1. Memory Management

One of the most important features of modern operating systems is the concept of *virtual memory*. Programs do not have direct access to physical memory, but instead each running process has its own virtual memory encapsulating physical memory access. The considered Windows operating systems in this thesis use 32-bit memory addresses and, therefore, can at most address 2^{32} bytes (= 4 gigabytes). This design guarantees that all programs are isolated from each other. Only the operating system is allowed to access physical memory. Thus, the system is responsible for loading and unloading memory blocks of a fixed size called *pages*. This process is referred to as *paging*. There is an *address space* for each running program, which is basically a *page table* telling the program's process which physical memory to use. Thus, the address space realizes virtual memory by granting access only to those parts of memory that belong to the program.

2.2.2. Kernel Mode vs. User Mode

Another important concept is the distinction between the CPU modes *kernel mode* and *user mode*. All Windows versions since Windows NT use these access modes in order to ensure that user applications do not overwrite system data. Additionally, each process's address space is divided into two separate parts of two Gigabytes: kernel memory and user memory. Thus, the operating system allows access to kernel memory only if the processor is in kernel mode, which presents the privileged mode. While in user mode – the non-privileged mode – only user-mode code can be run and only user memory can be accessed whereas the use of any other code or data is prohibited. Consequently, it is a security concept guaranteeing that in case of misuses or mistakes the system stability will not be affected.

2.2.3. Application Programming Interfaces

Application Programming Interfaces (API) are a set of functions provided by a software. Applications may use these functions in order to interact with the API provider obtaining data structures or executing required functions. Operating systems, control programs such as *database management systems* (DBMS), or communication protocols make APIs available in order to provide applications with necessary functionality. In this thesis, we focus on the APIs which are provided by the operating systems of the Windows NT family, particularly the *Windows API*[1] and the *native API*.

[1] http://msdn.microsoft.com/en-us/library/cc433218(VS.85).aspx

The Windows API

As mentioned before, processes are limited by their address space and cannot directly communicate with another. Furthermore, the application code is in user-mode and therefore cannot access system resources. In both cases system functionality is required. The Windows API, which was formerly called *Win32 API*, serves this purpose. It is a set of API functions included in *dynamic link libraries* (DLL) which can be used by Windows-based applications. It represents the interface for low-level programming providing services such as access to system resources like memory and devices, displaying graphics and formatted texts, and the integration of audio, video as well as networking services. Furthermore, usage of the Windows API ensures compatibility with all Windows versions.

According to Eilam [Eil05], the core Windows API can be divided into three categories: *GDI*, *user*, and *kernel*. GDI APIs (in `gdi32.dll`) are the system's graphic interface and provide low-level graphics services which offer various objects such as pens and brushes for drawing lines or displaying bitmaps. GDI APIs are also used by the *graphical user interface*-related (GUI) services and objects which can be accessed through the user APIs (in `user32.dll`). Examples of User API objects are controls, menus, and dialog boxes. While these two categories contain services for displaying content and user interaction, the kernel APIs (in `kernel32.dll`) make kernel-related services such as memory, object, process and thread management as well as file input and file output available to user applications. Yet, the kernel APIs do not process direct system calls into the Windows kernel. Instead, they make use of yet another set of APIs: the native APIs. We take a closer look at the native APIs in the next section.

Since we are dealing with IT security matters throughout this thesis, we focus on the kernel category. Further APIs which are in the focus of interest are those providing network services through sockets (in `wsock32.dll`, `ws2_32.dll`) as well as the APIs for advanced system management (in `advapi.dll`) which for example enable programs to query and edit the values of the *windows registry*.

Most often, application developers make little to no use of the Windows API. This is because it is considered as rather inconvenient to use as one usually needs many function calls and the initialization of data structures to perform single operations. Other interfaces like the *Microsoft Foundation Classes* (MFC) or the *.NET Framework* simplify the usage of those operations. But in the end, these are just wrappers around the API functions. Thus, the applications still use the Windows APIs although they do not access them directly.

The native API

The native API is the most basic and direct yet incompletely-documented interface into the kernel existing a layer below the Windows API (cf. Eilam [Eil05, p. 90]). Whenever an application calls an API exported by the `kernel32.dll` in order to access a specific system resource, the kernel API calls a native API which in turn validates the passed parameters and then uses a system call to switch to kernel mode and access the desired item or functionality. The reason for these double interface calls is to provide com-

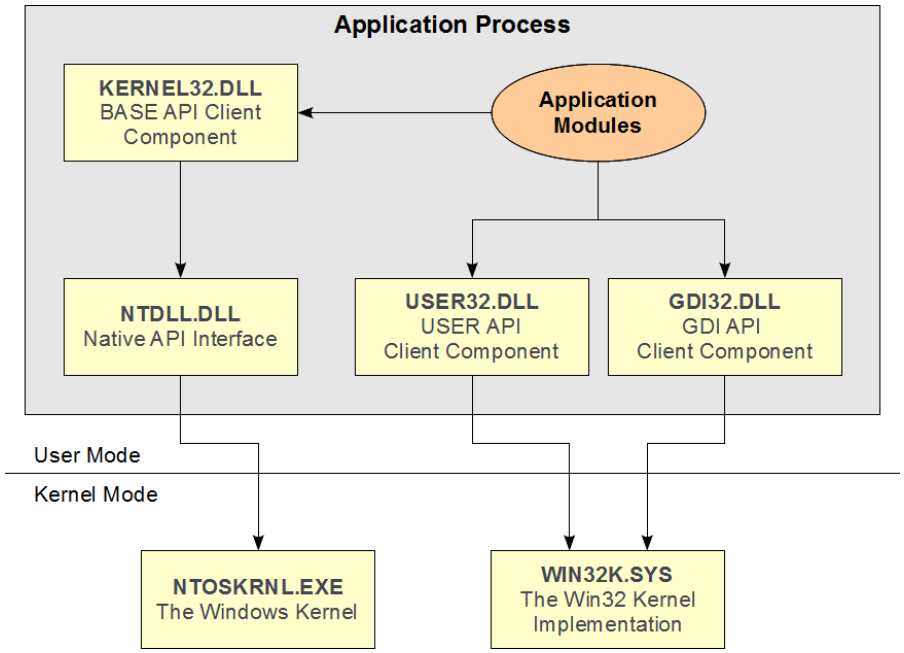

Figure 2.1.: The Windows interface DLLs and their relation to the kernel components according to Eilam [Eil05, p. 89]

patibility with the older Windows Versions before NT. Therefore, Microsoft encourages developers to use the Windows API in order to ensure compatibility.

According to Eilam [Eil05], the native API consists of two files: `ntdll.dll` and `ntoskrnl.exe`. While the `ntdll.dll` provides kernel functions to user-mode applications, services exported by `ntoskrnl.exe` can only be accessed from kernel mode. A further characteristic of the native APIs is that for each exported API there are two versions with two different name prefixes: **"Nt"** and **"Zw"**, as in `NtCreateProcess` and `ZwCreateProcess`. Functions with the prefix "Nt" are designed for user-mode callers, whereas the APIs beginning with "Zw" are designed for kernel-mode calls. If a user-mode caller uses a "Zw" function, the call will output the same result as with the "Nt" function. This is, because in user-mode both versions point to the same address. This is a security mechanism ensuring that no user-mode code can execute a call requiring kernel-level privileges.

It should be noted that neither the user APIs nor the GDI APIs make any use of the native API, because it only provides kernel-related and, thus, no graphics-related functionality. Figure 2.1 depicts the relation between the different interface DLL files and the associated kernel components.

While most applications use the Windows API, there are some exceptions: Some Windows services such as the *chkdsk*[2] application use native API services directly [Rus06]. However, there is also the possibility of malware using the native API in order to hide from anti-malware programs.

[2]http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/chkdsk.mspx

2.2.4. The Windows Executable Format

The file format of executable files on Windows operating systems such as `.exe` files or `.dll` files is called *Portable Executable* (PE). It is derived from the *Common Object File Format* (COFF). PE files are called "portable" because they can be executed on different architectures. It is intended that they can be run on all Windows versions and are independent of the CPU type used. Another characteristic is that they are relocatable, which means that they can be loaded at different addresses in virtual memory. There is a reason for that. Whenever a main PE file (which is usually an .exe file) is executed, it has to load further executable files such as several Windows APIs including the `kernel32.dll`. Since a lot of executable files can be loaded into a single address space, it is required that each has its own space in memory. Therefore, their memory addresses have to be dynamically assigned. If not, it would be possible that two or more executables are located at the same virtual addresses. Whenever an executable is loaded into memory, the Windows loader does not map the whole file but only those ranges which are needed. It reads the PE file and takes solely the corresponding parts. The mapped object in memory is then referred to as a *module*.

The structure of a PE file is outlined in Figure 2.2. Basically, the file consists of a couple of *headers* and various *sections*.

While the headers instruct the operating system how to handle the file, the sections contain the actual content of the executable's program. But these contents which have to be mapped into memory can be of different types. There are for example some which contain the executable's code which has to be executed and there are others which store its data. Furthermore, access rights for each section have to be specified which determine whether the content is read-only, writable, or executable. The most important section types are *code sections* and *text sections* containing the program code and *data sections* including essential data. Thus, based on a section's type the operating system can use the corresponding content accordingly and prevent misuses.

As mentioned above, apart from the executable's sections a PE file contains different headers within its structure. These headers are implemented as simple data structures which are defined in the `WINNT.h` file of *Microsoft's Windows Software Development Kit*[3].

The *MS-DOS header* (Structure: `IMAGE_DOS_HEADER`) serves the purpose of backward-compatibility with MS-DOS systems. Yet, in spite of this feature no PE file will actually run on a DOS system. Instead, the system will display the following error message: "This file cannot be run in DOS mode". Therefore, each Windows executable contains a small MS-DOS stub code wich outputs this message in case it is executed on a DOS system. Since there are probably very few people using MS-DOS in the meantime, there are only two fields in the MS-DOS header which are actually important: The values of the fields `e_magic` and `e_lfanew`. The value of `e_magic` has to be set to `0x5A4D` which is the ASCII representation for the letters "MZ" being the initials of Mark Zbikowski, a former Microsoft developer and architect of the MS-DOS file format. The `e_lfanew`

[3] http://www.microsoft.com/downloads/en/details.aspx?FamilyID=6b6c21d2-2006-4afa-9702-529fa782d63b

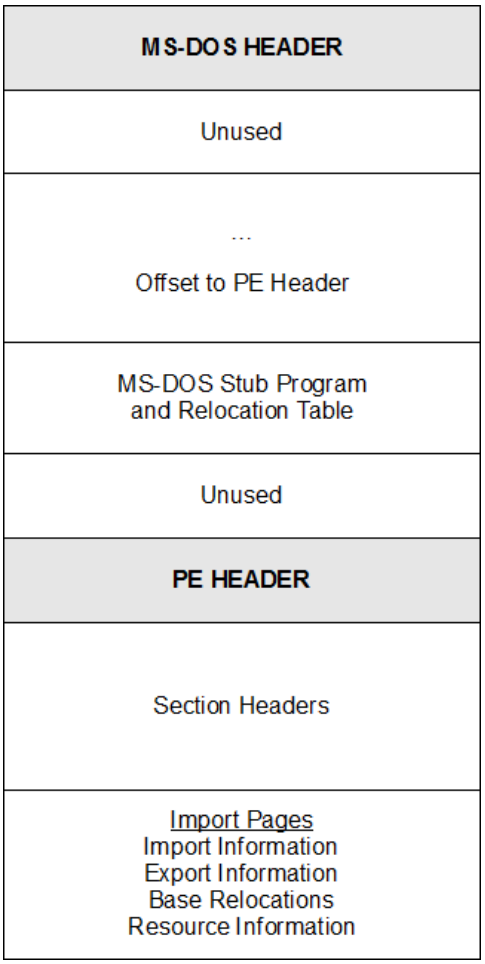

Figure 2.2.: Portable Executable (PE) file format according to Microsoft [Mic10]

field contains an offset pointing to the PE header.

The *PE header* (Structure: `IMAGE_NT_HEADER`) is the executable's actual header containing information which is relevant for the operating system. It consists of three fields: *PE signature*, *file header*, and *optional header*. The PE signature is usually set to `0x00004550` which is "PE00" in ASCII. The second and third member are data structures describing the file's contents and its basic information. Also, in spite of its name, the optional header is not at all optional to an executable because it provides information used in order to load the file. Among other things, the optional header contains an array of `IMAGE_DATA_DIRECTORY` structures which provide information regarding the address and size of important sections including certain `.idata` sections containing information about the module's *import table* and *export table*. These tables are used for *dynamic linking*. As described earlier, Windows applications do not have to consist of one single executable file. DLL files such as the various APIs mentioned above can be imported and used within the program. Thus, using the functionality of dynamic linking provides modularity and results in a reduction of program memory consumption. Yet, as the used functions of those modules are not imported until the application is executed no information concerning the memory addresses of the imported functions can be stored in the

PE file. Therefore, each module has an import table listing the imported modules and used functions and an export table which stores the names and *relative virtual addresses* (RVA) of its exported functions. In case a module is loaded, all functions of the import table are mapped and their addresses are identified and stored in the *import address table* (IAT). The memory addresses can be calculated by adding the current function's RVA from the export table to the module's base address.

Finally, the *section table* is an array of data structures immediately following the PE header. The structures are of the type `IMAGE_SECTION_HEADER`. There is a data structure for each section in the PE file describing the associated section's characteristics such as its location and size. The number of sections and thus the number of `IMAGE_SECTION_HEADER`s in the array is defined in the field `NumberOfSections` in the file header of the PE header.

For more information about PE files, please refer to the *Microsoft Portable Executable and Common Object File Format Specification* [Mic10] or the `WINNT.h`. Further insights into the PE files are provided by Pietrek [Pie02a] [Pie02b].

2.3. Techniques of Analyzing Executables

In the previous section, we discussed the different processes of a system and the way they are managed, the usage of APIs, and how executables are structured and treated by the Windows operating system. This information is crucial in order to analyze and understand the behavior of executables. This section outlines the techniques used by PyBox in order to reveal and monitor an executable's actions and describes possible ways to implement them. As the usage of sandboxes is a means of dynamic malware analysis, the term "dynamic malware analysis" is first defined and two different concepts of realization are depicted. Furthermore, the techniques of API hooking and DLL injection which in combination represent a way of analyzing executables are explained including possible implementation alternatives.

2.3.1. Malware Analysis

The process of malware analysis is an important step in order to design efficient detection techniques. When analyzing malware in order to determine its functionality and purpose, there is usually no source code available and if it is there is no guarantee that the compiled version is really based on this source. Various solutions for this problem exist. According to Bayer et al. [BMKK06, p. 68-70] and Willems et al. [WHF07, p. 33], they are classified into two categories: *static malware analysis* and *dynamic malware analysis*.

Static Malware Analysis

When using static malware analysis the malicious code is not executed. Instead, it must be disassembled and thus converted into assembler instructions. Then, analysis

techniques can be used to reveal the control and data flow. Advantages of this solution are that it is easier to implement and that it inspects the whole content of a malware sample. However, there are also some drawbacks. For instance, attackers can use *code obfuscation* and *byte obfuscation* techniques in order to impede the analysis process. The idea behind obfuscation is to change the code or execution flow in a way that makes it very hard to analyze while keeping the intended functionality. Furthermore, there are also types of malware for which a static malware analysis would be rather useless because they change dynamically and therefore will not execute the code which has been analyzed. Examples of these types are polymorphic and metamorphic worms which are basically self-modifying programs that change their appearance during runtime.

Dynamic Malware Analysis

Another way to learn about the actual behavior of a malware sample or an application in general is to actually run the software and observe the changes it causes to the system. This type of analysis is called *dynamic malware analysis*.

Of course, it is usually not advisable to execute the malware on a native system. Depending on the malicious code, it would be cumbersome to undo the changes to the system every time a malware sample has been analyzed. Furthermore, the computer might be connected to the internet or a local network, which, say in case of computer worms, could lead to the outbreak and infection of further computers. As a solution, various means exist to simulate the target system. A popular choice is to use *virtual machines* such as *VMware*[4] or *VirtualBox*[5]. The advantage is that a virtual machine provides a secure environment preventing malware escapes as well as the ability to use *snapshots* of the virtual machine allowing an analyst to restore the system to a infection-free state after the analysis process. Another possibility of simulating an operating system are *emulators* such as *Wine*[6]. While both methods can simulate the target system, they differ in the usage of computer resources. Within virtualization, virtual machines can directly run instructions on the CPU and access devices. In contrast to this, emulators simulate every system and hardware interaction through the emulation software.

By applying the functionality of either virtualization or emulation, we are able to run, observe, and analyze malware samples in a secure environment. There are two different concepts of dynamic malware analysis that can be applied for this purpose. The first concept entails running the malware to be analyzed and examine the system snapshots before and after the run. The advantage of this method is that it is quite relatively easy to implement. However, the analysis only reveals the results of the malware sample's execution, but not the activity during the execution. If the malware has undone its changes and left no marks, the actions will remain unnoticed. The second concept is analyzing the executable's activity during runtime, which also reveals those types of activities. Again, there exist two ways of analyzing executables during runtime. One way of system-wide scope is to implement windows kernel drivers that intercept system

[4]http://www.vmware.com
[5]http://www.virtualbox.org
[6]http://www.winehq.org

calls. This way, all process, registry and filesystem activity can be monitored. The methods using this concept are hidden from malware. This presents a major advantage as some malware samples check for software that might inspect them. However, these methods are only able to monitor all system calls. They cannot observe calls within the malicious code. For this purpose, another approach is used that is of executable-level scope meaning that only the activity of the corresponding malware sample is observed but not the entire system activity. More precisely, certain functions of the executable such as used APIs are intercepted and redirected to infiltrated monitoring functions. The interception and redirection of functions is described in more detail in the next section. The disadvantage of this method is that in certain situations, the malware might detect the changes caused in memory or the executable file. To avoid such situations, solutions using this approach require mechanisms that prevent their detection. This can be done by hiding the added functionality and careful implementation.

Since we focus on single malware samples in this thesis, we apply dynamic malware analysis with a VirtualBox virtual machine simulating the target system, and monitor the sample's activities during its runtime by using the latter concept described above.

2.3.2. Hooking

Hooking is a concept used to gain control of a program's execution flow without changing and recompiling its source code. This is achieved by intercepting function calls and redirecting them to infiltrated customized code. By the provision of customized code, any operation can be executed. Afterwards, the function's original functionality can be executed and the result can be either simply returned or changed and returned in order to transfer control back to the code which has called the hooked function. Therefore, hooking provides a perfectly suitable means that can be used for dynamic malware analysis.

Fields of Application

Hooking enables us to add customized functionality to an application. Hence, it is a useful technique which can be applied under various circumstances. Its main purposes are threefold:

- **Extending functionality:**
  Very often applications are considered nice and useful but lack a few features that would make them more complete. In such situations, one might consider developing one's own application based on the desired needs assuming although this often is time-consuming and cumbersome. Instead, we can use hooks in order to append the desired functionality to the existing application.
- **Debugging and reverse engineering:**
  Hooking techniques are often used for debugging purposes. They are a popular mechanism for unfolding details about what an application does in the background and how certain functionalities are implemented. In particular, these mechanisms can be used to reveal the purpose and behavior of a malware sample.

- **Malicious reasons:**
  Another field of application for hooks is malware. More precisely, some malware types use this concept to add unwanted functionality to applications or systems. For instance, rootkits can use hooks in order to obtain privileged access to a system and to hide its presence (cf. [HB08, p. 99 ff.]). Another example are so-called *wallhacks* that apply hooking techniques in order to put oneself in advantage in computer games.

Classification of Hooks

According to Hoglund and Butler [HB08, p. 99 ff.], there are two major categories of hooks: *kernel hooks* and *userland hooks*. As we have mentioned before, the virtual memory of a process is divided into user memory and kernel memory. Thus, kernel hooks are stored in kernel memory and userland hooks in user memory.

Kernel hooks are very well hidden because programs run in user mode and thus usually do not have access to kernel memory. Consequently, these hooks are very difficult to detect which explains their usage in rootkits. However, it is not easy to place them in this memory portion. Kernel hooks are often implemented through device drivers. Hoglund and Butler [HB08, p. 105 ff.] describe two different methods of using device drivers to place kernel hooks. One way is the manipulation of the so-called *System Service Dispatch Table* (SSDT), which stores the memory addresses of native system services providing the Windows API with access to system resources. Thus, there is the possibility to change addresses of the SSDT to a customized system service. The second method of implementing kernel hooks is to modify the *Interrupt Descriptor Table* (IDT) so that it points to an own-implemented interrupt handler, which is executed if the corresponding interrupt is triggered.

In contrast to kernel hooks, userland hooks are relatively easy to implement because a target process's user-level memory can be modified as long one has access to its address space. Yet, there is the drawback that these hooks can be detected by the target process. Techniques of implementing userland hooks are described in 2.3.2.

In this thesis, we apply userland hooks for the purpose of dynamic malware analysis. In order to counteract the drawback of being detectable, we use certain mechanisms in order to hide the analysis environment. Thus, in the following, we focus on characteristics and implementation methods of userland hooks.

API Hooking

In Section 2.2.3, we have seen that APIs are functions provided by a piece of software that enable a developer to use the software's resources or functionality. *API hooking* is the concept used to intercept those API functions. Like other DLLs, APIs are loaded into user memory when the executable is started. Thus, API hooking is a special type of hooking, or more precisely, an API hook is a special type of userland hook. Using API hooks, the code of API functions or entries pointing to their memory addresses can be modified in such a way that the corresponding API calls are intercepted and redirected

to the hook functions, which then can perform their own functionality.

API Hooking Methods

As mentioned before, in order to hook an API function, the original execution flow has to be intercepted and redirected to the hook function in order to take over control. There are different points in a process or system where such an interception can take place. Based on these points, various hooking methods exist. Examples are presented by Ivanov[Iva02] and Engelberth[Eng07, p. 31 ff.]. In this section, several of these methods are outlined.

Proxy DLL One possible hooking method is to use so-called *Proxy DLLs* or *Trojan DLLs*. Here, the original DLL which contains the API functions to be hooked is replaced by the Proxy DLL. This way, when an executable using the respective library is started, the modified library is loaded into the process's address space. Then, stub calls can be used in order to not only replace the original DLL but also to intercept the calls and additionally provide the ability to use the original functionality. One technique of realization entails the following steps:

1. Backup the original file by renaming or copying.
2. Replace it with our modified DLL by using the original DLL file name.
3. Execute our own code.
4. Forward calls to the original DLL, or reject them and transfer control back to the calling operation.

Thus, this technique is rather simple to implement. However, it also comes with some considerable disadvantages:

- One major drawback is that all functions that are exported in the original library have to be exported in the Proxy DLL as well. Otherwise, it would cause linking errors which could reveal the Proxy DLL's existence. This is especially inconvenient if there are a lot of exported functions in the original library but we only want to hook a small number of them.
- Furthermore, one has to be careful when implementing the stub calls. Each one of them has to use exactly the same amount and types of arguments as the functions in the original DLL. Otherwise, this would also lead to errors disclosing the hooking environment.
- Another major disadvantage is that this type of hook is very easy to detect as the DLLs are simply replaced by other files. Therefore, proxy DLL and original DLL almost certainly have different file sizes, which could reveal the proxy DLL's existence. In particular, checksums can be used in order to detect this kind of hooks.

IAT Hooking The *IAT hook* is derived from the layout of the PE file format, which we discussed in Section 2.2.4. According to Ivanov [Iva02], the idea of using the IAT for

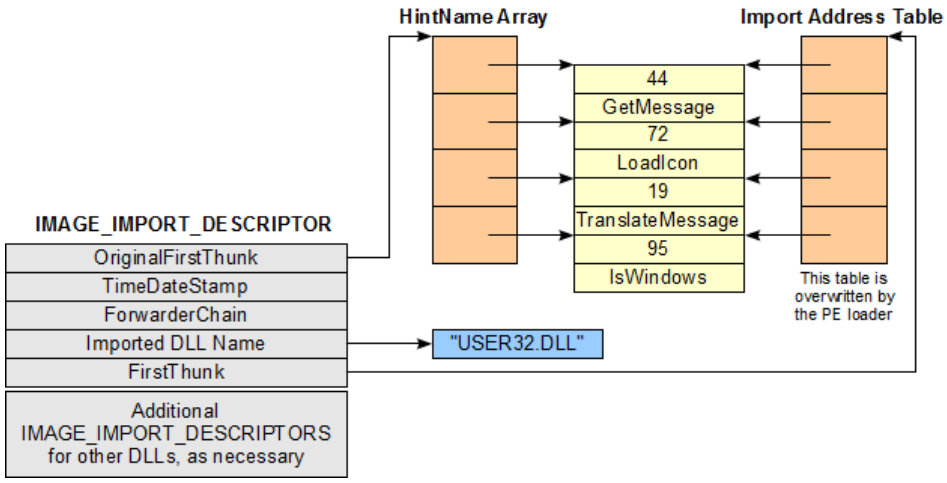

Figure 2.3.: The `IMAGE_BASE_DESCRIPTOR` data structure according to Pietrek [Pie02b]

hooking was first published by Pietrek [Pie94].

As mentioned before, API functions are provided in other executable files. If an application uses APIs, the DLL files containing the APIs functions are loaded into the process's address space as soon as the application is started. The information about all imported modules and functions is stored in the `.idata` section of the importing executable containing an array of `IMAGE_IMPORT_DESCRIPTOR` data structures. For each imported module, there is one `IMAGE_IMPORT_DESCRIPTOR` data structure providing information such as the name of the imported DLL file and also two pointers to arrays, one of them being the IAT. The `IMAGE_IMPORT_DESCRIPTOR`'s layout is depicted in Figure 2.3. During startup, the PE loader parses through all `IMAGE_IMPORT_DESCRIPTOR` structures and loads all relevant executables into memory, parses their export tables, and inserts the resulting address information about the imported functions into the executable's IAT. Consequently, when an API is used by the application, an indirect call using the IAT is executed in order to get to the function's actual address.

The IAT hook uses this layout by modifying the addresses stored in the IAT in order to redirect these calls to its own hook functions. In doing so all calls to the API functions are redirected to the respective hook function. In order to manipulate the IAT, we have to discover the location of the IAT by parsing the PE file format for the right `IMAGE_IMPORT_DESCRIPTOR` structure, locate the IAT array, and change the corresponding function's address information. However, the `.idata` section is not necessarily writable. In such a situation, the hook has to use the `kernel32.dll`'s `VirtualProtect` API in order to change the section's access rights. The technique of IAT hooking is described in more detail by Richter[Ric99], Robbins[Rob00], and Hoglund and Butler[HB08, p. 97-98].

Thus, IAT hooks are relatively simple to implement but they also have various disadvantages (cf. Hoglund and Butler[HB08, p. s98]):

- IAT hooks can be detected very easily.
- These type of hooks does not always work. If an application uses *late binding*,

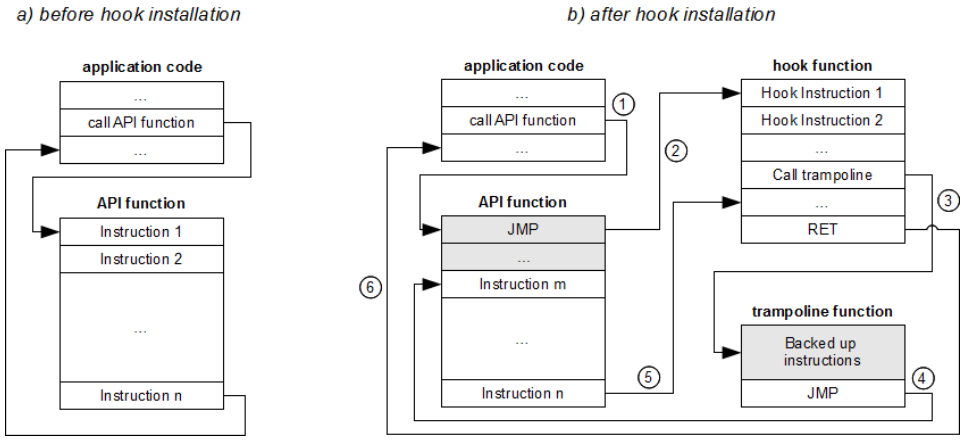

Figure 2.4.: Inline hooking (cf. Engelberth [Eng07, p. 33])

i.e. the referring function's memory address is not looked up at the application's initialization but during runtime, the IAT hook will be ineffective because the customized address information in the IAT is changed again to the correct address.

- If the API function is called dynamically during runtime by usage of the APIs `LoadLibrary` and `GetProcAddress`, the IAT hook will proof ineffective as well. This is because in this case the IAT is not used.

Inline Hooking The third and last hooking method described in this thesis is *inline hooking*. As we will see, it is more powerful than IAT hooking. Inline hooking can also be used for kernel hooks but is more commonly applied in userland hooks. Instead of modifying table entries or manipulating information that leads to the target API function, inline hooks directly overwrite the function's code bytes in memory. Of course, it would not be useful to simply overwrite the entire API function with own code. Instead, only the first instructions are overwritten with a five byte jump instruction to the hook function. Figure 2.4 illustrates the inline hook mechanism in more detail.

1. Somewhere in the application's code, the hooked API function is called with its parameters. This way, the instruction pointer is set to the address of the first instruction of the original API function.
2. However, the first instructions of the API function have been overwritten. Instead, an unconditional jump instruction is executed. The offset of the jump command is added to the instruction pointer. The consequence is that the instruction pointer is set to the address of the hook function.
3. Now, the hook can perform its own functionality and execute its instructions. In order to use the API function's original functionality, it is not possible to simply call the API because this function has been hooked, and doing so would end up in an infinite loop. Instead, the hook function can use the trampoline function in order to execute the actual API function. For this purpose, the trampoline function is called with a regular function call and with all its arguments. As the API is not called directly, the call of the trampoline function is also referred to as

detour.

4. The trampoline function executes the instructions, that have been backed up in Step 1. Subsequently, an unconditional jump to the rest of the original function after the overwritten bytes is used to perform its remaining instructions. After the function has completed its task, the result is returned to the hook function.
5. The hook function can now either return the received result to the application code which has called the API in the first place, or return an appropriate customized value.
6. Finally, the application code receives the result and can proceed.

In order to use inline hooking, we have to perform the following steps:

1. In a first step, we back up the first bytes of the API function to be hooked which will be overwritten, and store them in another memory area of the trampoline function. However, since instructions can be of different size, we have to make sure that no instructions are split. This would lead to a crash of the application and could reveal the hook's existence.
2. We have to insert an unconditional jump command to the address following the stored bytes in the trampoline function in order to ensure that the trampoline function will return to the original function and execute its remaining instructions. Thus, the target of the jump is the memory address after the backed up bytes in the original function.
3. Then, we overwrite the mentioned first bytes of the original API function code with an unconditional jump instruction. For this purpose, memory space of five bytes is needed. One byte is used for the jump instruction represented by the opcode `0xE9` and the remaining four bytes are required to determine the offset to the hook functions.

After these three steps, the hook function can perform all desired instructions and even call the original function by using the trampoline.

The advantages of this kind of API hook is that in contrast to IAT hooks, every type of function can be hooked and not only functions that are imported from other executables. Furthermore, it is irrelevant which type of command is used in order to call the API function. It does not matter whether the IAT or the APIs `LoadLibrary` and `GetProcAddress` are used to call the function. Neither does it matter whether late binding is used or not. This is why inline hooks are more effective than IAT hooks.

However, there are also a few disadvantages. Firstly, one has to be careful that no instructions are split when overwriting the original function. For instance, this problem can be solved by disassembly of the corresponding instructions. Another solution is to compare the byte pattern to familiar ones in order to determine how many bytes have to be overwritten. Moreover, there must not be any unconditional jumps in the bytes which are overwritten. This would cause the application to crash.

In this thesis, we use inline hooks in order to intercept API functions and determine the purpose and behavior of malware.

2.3.3. DLL Injection

So far, we discussed different types of malware analysis and different methods to intercept functions via hooks for the purpose of malware analysis. However, in view of our goal to use hooks in order to determine a malware's behavior there is one problem left. We covered how the hooks work but not how to place the hook functions in the target process's address space and how to initialize the hook installation process. The mechanism we apply for this purpose is called *DLL injection*. Subsequently, we have to ensure that the modifications of the code bytes in memory are performed in order to install the hooks.

Injection Methods

In the following, we three different DLL injection methods . According to Hoglund and Butler [HB08, p. 101], these methods were first introduced by Richter [Ric94].

Injection via Windows Registry The *injection via Windows registry* uses a certain registry key of the Windows registry in Windows Versions NT, 2000, XP, and 2003 in order to inject an arbitrary DLL file into the target process's address space. The key's registry path is `HKEY_LOCAL_MACHINES\Software\Microsoft\WindowsNT\CurrentVersion\WindowsAppInit_DLLs`. If a DLL file's path is added to this key, all applications using the `user32.dll` and thus using a GUI will load the DLL into its memory.

This injection method is very easy to implement but also entails several major disadvantages:

- As the registry key to be modified is only available in the Windows Versions NT, 2000, XP, and 2003, the injection technique is limited to those versions.
- This type of injection is dependent of the usage of the `user32.dll`. Therefore, it is not possible to inject a DLL into arbitrary processes but only those using a GUI.
- Using the injection via Windows registry, the customized DLL is loaded into all processes which import the `user32.dll`. Thus, it is not possible to select only certain processes or applications for injection.

Injection via Windows Hooks In the Windows operating system, if a key or a button is pressed, this causes a Windows event about which all running applications using a GUI are notified by messages. Microsoft provides so-called *Windows hooks* by which these messages can be intercepted. Thus, these hooks can be used in order to load a DLL into the address space of a remote process. Table 2.2 displays the function used to install a Windows hook.

The first argument `idHook` is an identifier that defines the hook's type depending on the triggered event. Altogether, there are fifteen different available hook types. The second argument `lpfn` points to the hook procedure to be executed if the corresponding event is triggered. `hMod` is a handle determining the module and thus the DLL containing the procedure. The last argument `dwThreadId` identifies the thread to which the hook is attached. If it is defined as `NULL`, the hook is attached to all threads thus representing

Argument	Type	Comment
idHook	int	
lpfn	HOOKPROC	
hMod	HINSTANCE	
dwThreadId	DWORD	
Return Type	HANDLE	

Table 2.2.: The `SetWindowmsHookEx` interface

Argument	Type	Comment
hProcess	HANDLE	
lpThreadAttributes	LPSECURITY_ATTRIBUTES	
dwStackSize	SIZE_T	
lpStartAddress	LPTHREAD_START_ROUTINE	
lpParameter	LPVOID	
dwCreationFlags	DWORD	
lpThreadId	LPDWORD	out
Return Type	HANDLE	

Table 2.3.: The `CreateRemoteThread` interface

a system-wide hook. The return value of the API function is a handle, which can later be applied to uninstall the hook using the API `UnhookWindowsHookEx`.

Consequently, injection via Windows hooks offers some advantages in comparison to the injection via Windows registry:

- There are no limitations regarding Windows versions.
- The hooks can be uninstalled as soon as they are not required any more.
- By using the last argument of `SetWindowsHookEx` it is possible to solely hook a certain thread and use the DLL focused on the target.

However, there is also the drawback that Windows hooks can have a significant impact on the system performance since for each event message, the system's list of Windows hooks has to be parsed.

Injection via Remote Thread The last DLL injection method described in this thesis is the *injection via remote thread*. In contrast to the methods described above, this technique does not make use of any Windows settings. Instead, a small program has to be written that creates an additional thread in the target process. Subsequently, the created thread loads the DLL file. For this purpose, two Windows API functions which are provided by `kernel32.dll` are required: `CreateRemoteThread` and `LoadLibrary`.

`CreateRemoteThread` displayed in Table 2.3 takes 7 arguments. We focus on the the three most ones important regarding DLL injection. The first argument `hProcess` takes the process handle of the target process, in which the new thread is created. The

handle can be obtained via the `OpenProcess` API. `OpenProcess` only needs the target process identification number¿[7] The second important argument is the fourth parameter `lpStartAddress`. This argument takes a pointer to the function, which is executed as soon as the thread is created. In order to load a DLL file into the process address space, the memory address of the `LoadLibrary` API has to be passed to this argument. The memory address of `LoadLibrary` can be located by examining the injecting program's virtual memory since Windows applications usually store this API at the same address due to performance reasons. Therefore, the API functions `GetProcAddress` and `GetModuleHandle` can be used to locate `LoadLibrary`. As `LoadLibrary` requires the DLL file's path as an argument, the path has to be stored as a char array in the target process's virtual memory. This can be done by applying the `VirtualAllocEx` and `WriteProcessMemory` APIs. The memory address must be passed to the fifth argument of `CreateRemoteThread`, the so-called `lpParamater`.

In comparison to the DLL injection methods mentioned above, the injection via remote thread performs considerably better. Also, it is possible to focus on a single target process. Consequently, there are no redundant injections into further processes. On the other hand, there is the drawback that this method does not work for older Windows versions before Windows NT. Furthermore, we have to note that in order to inject executables into system processes using this method, one has to obtain debug privileges. Otherwise, there might be access violations.

In this diploma thesis, we use DLL injection via remote thread because this method has a relatively low impact on the system performance, and it can be used to inject our code into all processes, no matter whether they use a GUI or not.

Post-Injection: Hook Installation Initialization

As soon as the modified executable is loaded into the target process address space, our analysis environment is almost complete. The DLL containing the hooking environment is already loaded but the hooks are not yet installed. For the installation process, we have to ensure that the DLL uses its main entry point. This means that the code compiled to our DLL has to implement the function `DllMain` executing all required tasks to install the hook library. Further implementation details concerning this issue can be found in Section 3.4.6.

2.4. Related and Concurrent Work

In this section, we briefly describe selected publications that are related to this diploma thesis.

[7] The process identification numbers of all running processes can for example be listed by the Windows Task Manager.

2.4.1. Related Work

The increasing amount and diversity of malware has caused IT security researchers to move towards automating the analysis of malware samples. Therefore, dynamic malware analysis and automation are no new topics in this field. In particular, there exists an array of publications and tools on this topic. In this section, we review some of this work.

CWSandbox / GFI Sandbox

CWSandbox[8] is a research tool developed by the Chair for Practical Informatics 1 at the University of Mannheim. Licensed by the company Sunbelt[9], it is now sold as a commercial product called *GFI Sandbox*. GFI Sandbox is said to be the industry's leading tool concerning dynamic malware analysis.

It can be used to run a behavioral analysis of a malware sample during its runtime and thus reveal its activities. In particular, it provides information about

- executed applications,
- caused system changes such as changes to the Windows registry or to the file system, and
- generated network traffic.

It also emulates user interaction in order to create an environment for the malware sample that is as authentic as possible while not jeopardizing system security. More precisely, security is ascertained by running the analysis in a virtual machine. For this purpose, GFI Sandbox supports VMWare[10] as virtualization software. After the monitoring process, a detailed analysis about the monitored actions is created. This analysis is written to an XML file and can be used for further analysis. Moreover, it can be compared to multiple other reports and one may derive the threat and severity posed by the observed object.

Much like PyBox, CWSandbox monitors API calls in order to draw conclusions of the malware sample's activities. Also, according to Willems et al. [WHF07], CWSandbox makes use of inline hooking as the API hooking method as well as DLL injection via a remote thread in order to intercept, observe, and log every relevant API call with all its arguments. Like PyBox, the hooks are installed in user-mode allowing the environment to focus on the observed executables. Thus, every single action performed by the malware sample is inspected and revealed to the security researcher.

[8] http://mwanalysis.org/
[9] http://www.sunbeltsoftware.com/
[10] http://www.vmware.com/

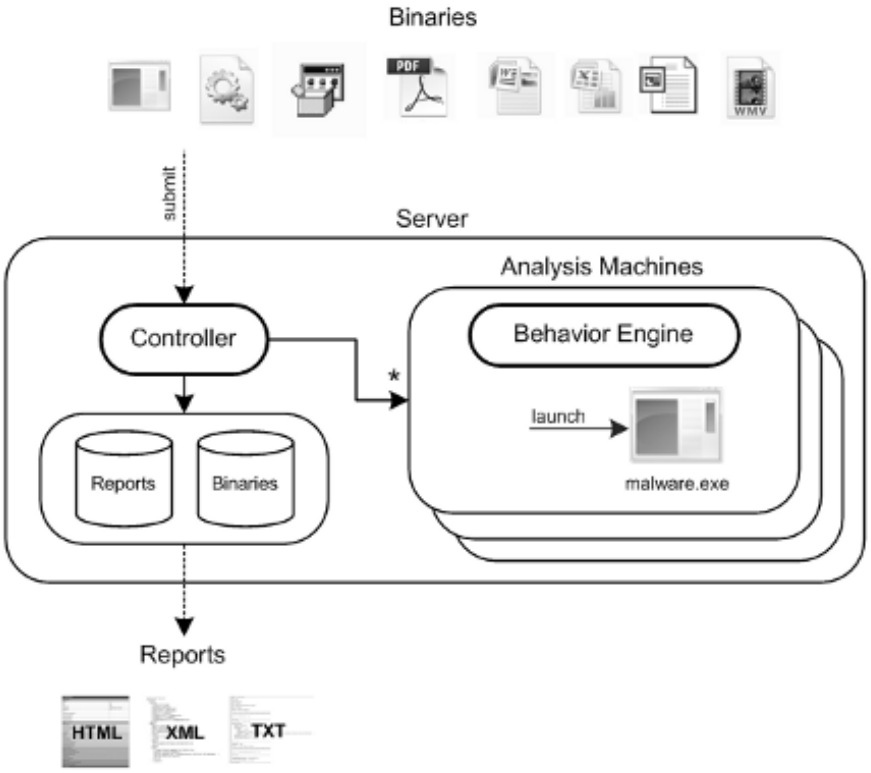

Figure 2.5.: Joe Sandbox analysis procedure according to Joe Security [joe11]

Joe Sandbox

Joe Sandbox is another another tool that enables fully automated analyses of malware samples. Like CWSandbox and PyBox, it executes malware in a secure, controlled environment and provides detailed reports as output. The analysis procedure of Joe Sandbox is depicted in Figure 2.5.

As we can see, Joe Sandbox consists of two basic elements: a *controller* and a set of *analysis machines*. The controller manages the analysis process, whereas the analysis machines provide the secure environments in which the objects to be observed are executed and monitored.

The analysis process requires six steps:

1. The binary to be monitored is submitted to Joe Sandbox and is stored in a database.
2. An analysis machine is created. It incorporates a so-called *behavior engine* which provides the monitoring functionality.
3. The submitted binary is copied to the analysis machine.
4. The binary is executed and monitored. During the execution, each observed event is sent to the controller and evaluated.

5. The analysis machine is restored to a clean system state.
6. The report containing all observed behavior is created and sent to the user.

The primary difference between Joe Sandbox and PyBox or CWSandbox lies in the behavior engine. While PyBox and CWSandbox use hook libraries installing user-mode hooks, the behavior engine of Joe Sandbox is implemented as a Windows driver and therefore runs in kernel mode. This implies two advantages. Firstly, it can also detect malware running in kernel mode. Secondly, it is much more difficult for malware to detect the analysis environment.

Based on the monitored information, Joe Sandbox tries to detect known behavioral schemes that are typical for malware. For this purpose, it compares the received data to known signatures in its signature engine. Furthermore, Joe Sandbox provides support for several operating schemes. It can be run in combination with either a virtualization software such as VirtualBox or VMWare, or with a emulator. However, the operating systems on which the analysis has to occur are limited to the Windows Versions XP, Vista, and 7.

2.4.2. Concurrent Work

Due to the benefits of automated dynamic malware analysis solutions and the easy extendability of such solutions developed in Python, there are currently several projects addressing the implementation of Python-based sandboxes. Apart from this thesis, there is a further project which has been named "PyBox" and pursues the same goal (cf. Leder and Plohmann [LP10]). However, they pursue a different approach: While in our approach a separate hook library is implemented and compiled providing the monitoring functionality, Leder and Plohmann inject a Python interpreter into the process to be monitored and provide their monitoring functionality within external Python scripts. This provides more flexibility and a higher degree of configurability.

2.5. Summary

At the beginning of this chapter, we discussed malware in its various forms and how samples can be detected. Like on any other operating system, these threats affect the behavior of Windows. Hence, in order to be able to analyze, monitor and predict the caused changes to the system, it is important to provide a basic understanding of the system's functionality. This is why in the second part of this chapter, we outlined some major fundamentals of the Windows operating system such as memory management, the formats of executable files and how processes are created out of these executable files. Based on these fundamentals, different hooking and injection methods have been introduced which serve the purpose of injecting customized code into other processes, intercepting functions and redirecting them to the infiltrated functionality which in turn can report the malware's activity.

Chapter 3.

Implementation

In this chapter, we describe the functionality of PyBox, how it works, and how it is implemented. In particular, we detail to what purposes PyBox can be used, how one can make use of its functionality, and what happens inside PyBox when it is executed.

In the first section, we define the design goals PyBox has to meet. For this purpose, we specify functional design goals that have to realized in order to provide a suitable malware analysis environment as well as non-functional design goals that have to considered.

In Section 3.2, we outline the setup of PyBox including all of its components. In particular, we provide an overview of these components' purposes and describe how they work together and communicate with each other. Additionally, the various steps to produce and output the required information in the form of an XML-based report are outlined.

Based on this overview of the PyBox setup, the subsequent sections outline the implementation details concerning different components of the analysis environment. In Section 3.3, we describe the applied software simulating the target system. The implementation of the hook library including the hook installation and callbacks is covered in Section 3.4. In Section 3.5, we detail the implementation of the analysis tool that processes the monitored information of the hook library and writes them to an XML file. Finally, we discuss the communication methods used by the hook library and the analysis in order to exchange information such as the monitored activites or the settings defining which functions have to be intercepted.

3.1. Design Goals

Considering the task of this thesis and the described prerequisites, we can derive several design goals which can be divided into functional and non-functional goals. The functional design goals concern the required features of PyBox which are listed in the following.

Secure Environment A sandbox has to provide a secure environment in which an executable such as a malware sample can be executed. Therefore, we have to ensure that the malware finds no opportunity to replicate itself by means of a network device or any other device. Furthermore, it is important to provide a feature that allows to undo the

changes caused by the observed executable guaranteeing that an analysis does not have any influence on a subsequent one.

Monitoring System Interaction In order to draw conclusions about the behavior of the observed executable, we have to monitor its interaction with the operating system. Thus, the analysis environment has to control all specified system interaction activities of the executable.

Traceability Some malware samples use detection mechanisms, which can tell when they are scanned. Therefore, we have to implement functionality that hides the analysis environment as well as possible in order to not be detected.

Machine-readable Report Finally, the program's output has to provide a machine-readable report containing all relevant information. The report has to be exported to a file so that it can be put to further use.

Apart from the functional design goals, there are some non-functional design goals that have to be considered before focusing on the implementation details. These non-functional design goals mainly concern the usage and further development of PyBox.

Usability PyBox is an open-source sandbox, which is implemented for usage and further development in teaching and research. Hence, it should be easy to use. A potential analyst should not have difficulties when trying to configure and utilize the functionality of PyBox. Also, the output report has to provide its information in a understandable way allowing to draw valueable conclusions from it.

Extendability The analysis environment serves the purpose of observing the behavior of executables in Windows. However, in view of the development concerning malware attacks on other platforms such as mobile devices, future developers might consider to extend the functionality of PyBox in order to analyze executables on other operating systems as well. Therefore, PyBox should be extendable.

3.2. System Overview

The analysis process consists of four components: the PyBox virtual machine, the PyBox analysis tool, the hook library, and the target process.

PyBox is a sandbox and serves the purpose of analyzing malware. In order to prevent the malware from causing permanent changes to the analysis system, we run the analysis in a virtual machine. A snapshot of the original system state ascertains a clean system state for each analysis process. Thus, we can perform the analysis uninfluenced by earlier changes through prior analyses.

During the ordinary execution of an executable, a large part of the activity happens in the background and is not shown to the user. The user only perceives the result of these activities. In particular, malware usually is designed to hide its execution. Usually, neither the user nor other software such as virus scanners are supposed to detect the

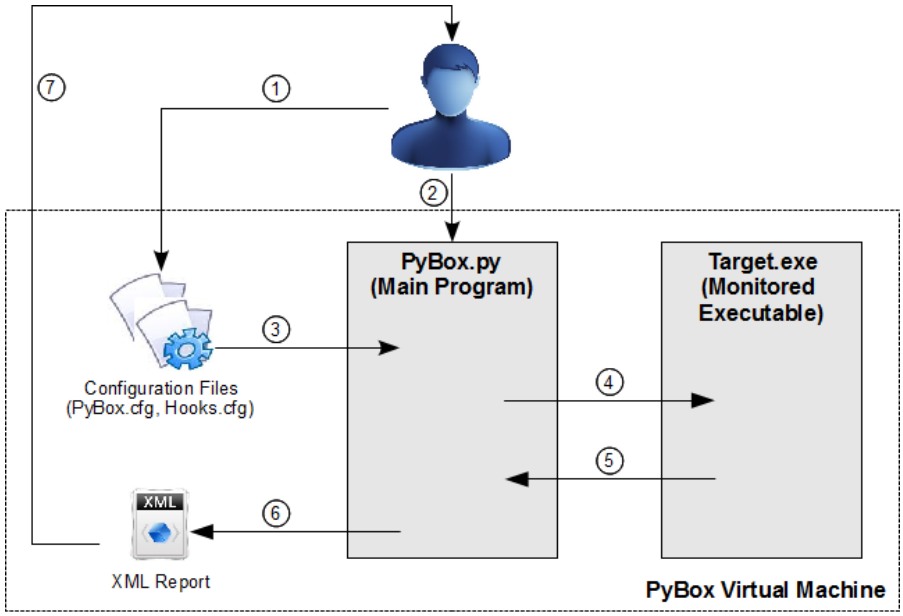

Figure 3.1.: PyBox analysis procedure

malware so that it can serve its malicious purpose as long as possible. Therefore, PyBox has to reveal these actions in the target process, specifically those which make use of system functionality. In PyBox, we achieve this through monitoring the calls to the native API by using API hooking and DLL injection. For this purpose, two components are necessary: the *hook server* and the *hook library*.

The PyBox analysis tool `PyBox.py` acts as hook server. It adjust the setup according to the settings defined in the configuration files, creates the target process, injects the hook library into the target process, receives and processes the log data, and generates a report. The hook library, which is the file `pbMonitor.dll`, implements the hooking and monitoring functionality. It is responsible for installing the specified hooks, monitoring the system calls, and the creation of log entries. As the two components are closely related and have to interact with each other, a means of *inter-process communication* (IPC) is implemented that allows the two processes to communicate with each other and exchange information.

The necessary steps of the analysis process are depicted in Figure 3.1 and described in the following:

1. As soon as the PyBox virtual machine is up and running, and a malware sample to be observed is provided, the user configures the two configuration files `pybox.cfg` and `hooks.cfg`, which belong to the PyBox analysis tool. In `pybox.cfg`, we can configure the main settings such as the file path to the hook library (`pbMonitor.dll`) to be injected, the file path to the target application as well as the path to the folder into which the report is written. In `hooks.cfg` we can specify the native API functions to be hooked including customized return values if the result of the

original functions shall not be returned.

2. Having configured the settings, we are able to execute the PyBox analysis tool (`PyBox.py`). In doing so, we pass the path the to created `pybox.cfg` file as an argument and run the analysis process.
3. The PyBox analysis tool reads all configuration files and adjusts all settings.
4. It creates the target application's process and uses DLL injection in order to inject the hook library `pbMonitor.dll` into the target's address space. The hook library installs itself and hooks all API functions specified in `hooks.cfg` by redirecting them to the corresponding callback functions also included in `pbMonitor.dll`.
5. With the hooks being installed, the hook library creates a log and sends it to the PyBox main application each time a hooked API function is called during the execution of the target process.
6. As soon as the execution of the observed object is terminated, the logged data are written to a XML file. Thus, the information of the analysis process can be stored and exported for further use.

Following this procedure, the virtual machine can be restored to the original system state by usage of the snapshot mentioned above.

3.3. Sandbox Environment

In the previous section, we mentioned that we use a virtual machine in order to provide a secure environment in which an arbitrary malware sample can be executed. For this diploma thesis, we use VirtualBox[1] (version 4). VirtualBox is a virtualization software for x86, AMD64, and Intel64 architectures. According to Oracle [Ora11a, p. 11], it can be run on various operating systems such as Windows, Linux, Mac OS and OpenSolaris to run various guest operating such as all NT-based Windows versions and Linux versions using kernel 2.4 or 2.6. The virtual machine used for PyBox runs a Windows XP operating system. All network devices are disabled in order to prevent the propagation of malware. Also, VirtualBox offers a snapshot functionality. This enables us to restore a clean system state for each new analysis process. The VirtualBox SDK [Ora11b] provides an API with a set of functions allowing to control virtual machines via console or programming languages such as Python and C++.

3.4. Hook Library

The hook library implements the hooking and monitoring functionality. As we have described in 2.2.1, each process has only access to its own address space and thus can only execute code that is located within its memory. However, since we want our own

[1] http://www.virtualbox.org

customized code to be executed in the context of the target process, we have to implement the code as a DLL and must load it into the target process's address space.

In Chapter 2.2.4, we have described the executable file format of Windows. Other operating systems such as Linux use different formats such as the ELF format which we will outline in Chapter 4.1. Obviously, executables are system-specific. They can only run on the system they are compiled for. Therefore, we have to use the respective file format and cannot generate a common hook library for different operating systems. In order to use PyBox on different operating systems, a seperate new hook library has to be implemented for each one.

Thus far, PyBox is limited to Windows NT-based operating systems and thus only provides a hook library for Windows which is named `pbMonitor.dll`. Hook libraries for different operating systems are beyond the scope of this diploma thesis.

In this section, we discuss the implementation details of the hook library. First, we outline the hook library's structure and the order in which its components are used. Then, the various components are described. We depict how the function hooking method is implemented and how the callback and trampoline functions are realized. Additionally, callback examples demonstrate their utilization. Subsequently, we describe some special detection prevention hooks that hide the analysis framework. Finally, we demonstrate how the hooks are installed.

3.4.1. Hook Library Overview

`pbMonitor.dll` is implemented in *Visual C++*. C++ was chosen as programming language since we cannot use Python to create a DLL and we have to make much use of various API functions provided by Windows, which often require the use of specific C data structures. Therefore, C++ is an appropriate choice to implement the required functionality.

The structure of `pbMonitor.dll` is depicted in Figure 3.2. The file `dll-injection.h` provides the functionality to inject the hook library into other processes that are created by the currently observed target process ensuring that all executed processes of called by a malware sample are monitored. The specified settings concerning all hooks are defined in an array in `settings.h`. The array specifies which functions have to be hooked. Furthermore, it provides data required for hooking them such as module information of each function as well as the corresponding memory addresses of the callback and trampoline functions. The information concerning the settings of PyBox is retrieved from the PyBox analysis tool and filled in as soon as the target process is started. `log.h` implements the functions and required data structures to create a log entry. The retrieval process of the settings, their structure as well as the creation of logs are described in more detail in Chapter 3.6.

While `apiHook.h` implements a class providing the functionality to install single hooks, the files `callback.h` and `trampoline.h` implement the callback and trampoline functions containing the added customized code. Additional native API data structures, which are required by the callback and trampoline functions, are defined in

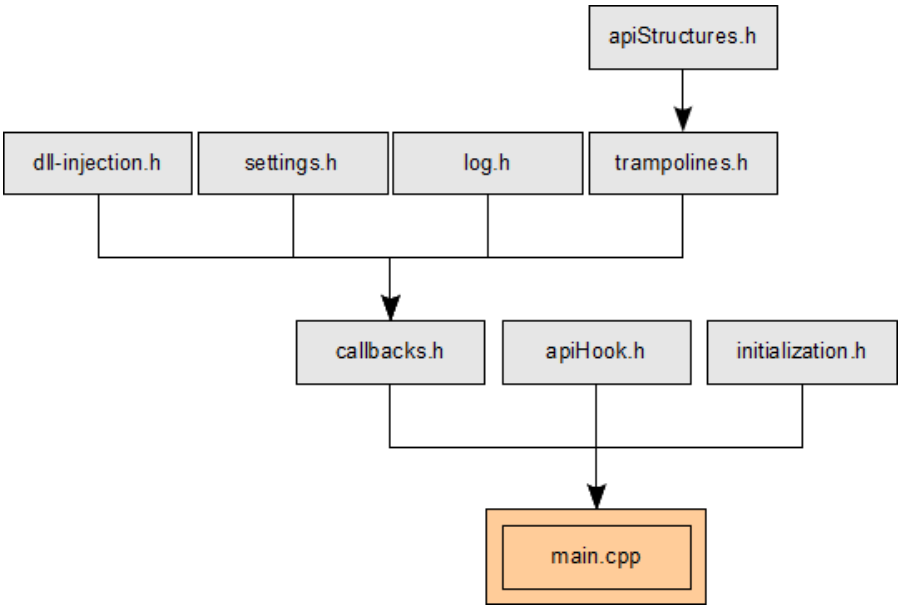

Figure 3.2.: Hook library layout

`apiStructures.h`.

All the functionality provided by the various headers mentioned are applied in `main.cpp` in the DLL main entry point initializing the hook installation.

3.4.2. Function Hooking

In PyBox, we make use of API hooking in order to intercept specific functions by redirecting them to our customized code that implements the monitoring functionality. In Chapter 2.3.2, we have described three different hooking methods which can be used for API hooking. In our hook library, we use inline hooking as our chosen method. Therefore, we have to overwrite the first few code bytes in memory of the functions which we want to intercept. The overwritten code bytes are stored in a trampoline function and are replaced with a jump instruction to the callback function running our own code. We have to store the original bytes because we do not necessarily want to prevent the execution of all API functions which are covered by PyBox. Instead, we just want to intercept those functions in order to observe the execution flow and still be able to call the original functionality as if nothing happened.

We choose inline hooking because it seems to be the most appropriate userland hooking method regarding our objective to monitor malware. According to Willems et al. [WHF07, p. 33], inline hooking is one of the more efficient and effective methods. Therefore, this method is applied in CWSandbox as well. Furthermore, it is not as easy to detect as other hooking methods, and it is not specific to a certain function type.

```
170 void __stdcall apiHook::installHook()
171 {
172   setDebugPrivileges();
```

```
173   HANDLE h_process = GetCurrentProcess();
174   if (h_process != 0)
175   {
176     unsigned char byte;
177     DWORD jump_distance, dwOldProtect;
178     short offset =
          calculateBytesToOverwrite(h_process,this->address_api);
179
180     // Backup first bytes of original API function
181     DWORD temp_api = this->address_api;
182     DWORD temp_trampoline = this ->address_trampoline;
183     VirtualProtect((void *)this->address_trampoline, offset+5,
          PAGE_WRITECOPY, &dwOldProtect);
184     for (int i=0; i<offset; i++)
185     {
186       byte = readByteFromMemory(h_process, temp_api);
187       writeByteToMemory(h_process, temp_trampoline++, byte);
188       byte = 0x90;
189       writeByteToMemory(h_process, temp_api++, byte);
190     }
191     VirtualProtect((void *)this->address_trampoline, offset+5,
          PAGE_EXECUTE, &dwOldProtect);
192
193     // Write JMP operation back to original API function
194     byte = 0xE9;
195     jump_distance = temp_trampoline - (temp_api + 5);
196     writeByteToMemory(h_process,temp_trampoline++,byte);
197     writeOffsetToMemory(h_process, temp_trampoline, jump_distance);
198
199     // Writing hook to the original function address
200     VirtualProtect((void *)this->address_trampoline, 5+offset,
          PAGE_EXECUTE, &dwOldProtect);
201     temp_api = this->address_api;
202     VirtualProtect((void *)this->address_api, 5, PAGE_WRITECOPY,
          &dwOldProtect);
203     byte = 0xE9;
204     jump_distance = this->address_api - (this->address_callback + 5);
205     writeByteToMemory(h_process, temp_api++, byte);
206     writeOffsetToMemory(h_process, temp_api, jump_distance);
207     VirtualProtect((void *)this->address_api, 5, PAGE_EXECUTE,
          &dwOldProtect);
208
209     FlushInstructionCache(h_process, NULL, NULL);
210     CloseHandle(h_process);
211   }
212 }
```

Listing 3.1: Inline hooking

The implementation of the inline hooking method used in PyBox is depicted in Listing 3.1. At first, we determine how many bytes we have to backup and copy to the trampoline function. We cannot simply copy the first five bytes which required for the jump instructions because not all instructions use exactly five bytes. Some are shorter whereas some need more space. Therefore, we have to ensure that no instructions are split up and that we only copy full instructions to the trampoline. Otherwise, the hook would

crash the whole target process and would reveal its presence. One way to determine the required byte number is to disassemble the byte codes at this memory address. Here, we use another approach by exploiting a certain characteristic of API functions: Most API functions have a similar or even the same byte patterns. Thus, we simply have to scan the first bytes of a target function and match it to the byte patterns we know. From these patterns we can derive the number of bytes which we have to copy and overwrite. This is done by the function `calculateBytesToOverwrite` (line 178).

Now that we know the number of the corresponding bytes, we can copy them to the trampoline function before they are replaced with the jump instruction. As these bytes represent code that has to be executed, we have to change the page protection setting of this memory region from `PAGE_EXECUTE` to `PAGE_READWRITE` using the API `VirtualProtect` (line 183) in order to manipulate the bytes at these memory addresses. As soon as we have placed our code, we have to change the protection settings back to `PAGE_EXECUTE` (line 191) so that the code can be executed. In order to backup and replace the bytes, we have to read each byte, replace it with a `NOP` byte, and write the read byte to the trampoline. We use the API functions `ReadMemory` and `WriteMemory` to edit the memory at the specified addresses. `NOP` bytes are represented by the byte code `0x90` and tell the CPU to do nothing and skip over them. Here, we use `NOP` bytes because we do not necessarily need all the overwritten bytes in order to place the five-byte jump instruction.

In the next step, we write a jump instruction to the end of our trampoline function (lines 196, 197), which directs the execution flow of the trampoline instruction back to the original function after the backed up bytes have been executed so that the untouched rest of the original function can be executed. Thus, by calling the trampoline function, the full original functionality of the hooked API is provided.

Finally, we can calculate the jump distance from the original function to the hook function (line 135) and write the jump instruction `0xE9` combined with the calculated jump offset to the created `NOP` array (lines 136, 137) at the first memory address of the original function. Having set the page protection settings back to being executable, the function is ready to install hooks.

3.4.3. Callbacks and Trampolines

In the previous section, we have seen how inline hooking is implemented in our hook library. The result of placing a hook is that the execution flow is redirected through the callback and trampoline function. In this section, we describe the implementation of these two functions.

The callback function provides our own customized code. Within this function, we can practically do anything we want. As our goal is to observe the behavior of an executable, we implement the monitoring functionality in the callback function. Hence, this is where we create log entries and send them to the PyBox analysis tool. In case we still want to execute the hooked API function, we cannot simply call the original API function. This would lead us into an endless loop. For this purpose, we have to call the trampoline function, which contains the backed up first bytes of the original function

and then directs the execution flow to the untouched rest of the original function.

```
return_type calling_convention callback_function(arg1,...,argn)
{
  return_type status;
  info = obtain_iformation(arg1,...,argn);
  if (prevent_execution == false)
  {
    status = trampoline_function(arg1, ... ,argn);
  }
  else
  {
    status = customized_return_value;
  }
  create_log_entry(info);
  return status;
}
```

Listing 3.2: Callback function structure

The structure of a callback implementation is depicted in Listing 3.2. In such a function, all we do is either calling the original function via the trampoline function, or preventing its execution returning a customized value, and creating a log entry. The variables `prevent_execution` and `customized_return_value` represent settings that are defined in the `settings.h` header file for all hooks. The value of `prevent_execution` determines whether or not the trampoline function is called in order to execute the actual API function. If it is not called, we have to specify a customized value that returns an appropriate value to the caller. The function `create_log_entry` notifying the analysis environment of the API call and providing it with all relevant information about the call's parameters is called at the end of the callback function before the result is returned.

In contrast to the callback functions, we do not have to implement any logic in the trampoline functions because the content of the trampoline function is filled in by the hook installation described in the previous section. We only have to ensure that the trampoline provides enough space for the bytes that have to be copied from the original function. Therefore, we insert an `NOP` array of an appropriate size. For PyBox, we do not need to copy more than ten bytes, so that it suffices to use ten `NOP`s for each trampoline.

Other important aspecs are the consideration of the correct return value and argument types of each callback and trampoline function as well as the correct *calling convention*. If we do not take these aspects carefully into account, the application will crash since, as soon as the hook becomes effective and redirects the execution flow to the callback function, the stack is already aligned. The calling convention, however, defines how a function is called. In particular, it specifies how arguments are pushed on the stack, and in what order, and whether the caller or the callee manages the stack. According to Microsoft [Neta], Windows supports different calling conventions: the C calling convention *Cdecl*, the standard calling convention *StdCall* as well as *ThisCall* and *FastCall*. As the Windows API functions are implemented in the standard calling convention, we have to implement the callbacks and trampolines accordingly. Moreover, we have to use the same number and types of arguments as in the original API functions. If we did

Argument	Type	Comment
FileHandle	PHANDLE	out
DesiredAccess	ACCESS_MASK	
ObjectAttributes	POBJECT_ATTRIBUTES	
IoStatusBlock	PIO_STATUS_BLOCK	out
AllocationSize	PLARGE_INTEGER	optional
FileAttributes	ULONG	
ShareAccess	ULONG	
CreateDisposition	ULONG	
CreateOptions	ULONG	
EaBuffer	PVOID	optional
EaLength	ULONG	
Return Type	NTSTATUS	

Table 3.1.: The `NtCreateFile` Interface

not consider this aspect, the stack would no longer be aligned and the arguments could not be accessed. Consequently, this would affect the execution of the application and eventually lead to a crash. For these reasons, we use exactly the same arguments and return values as the original functions for our callback and trampoline functions.

3.4.4. Callback Examples

In the previous section, we discussed the structure of callback functions and what aspects we need to consider when implementing them. However, although this structure remains more or less the same, the exact implementation varies from callback function to callback function. Therefore, it is instrumental to demonstrate the actual code of different callback functions implemented in `pbMonitor.dll`. More precisely, we present three different examples of native API function callbacks, each one being from a different category. At first, we show `NtCreateFile_callback` from the category of file management APIs, secondly `NtSetValueKey_callback` from the registry category, and finally `NtCreateProcessEx_callback` from the process management category.

The NtCreateFile Callback Function

The function `NtCreateFile`[2] is provided by the native API. It is usually called in order to create a new file or to open an existing one. Its structure is depicted in Table 3.1. Like all native API functions, it has the Windows return type `NTSTATUS`, which is basically the type `long` in C/C++, and it uses the standard calling convention.

The interface requires eleven arguments that all have to be passed for a function call. In the following, we describe the individual arguments and their purpose:

[2] http://msdn.microsoft.com/en-us/library/ff566424(v=vs.85).aspx

1. `FileHandle` is a pointer to a file handle corresponding to the created or opened file after the function has been executed.
2. `DesiredAccess` defines the requested access rights concerning the file.
3. `ObjectAttributes` is a pointer to a structure containing information about the requested file such as the file name and its path.
4. `IoStatusBlock` is a pointer to a structure providing information about the completion status of the called function. For instance, it could return the values `FILE_OPENEND` or `FILE_CREATED`.
5. `AllocationSize` is an optional pointer to the size of the initial allocation in case a file is created or overwritten. This pointer can also be NULL implying that no allocation size is specified.
6. `FileAttributes` contains flags specifying the file's attributes. Such attributes can for example mark a file as read-only or hidden.
7. `ShareAccess` determines if a file can be accessed by different threads and how they can access it.
8. `CreateDisposition` provides information about how to react if the corresponding file already exists or if it does not exist.
9. `CreateOptions` determines additional options that become active as soon as the file is created.
10. `EaBuffer` is an additional argument used by drivers.
11. `EaLength` is an additional argument used by drivers.

As mentioned before, if we want to implement the callback function, we must use the same calling convention and the same arguments in the same order as the original function. Therefore, the callback function has to be implemented as depicted in Listing 3.3.

```
16 NTSTATUS NTAPI NtCreateFile_callback(
17     __out       PHANDLE FileHandle,
18     __in        ACCESS_MASK DesiredAccess,
19     __in        POBJECT_ATTRIBUTES ObjectAttributes,
20     __out       PIO_STATUS_BLOCK IoStatusBlock,
21     __in_opt    PLARGE_INTEGER AllocationSize,
22     __in        ULONG FileAttributes,
23     __in        ULONG ShareAccess,
24     __in        ULONG CreateDisposition,
25     __in        ULONG CreateOptions,
26     __in_opt    PVOID EaBuffer,
27     __in        ULONG EaLength
28 )
29 {
30     NTSTATUS status;
31     ...

57     if (hook_settings[0].preventExecution == FALSE)
```

```
58   {
59     // Call trampoline function
60     status = NtCreateFile_trampoline(FileHandle, DesiredAccess,
           ObjectAttributes, IoStatusBlock, AllocationSize,
           FileAttributes, ShareAccess, CreateDisposition, CreateOptions,
           EaBuffer, EaLength);
61     executed = 1;
62   }
63   else
64   {
65     // Get customized return value
66     status = (NTSTATUS)hook_settings[0].returnValue;
67   }
68
69   // Create log entry and return
70   SOCKET_ADDRESS_INFO sai = {0};
71   createLog(0, object, L"", "", executed, DesiredAccess,
       FileAttributes, ShareAccess, CreateDisposition, CreateOptions,
       sai, status);
72   return status;
73 }
```

Listing 3.3: The `NtCreateFile` callback function

As we can see, we use exactly the same arguments for the callback function that are also required for the actual API. Furthermore, the attribute `NTAPI` in line 16 defines the appropriate calling convention. `NTAPI` is defined as `__stdcall` informing the compiler to use the standard calling convention.

Having gathered all information regarding the passed arguments, the hook settings have to be checked to determine whether or not to call the actual API function. If not, a customized return value has to be returned. This check is implemented in line 57 using a value specified in the `hook_settings` array. This array contains a structure for each hook providing the respective hook information. The hook information structure for `NtCreateFile` is the first one in the array, i.e. it carries index value zero. If the hook settings determine to execute the hooked API function's orginal functionality in case it is called, the `preventExecution` value of the corresponding settings is set to `FALSE`. As a consequence, the trampoline function is called. To do this, the callback forwards the received arguments to the trampoline function (line 60). The trampoline's output value is stored in the variable `status`, which is eventually returned to the object that has called the API function (line 72). If `preventExecution` is set to `TRUE`, on the other hand, the trampoline function will not be called. Instead, the `status` variable is set to a customized value (line 66) defined in the settings with the key `returnValue` beforehand. After the return value has been determined, the callback creates a log entry notifying the analysis environment of the called API function and providing all relevant information. For this purpose, all relevant data obtained from the arguments passed to the callback function are forwarded to the log entry (line 71). Finally, the callback returns the value of `status` to the object that has called the API function.

Considering the creation of the log entry, a few aspects are important to note. As mentioned before, the function `createLog` receives various data providing detailed in-

Argument	Type	Comment
KeyHandle	HANDLE	
ValueName	PUNICODE_STRING	
TitleIndex	ULONG	optional
Type	ULONG	
Data	PVOID	optional
DataSize	ULONG	
Return Type	NTSTATUS	

Table 3.2.: The `NtSetValueKey` Interface

formation about the respective API call. For instance, an analyst can derive valuable insights from the arguments `DesiredAccess` and `CreateDisposition`. More specifically, these arguments entail information about the purpose of an API call, such as an attribute indicating whether the associated file has been opened, created, or overwritten. Another important piece of information that is passed to the log entry is the file path of the target file associated with the API call. All information concerning the target file of a `NtCreateFile` call can be found in the argument `ObjectAttributes`. As described above, `ObjectAttributes` is a pointer to a data structure of type `OBJECT_ATTRIBUTES`[3]. This structure contains the two fields providing the information we are looking for: `RootDirectory` and `ObjectName`. In order to obtain the complete path to a target file, we simply have to combine these two parts. While `ObjectName` is an object of type `UNICODE_STRING`[4] storing the file path in its member `Buffer`, `RootDirectory` is a file handle and, therefore, does not provide the information directly in the form of a text string. Hence, we first have to resolve this handle to the associated file name. For this purpose, we have to use another native API function: `NtQueryInformationFile`[5]. Having obtained the required information, both strings are combined in order to provide a complete file path. The resulting string is stored in the variable `object` and finally used to create the associated log entry.

The NtSetValueKey Callback Function

Another example of an implemented callback function is `NtSetValue_callback`. The interface `NtSetValueKey`[6] is applied to change the value of a registry key in Windows. Its interface is outlined in Table 3.2.

The interface receives six arguments:

1. `KeyHandle` is a handle to the registry key that contains the value to be changed.
2. `ValueName` is a pointer to a structure including the name of the value.
3. `TitleIndex` is a reserved parameter, which is usually set to `NULL`.

[3]http://msdn.microsoft.com/en-us/library/ff557749(v=vs.85).aspx
[4]http://msdn.microsoft.com/en-us/library/aa380518(v=vs.85).aspx
[5]http://msdn.microsoft.com/en-us/library/ff567052.aspx
[6]http://msdn.microsoft.com/en-us/library/ff567109(v=vs.85).aspx

4. `Type` determines the type of the value that is created or changed. Possible values are `REG_DWORD`, which represents the unsigned integer type or `REG_SZ`, which represents a null-terminated unicode string.
5. `Data` is a pointer to the memory address with the data that is supposed to be written to the key value.
6. `DataSize` determines the size of the data in bytes.

As described in Section 5.2, hooking this API function is quite important in order to reveal the behavior of a malware sample, because this interface is almost certainly used in case the target process causes changes to the Windows registry. Akin to the previous example, we have to retrieve all relevant information, execute the requested functionality with regard to the specified hook settings defined in `settings.h`, create a log entry providing the information required to determine the API call's purpose, and return an appropriate return value. The implementation of this callback is presented in Listing 3.4.

```
370 NTSTATUS NTAPI NtSetValueKey_callback(
371   __in      HANDLE KeyHandle,
372   __in      PUNICODE_STRING ValueName,
373   __in_opt  ULONG TitleIndex,
374   __in      ULONG Type,
375   __in_opt  PVOID Data,
376   __in      ULONG DataSize
377 )
378 {
379   NTSTATUS status;
380   wchar_t object[MAX_STR_BUFFER] = L"";
381   object[MAX_STR_BUFFER-1] = '\0';
382   wchar_t value_name[MAX_STR_BUFFER] = L"";
383   value_name[MAX_STR_BUFFER-1] = '\0';
384   unsigned char executed = 0;
385
386   //Determine object name
387   if (KeyHandle != NULL)
388   {
389     KEY_NAME_INFORMATION kni = {0};
390     ULONG rec_size;
391     if (NtQueryKey(KeyHandle, KeyNameInformation, &kni, sizeof(kni),
          &rec_size) == STATUS_SUCCESS)
392       if (kni.NameLength > 0)
393         wcsncat_s(object, kni.Name, MAX_STR_BUFFER-1);
394   }
395
396   // Determine value name
397   if (ValueName != NULL)
398     if (ValueName->Length > 0)
399       wcsncat_s(value_name, ValueName->Buffer, MAX_STR_BUFFER-1);
400
401   // Call trampoline or determine customized return value
402   if (hook_settings[7].preventExecution == FALSE)
403   {
404     status = NtSetValueKey_trampoline(KeyHandle, ValueName,
          TitleIndex, Type, Data, DataSize);
```

```
405     executed = 1;
406   }
407   else
408   {
409     status = (NTSTATUS)hook_settings[7].returnValue;
410   }
411
412   // Create log entry and return
413   SOCKET_ADDRESS_INFO sai = {0};
414   createLog(7, object, value_name, "", executed, 0, Type, 0, 0, 0,
        sai, status);
415   return status;
416 }
```

Listing 3.4: The `NtSetValueKey` callback function

We use the same arguments as for the API function and define the correct calling convention analogous to the previous callback example. In line 370, we check whether or not the trampoline function is used in order to provide the API's functionality. If the `preventExecution` member of the corresponding hook settings is set to `FALSE`, the trampoline function is called (line 404). Otherwise, the customized return value defined in the `returnValue` member of the hook settings is retrieved. Obviously, the implementation of this callback function is quite similar to the previous example. However, there are also some noteworthy differences.

In order to create or edit a registry key's value, two pieces of information are required: The name of the key and the name of the value. These two strings have to be passed to `createLog`. The value of `object` passed as the second argument specifies the registry key, the other one `value_name` passed as the third argument contains the name of the key's value.

While the value name can be simply retrieved by reading the member `Buffer` of the argument `ValueName`, the key name is not passed as a string to the `NtSetValueKey` API function. Instead, a handle associated with the key is provided. However, a handle is a process-specific numeric identifier, which means that we cannot pass the key handle to the log entry because the analysis tool will not be able to resolve the handle. Hence, the handle would provide no additional information. The solution to this problem is a resolution of the handle to the key's name within the callback function. For this purpose, we can use the native API function `NtQueryKey`[7] applied in line 391 to obtain the key's name. The interface is depicted in Table 3.3. It can provide various information about a registry key. In order to obtain the name, we have to initiate an empty `KEY_NAME_INFORMATION` structure and call the interface passing the handle as first argument and the created structure as third argument. Furthermore, we need to specify which kind of information we are looking for. Since we are interested in the key's name, `KeyNameInformation` is passed as the second argument. `KeyNameInformation` is defined as a named constant in an enumeration representing the value `3`. The output is stored in the `KEY_NAME_INFORMATION` structure which is defined as `kni`. Finally, the requested string can be found in its member `kni.Name`. Based on this information, we

[7] http://msdn.microsoft.com/en-us/library/ff567060(v=vs.85).aspx

Argument	Type	Comment
KeyHandle	HANDLE	
KeyInformationClass	KEY_INFORMATION_CLASS	
KeyInformation	PVOID	out, optional
Length	ULONG	
ResultLength	PULONG	out
Return Type	NTSTATUS	

Table 3.3.: The `NtQueryKey` Interface

are able to complete the required log information of this callback function, and to call `createLog` (line 414) and return.

The NtCreateProcessEx Callback Function

The `NtCreateProcessEx_callback` function is the last example of a callback function presented here. The `NtCreateProcessEx` API is used in Windows XP to create a new process. The Windows API provides only one interface for the process creation throughout all Windows versions: `CreateProcess`[8]. In contrast to this, the native API entails different interfaces for different system versions: `NtCreateProcess` for Windows 2000, `NtCreateProcessEx` for Windows XP, and `NtCreateUserProcess` for Windows Vista and 7. The problem with these functions is that they are not documented by Microsoft, i.e. there is no official resource we can refer to. As the test environment is developed and tested in a Windows XP operating system, we focus on `NtCreateProcessEx` in this thesis.

```
533 NTSTATUS NTAPI NtCreateProcessEx_callback(
534   OUT PHANDLE ProcessHandle,
535   IN ACCESS_MASK DesiredAccess,
536   IN POBJECT_ATTRIBUTES ObjectAttributes OPTIONAL,
537   IN HANDLE ParentProcess,
538   IN BOOLEAN InheritObjectTable,
539   IN HANDLE SectionHandle OPTIONAL,
540   IN HANDLE DebugPort OPTIONAL,
541   IN HANDLE ExceptionPort OPTIONAL,
542   IN HANDLE Unknown
543 )
544 {
545   // Initiate return value
546   NTSTATUS status;
547   wchar_t object[MAX_STR_BUFFER] = L""; object[MAX_STR_BUFFER-1] = '\0';
548   unsigned int wcRemaining = MAX_STR_BUFFER - 1;
549   unsigned long object_pid = 0;
550   unsigned char executed = 0;
551
552   if (hook_settings[10].preventExecution == FALSE)
553   {
```

[8] http://msdn.microsoft.com/en-us/library/ms682425(VS.85).aspx

```
554     // Call trampoline function
555     status = NtCreateProcessEx_trampoline(ProcessHandle, DesiredAccess,
            ObjectAttributes, ParentProcess, InheritObjectTable,
            SectionHandle, DebugPort, ExceptionPort, Unknown);
556     executed = 1;
557
558     if (*ProcessHandle != NULL)
559     {
560       // Determine object PID
561       PROCESS_BASIC_INFORMATION pbi = {0};
562       if (NtQueryInformationProcess(*ProcessHandle,
              ProcessBasicInformation, &pbi,
              sizeof(PROCESS_BASIC_INFORMATION), NULL) == STATUS_SUCCESS)
563         object_pid = pbi.UniqueProcessId;
564
565       // Determine object name
566       UNICODE_STRING *pImageFileName = 0;
567       int nSizeAlloc = sizeof(UNICODE_STRING) + sizeof(wchar_t) *
              MAX_STR_BUFFER, nDispose=0;
568       pImageFileName=(UNICODE_STRING *)malloc(nSizeAlloc);
569       if (NtQueryInformationProcess(*ProcessHandle,
              ProcessImageFileName, pImageFileName, nSizeAlloc, NULL) ==
              STATUS_SUCCESS)
570         if (pImageFileName->Length > 0)
571           wcsncat_s(object, pImageFileName->Buffer, wcRemaining);
572     }
573     else if (ObjectAttributes != NULL)
574     {
575       // Determine object name
576       if (ObjectAttributes->RootDirectory != NULL)
577       {
578         IO_STATUS_BLOCK ioStatus = {0};
579         FILE_NAME_INFORMATION fni = {0};
580         if (NtQueryInformationFile(ObjectAttributes->RootDirectory,
                &ioStatus, &fni, sizeof(fni), FileNameInformation) ==
                STATUS_SUCCESS)
581         {
582           if (fni.FileNameLength > 0)
583           {
584             wcsncat_s(object, fni.FileName, wcRemaining);
585             wcRemaining = wcRemaining - wcslen(object);
586             wcsncat_s(object, L"\\", wcRemaining--);
587           }
588         }
589       }
590       if (ObjectAttributes->ObjectName != NULL)
591         if (ObjectAttributes->ObjectName->Length > 0)
592           wcsncat_s(object, ObjectAttributes->ObjectName->Buffer,
                wcRemaining);
593     }
594   }
595   else
596   {
597     // Get customized return value
598     status = (NTSTATUS)hook_settings[10].returnValue;
599     // Determine object name
```

```
600       ...

620     }
621
622     // Create log entry
623     SOCKET_ADDRESS_INFO sai = {0};
624     createLog(10, object, L"", "", executed, DesiredAccess, object_pid,
            pid, 0, 0, sai, status);
625     return status;
626 }
```

Listing 3.5: The `NtCreateProcessEx` callback function

The `NtCreateProcessEx_callback` is very similar to the two previous examples. It checks whether or not the trampoline function has to be applied (line 552). If the trampoline function is called, a handle represents the created process. For the same reasons as in the previous example, we have to resolve the handle in order to obtain relevant information so that we can forward it to the analysis environment. In particular, we need to know the corresponding process identification number as well as the file path to the executable of the created process. Akin to the `NtQueryKey` interface, there is also an API function for resolving process handles. The wanted function is the `NtQueryInformationProcess`[9] interface, which is depicted in Table 3.4. It provides information about a specific process.

Argument	**Type**	**Comment**
ProcessHandle	HANDLE	
ProcessInformationClass	PROCESSINFOCLASS	
ProcessInformation	PVOID	out
ProcessInformationLength	ULONG	
ReturnLength	PULONG	out, optional
Return Type	NTSTATUS	

Table 3.4.: The `NtQueryInformationProcess` Interface

In order to retrieve the two required items, the callback calls this interface twice in line 562 and in line 569. The API call can provide quite different types of information based on the value of the second argument `ProcessInformationClass`. According to this argument, an appropriate data structure has to be created for each call and passed as the third argument `ProcessInformation`. The API function then writes the requested data to this data structure.

In line 562, the callback queries basic process information including the process identification number. A `PROCESS_BASIC_INFORMATION` data structure serves as container for the requested data. The received process handle, the `ProcessBasicInformation` enumerator constant, the created data structure as well as its size are passed as arguments. After the function has been successfully executed, the process identification number can be found in the structure's `UniqueProcessID` member.

[9] http://msdn.microsoft.com/en-us/library/ms687420(v=vs.85).aspx

In order to obtain the filepath of the executable associated with the observed process, the callback makes use of the same API function. The query is analogous to the query above, but uses two different arguments. Instead of `ProcessBasicInformation`, the API call receives the enumerator constant `ProcessImageFileName` argument. Therefore, the callback also has to use another data structure (`UNICODE_STRING`) as container for the requested information. Having called the interface with the mentioned arguments, the data structure entails the file path in the member `Buffer`.

If for some reason the process could not be created, the handle is set to `0` and cannot be used in order to retrieve any information concerning the specified executable. However, an analyst is not only interested in the successfully executed API calls, but also in those that caused errors. Therefore, the callback also implements another mechanism to provide the required data (lines 573 - 593). In this scenario, the callback uses the optional argument `ObjectAttributes` analogous to `NtCreateFile_callback`, described in the first callback example, in order to determine the executable's file path.

Having gathered all relevant data, the callback can finally create a log entry and return the appropriate value. (lines 624, 625)

3.4.5. Detection Prevention

As we have detailed in the previous sections, the main purpose of the hook library is to install hooks for the purpose of monitoring functionality. However, there is also another aspect when using the hook library: Various malware samples implement methods inspecting the system environment in order to detect anti-malware software or analysis software. If such a piece of software is detected, different scenarios can occur. For example, the malware sample could kill the corresponding process or delete the detected files. Another scenario is that the malware might behave differently. Therefore, we must try to avoid the detection of our analysis environment.

Possible techniques applied by malware in order to examine the system are to list all running processes or to scan the file system. Often, certain API functions are applied in order to implement the required detection techniques. In PyBox, we consider two different methods. The first method is to use the API function `CreateToolhelp32Snapshot` in order to create a snapshot containing a list of all running processes and to parse this list using the API functions `Process32First` and `Process32Next`. The second method considered is to use the API functions `FindFirstFile` and `FindNextFile` in order to scan the filesystem.

Since we want to prevent the detection of our analysis environment by such methods, we have to hide the analysis tool's process from being scanned and to avoid the inspection of the PyBox folder in the filesystem. Once again, we can use the already implemented hooking functionality in order to intercept these API functions and hide the corresponding files and processes. Analogous to the other hooks, we have to implement further callback and trampoline functions. As we have already described in detail how to implement these functions, we do not list their exact implementation here. Instead, we focus on their functional principle which is depicted in Figure 3.3.

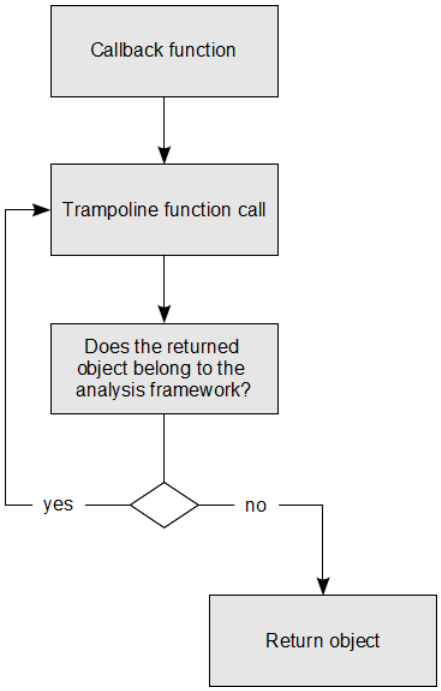

Figure 3.3.: PyBox detection prevention procedure

Since the API functions are used to iterate through the list of items to be scanned, the callback implementations are relatively simple. At first, the function call to the hooked API function is redirected to the callback function, which immediately calls the trampoline function. The trampoline function returns an object. Based on this object, we can tell whether it belongs to the analysis framework or not. In case of `Process32First` or `Process32Next`, the object is compared to the process identification number of the analysis tool. In case of `FindNextFile`, the filename of the returned object is compared to the folder name of PyBox and the hook library's filename. If the object belongs to the analysis framework, the trampoline function will be called again until it returns an object which does not belong to the analysis framework or until there is no more object left in the list to be queried. This object is finally returned.

3.4.6. Installation

Thus far, we have described how function hooking and the corresponding callback and trampoline functions are implemented in `pbMonitor.dll` as well as how to hide the hooking and monitoring functionality. However, even if the hook library has been injected into the target process's address space, the hooks are not yet installed. We first have to notify the hook library that it has to obtain the hook settings data, and install all specified and required hooks after it has been loaded. For this purpose, we use the library's main entry point `DllMain`[10] which is outlined in Listing 3.6.

[10]http://msdn.microsoft.com/en-us/library/ms682596(v=vs.85).aspx

```
82 BOOL WINAPI DllMain (HANDLE hDll, DWORD dwReason, LPVOID lpReserved)
83 {
84   switch (dwReason)
85   {
86     case DLL_PROCESS_ATTACH:
87       readSettings();
88       initializeLog();
89       initializeNativeApiFunctions();
90       setCallbackAddresses();
91       setTrampolineAddresses();
92       installHooks();
93       setInjectionTime();
94       break;
95
96     case DLL_PROCESS_DETACH:
97       break;
98
99     case DLL_THREAD_ATTACH:
100      break;
101
102    case DLL_THREAD_DETACH:
103      break;
104  }
105  return TRUE;
106 }
```

Listing 3.6: DLL entry point

There are four different situations in which `DllMain` is called automatically. The argument `dwReason` specifies the respective situation. Therefore, this argument can be used in order to perform appropriate operations in case any of the four situations occurs. In the following, the four different scenarios are outlined.

1. `DLL_PROCESS_ATTACH`: The DLL is loaded by a process.
2. `DLL_PROCESS_DETACH`: The DLL is unloaded by a process.
3. `DLL_THREAD_ATTACH`: The current process creates a new thread.
4. `DLL_THREAD_DETACH`: A thread exits.

We can use the `DLL_PROCESS_ATTACH` scenario in order to perform the required installation steps rendering the hooks to become effective. For this purpose, `DllMain` executes several functions. At first, it reads all specified settings (line 87). The settings include the information, particularly which API functions to hook and to monitor. Subsequently, the logging environment has to be set up (line 88) allowing the callback functions to create log entries. The logging process is described in more detail in Section 3.6. In line 90, the native API functions required by the hook library are located in the process's virtual memory. This is necessary in order to be able to call them. Subsequently, the memory addresses of all callback and trampoline functions are written to the hook settings (lines 91, 92). This has to be done during runtime as these addresses required for hooking the corresponding functions cannot be determined beforehand. Finally, all specified hooks are installed (line 93) and the initial timestamp is set (line 94). Timestamps are required

in order to provide the log entries with the time interval that has passed since the hook library has been injected.

3.5. Analysis Tool

Apart from the hook library, which resides inside the target process and sends information about the observed activities, we need to provide an analysis environment that initiates the required monitoring and information processing functionality, and produces the output. Within PyBox, these services are provided by the PyBox analysis tool. It has various tasks that we already rudimentarily described in Section 3.2. In this section, we discuss these tasks in more detail and outline their implementation.

We start by providing an overview of the PyBox analysis tool and describe the functionality of its modules as well as their application. Subsequently, we describe the configuration management of PyBox. Then, we take a closer look at the creation of the target process and how the hook library is placed inside this process. Finally, we explain the processing of the logged information and the generation of the final XML-based report.

3.5.1. Analysis Framework Overview

In contrast to the hook library, the PyBox analysis tool has to be controlled by the user. Therefore, we have to provide means of user interaction enabling an analyst to configure the analysis environment according to one's analysis goals. As soon as PyBox is configured and executed, the target executable's process has to be created, and the hook library needs to be injected and installed according to the configured settings. While the hook library sends log entries from the hooked target process, the analysis tool receives the logged information. After the monitored process is terminated, the analysis tool processes the required data. Finally, the result of this process is written into a machine-readable report, which can be put to further use. All-in-all, the analysis framework must provide the following functionality:

- configuration management;
- process creation;
- library injection;
- log data processing;
- report generation.

The PyBox analysis framework is implemented using the *Python programming language*.[11] According to Seitz [Sei09], Pyhton is one of the most popular programming languages when it comes to hacking and reverse engineering. Another reason that makes

[11] http://www.python.org/

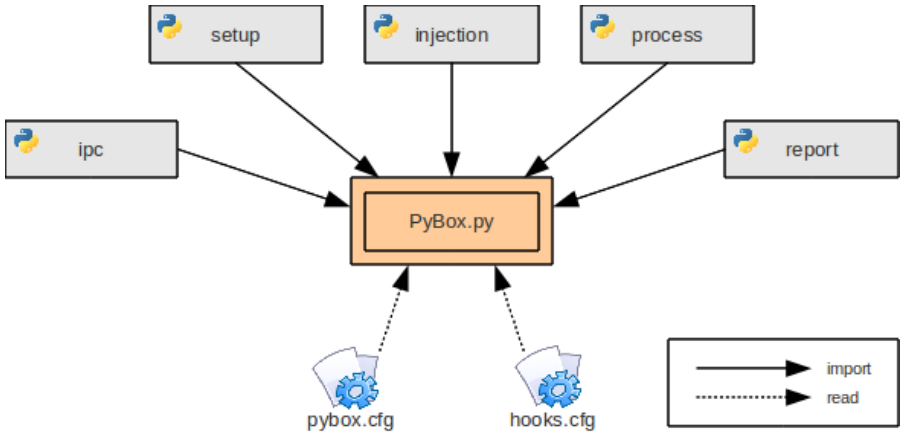

Figure 3.4.: PyBox analysis tool layout

it appealing in the context of this thesis is its simplicity with regards to writing and understanding its code, which makes it suitable for teaching purposes. Furthermore, it provides low-level support and enables us to produce C compatible code. The latter is important since we create and use C data types in order to communicate and interact with the hook library or Windows API functions in general.

In order to fulfil these requirements, PyBox includes various packages, modules and configuration files. Figure 3.4 illustrates all components and their relationships.

In the upper part of the Figure 3.4, the five packages belonging to the PyBox analysis tool are depicted. The package `setup` is responsible for reading the provided configuration files extracting their information. Furthermore, it converts this information into objects that can be used by the analysis tool. The package `process` allows to execute process-related operations such as the creation of the target process. Additionally, it provides required process-specific information. The DLL injection method is implemented in the package `injection`. The final report generation operations are made available by the package `report`. More precisely, it offers the functionality to process the data received from the monitored process and turn them into a XML-based report. The communication and interaction with the target process is implemented in the package `ipc`. Basically, it sends the acquired settings to the observed process and receives the corresponding log messages. This package is not part of this chapter. We describe it in detail in Chapter 3.6.

In the lower part of Figure 3.4, we display the two configuration files `pybox.cfg` and `hooks.cfg`. Using these files, an analyst is able to configure PyBox. Thus, he or she can for example determine the used hook library and the hooks to be installed.

All the described packages and configuration files are applied by the Python module `PyBox.py` combining the various functionalities. It first reads all settings defined in the configuration files `pybox.cfg` and `hooks.cfg`. Then, the module creates the target process using the package `process` and injects the hook library installing all specified hooks via the package `injection`. During the execution of the target process, `PyBox.py` receives all sent log information using the package `ipc`. After the observed process has

been terminated, all data are processed and captured in an XML file by applying the functionality of package `report`.

3.5.2. Setup and Configuration Files

As mentioned in the last section, there are two configuration files that enable the analyst to adjust the PyBox configuration. The files are separated from each other because they serve different purposes. While `pybox.cfg` provides basic information required by the analysis framework, `hooks.cfg` defines settings concerning the implemented hooks to be loaded into the target process. In the following, both the layout and the configuration of the two files are outlined.

The Configuration File pybox.cfg

The file `pybox.cfg` contains five pieces of information required by the analysis framework. PyBox cannot be executed without these information. The layout of `pybox.cfg` is shown in Listing 3.7.

```
# Path to target executable of this analysis
EXE_TARGET = "C:\path_to_binary\binary.exe"

# Path to the hook library
LIB_PBMONITOR = "C:\path_to_hook_library\pbMonitor.dll"

# Path to config file containing information about all hooks
CFG_HOOKS = "C:\path_to_hook_settings\hooks.cfg"

# Output folder to which the report will be written
LOG_FOLDER = "C:\path_to_output_folder_of_logs"

# Timeout interval after which the observed executable is terminated (in
    seconds)
TIMEOUT = "120"
```

Listing 3.7: The configuration file `pybox.cfg`

The target executable that has to be monitored is specified via `EXE_TARGET` while the value of `LIB_PBMONITOR` has to be set to the path of the hook library, which is called `pbMonitor.dll` in this thesis. This parameter is required by the analysis tool for the ability to inject the specified hook library into the target process. In `CFG_HOOKS`, we have to specify the file path to the configuration file `hooks.cfg` defining the settings concerning all API functions that have to be hooked. We describe this file in more detail below. The `LOG_FOLDER` has to be set to the folder into which the analysts wants PyBox to export its analysis reports.

These information are read using the package `settings` and stored into a Python `dictionary` object with the keys `LIB_PBMONITOR`, `CFG_HOOKS`, and `LOG_FOLDER`. Thus, all required information is obtained by using the dictionary with the corresponding key.

The Configuration File hooks.cfg

In contrast to the `pybox.cfg` configuration file, `hooks.cfg` does not have any effect on the execution of the PyBox analysis tool. Instead, it influences the execution of the hook library inside the target process. More specifically, it affects the installation process of the hook library, particularly which hooks to install and, thus, which API functions to monitor. The structure of this configuration file is outlined in Listing 3.8.

```
# File Management APIs
DLL=ntdll.dll,API=NtCreateFile,Intercept=1,PreventExecution=0,ReturnValue=0
DLL=ntdll.dll,API=NtOpenFile,Intercept=1,PreventExecution=0,ReturnValue=0
DLL=ntdll.dll,API=NtReadFile,Intercept=1,PreventExecution=0,ReturnValue=0
DLL=ntdll.dll,API=NtWriteFile,Intercept=1,PreventExecution=0,ReturnValue=0
DLL=ntdll.dll,API=NtDeleteFile,Intercept=1,PreventExecution=0,ReturnValue=0

# Registry Management APIs
DLL=ntdll.dll,API=NtCreateKey,Intercept=1,PreventExecution=0,ReturnValue=0
DLL=ntdll.dll,API=NtOpenKey,Intercept=1,PreventExecution=0,ReturnValue=0
DLL=ntdll.dll,API=NtSetValueKey,Intercept=1,PreventExecution=0,ReturnValue=0
DLL=ntdll.dll,API=NtQueryValueKey,Intercept=1,PreventExecution=0,ReturnValue=0

# Process Management APIs
DLL=ntdll.dll,API=NtCreateProcess,Intercept=1,PreventExecution=0,ReturnValue=0
DLL=ntdll.dll,API=NtCreateProcessEx,Intercept=1,PreventExecution=0,ReturnValue=0
DLL=ntdll.dll,API=NtOpenProcess,Intercept=1,PreventExecution=0,ReturnValue=0
DLL=ntdll.dll,API=NtTerminateProcess,Intercept=1,PreventExecution=0,ReturnValue=0
DLL=ntdll.dll,API=NtCreateSection,Intercept=1,PreventExecution=0,ReturnValue=0
DLL=ntdll.dll,API=NtCreateThread,Intercept=1,PreventExecution=0,ReturnValue=0

# Network/Socket APIs
DLL=wsock32.dll,API=connect,Intercept=1,PreventExecution=0,ReturnValue=0
DLL=wsock32.dll,API=send,Intercept=1,PreventExecution=0,ReturnValue=0
DLL=wsock32.dll,API=recv,Intercept=1,PreventExecution=0,ReturnValue=0
DLL=ws2_32.dll,API=getHostByName,Intercept=1,PreventExecution=0,ReturnValue=0
DLL=ws2_32.dll,API=connect,Intercept=1,PreventExecution=0,ReturnValue=0
DLL=ws2_32.dll,API=send,Intercept=1,PreventExecution=0,ReturnValue=0
DLL=ws2_32.dll,API=sendto,Intercept=1,PreventExecution=0,ReturnValue=0
DLL=ws2_32.dll,API=recv,Intercept=1,PreventExecution=0,ReturnValue=0
DLL=ws2_32.dll,API=recvfrom,Intercept=1,PreventExecution=0,ReturnValue=0
DLL=ws2_32.dll,API=WSAConnect,Intercept=1,PreventExecution=0,ReturnValue=0
DLL=ws2_32.dll,API=WSASend,Intercept=1,PreventExecution=0,ReturnValue=0
DLL=ws2_32.dll,API=WSARecv,Intercept=1,PreventExecution=0,ReturnValue=0
```

Listing 3.8: The configuration file `hooks.cfg`

Each non-empty line and not starting with a `#` represents another hook. For each hook five settings have to be defined. The API to be hooked is identified by its name and the name of the module it can be found in. The setting `DLL` specifies the module name and the setting `API` the name of the API function to be hooked. If the analyst wants to hook the API, he or she has to set the setting `Intercept` to `1`. Otherwise, it has to be set to `0`. In case, the analyst wants to prevent the API from being executed and to return a customized value he or she can set the settings `PreventExecution` to `1` and specify a customized return value in the setting `ReturnValue`. Since we almost

exclusively hook native API functions, the return value specified has to be a value of type `NTSTATUS`. All possible values can be looked up in the Microsoft MSDN library.[12]

For all hooks defined in this configuration file, there have to be callback and trampoline implementations in the hook library. The API hooks defined in Listing 3.8 are all available hooks at the time of writing this thesis. Moreover, all functions that can be monitored by PyBox are listed in Appendix A.

During the start of the execution of `PyBox.py`, all hooks are read and each one is mapped to a C structure `MMF_SETTINGS_ITEM`. All hooks are stored as an array of such structures. This hook settings array is present both in the PyBox main application and the hook library. Thus, each hook has a specific identification number, and the module name and API name are mapped to its identification number. Once, the array is created out of the configuration file, the hook library can obtain the array and install all specified hooks.

3.5.3. Target Process Creation and DLL Injection

In the previous section, we described how the configuration files are read, and how the configured settings are used, and how they are made available. The next step is to start the target process with the installed hook library and monitor its behavior. To do this, yet again three steps are necessary. First, we have to create the process but at the same time make sure that no instruction is executed until the installation process of the hook library is completed. Otherwise, we would risk that API calls are executed without being monitored. Therefore, the second step is to inject the hook library into the suspended process followed by the third step: resuming the process. In the following, we take a closer look at each of these steps.

Step 1: Create the Target Process

In order to start the target process, we first have to know the path to the executable file which we want to monitor. This path is specified in the configuration file `pybox.cfg` and therefore stored in a settings object. Having gathered all the relevant information, we are able to create a process for the target executable. This is done by applying the Windows API `CreateProcessA`[13]. The implementation of the process creation used in PyBox is outlined in Listing 3.9.

```
4  class process_manager:
5    ...

13   def pcreate(self, path_to_exe):
14     self.exe        = path_to_exe
15
16     # Initialize parameters for CreateProcessA
17     creation_flags  = CREATE_SUSPENDED
```

[12] http://msdn.microsoft.com/en-us/library/cc704588(v=PROT.10).aspx
[13] http://msdn.microsoft.com/en-us/library/ms682425(v=vs.85).aspx

```
18      startupinfo      = STARTUPINFO()
19      startupinfo.cb   = sizeof(startupinfo)
20      process_information = PROCESS_INFORMATION()
21
22      # Create Process in suspended mode
23      if kernel32.CreateProcessA(self.exe,
24        None,
25        None,
26        None,
27        None,
28        creation_flags,
29        None,
30        None,
31        byref(startupinfo),
32        byref(process_information)):
33      self.pid = process_information.dwProcessId
34      self.tid = process_information.dwThreadId
35      self.h_process = process_information.hProcess
36      self.h_thread = process_information.hThread
37      print "Process successfully launched with PID %d." % self.pid
38      return True
39    else:
40      print "Process could not be launched!"
41      raise WinError()
42      return False
```

Listing 3.9: Target process creation

The process creation is implemented in the class `process_manager` in the package `process`. Like most Windows API functions, `CreateProcessA` requires the input of several arguments displayed in 3.5. However, many of them are optional and we do not need to specify all of them. Therefore, the other ones do not concern us in this thesis and will not be described here. For more information on this API call, refer to the Microsoft Developer Network (MSDN) website. The arguments we need are `lpApplicationName`, `dwCreationFlags`, `lpStartupInfo`, and `lpProcessInformation`. The first argument receives the file path to the executable file whereas the last two are pointers to structures of type `STARTUPINFO` and `PROCESS_INFORMATION` which have to be initialized (lines 18 - 20). The `STARTUPINFO` structure specifies how the process has to be started. In the `PROCESS_INFORMATION` structure important process information such as the process handle are stored. The argument `dwCreationFlags` plays an important role in our scenario. This parameter can be assigned certain flags that dictate how a process is created. Here, we use the flag `CREATE_SUSPENDED` (line 17). As mentioned before, we have to ensure that the process is created, but not executed until the hook library is injected and all hooks are installed. For this purpose, we use the `CREATE_SUSPENDED` flag. This way, the process is created, but its main thread is not yet started.

Step 2: Inject the Hook Library

Having created the target process, we possess all relevant process information. Hence, we are ready to inject the hook library into its memory and install all hooks. We have

Argument	Type	Comment
lpApplicationName	LPCSTR	optional
lpCommandLine	LPSTR	optional
lpProcessAttributes	LPSECURITY_ATTRIBUTES	optional
lpThreadAttributes	LPSECURITY_ATTRIBUTES	optional
bInheritHandle	BOOL	
dwCreationFlags	DWORD	
lpEnvironment	LPVOID	optional
lpCurrentDirectory	LPCTSTR	optional
lpStartupInfo	LPSTARTUPINFO	
lpProcessInformation	LPPROCESS_INFORMATION	
Return Type	BOOL	

Table 3.5.: The `CreateProcessA` Interface

already described the various DLL injection methods in Section 2.3.3. In this thesis, we use the injection via remote thread.

In order to load the hook library into the target process's memory, we have to make it call the Windows API `LoadLibrary`. This interface receives only one argument which is the path to the library to be loaded. Therefore, we need to execute the `LoadLibraryA`[14] function and a pointer to the string that contains the path to the hook library. The implementation of the DLL injection method is depicted in Listing 3.10.

First, we have to allocate some space (line 6) in the remote process's memory in order to store the path to the DLL (line 15) to be loaded. The API function `VirtualAllocEx` used to allocate the required memory space returns a pointer to the address of the string in which the library path is stored. Then, we have to find out the memory address of `LoadLibraryA`. By using the API functions `GetModuleHandle` (line 18) and `GetProcAddress` (line 24), we actually search the memory address of `LoadLibraryA` in the address space of the analysis tool and not inside the remote process. Here, we benefit from the fact that this interface is present in all Windows processes and is located at the exact same location in all of them. Therefore, we can use the acquired address for the remote process as well. Finally, we have to create a new thread inside the remote process in order to call the `LoadLibraryA` function. To do this, we have to use the API `CreateRemoteThread`[15] (line 32) with the first parameter `hProcess` being the handle to the created process, the third argument `lpStartAddress` being the address of the `LoadLibraryA` function, and the fourth argument `lpParameter` being the pointer to the file path of the hook library. If this procedure is completed successfully, the hook library has been loaded into the remote process's address space. At this point, the DLL's main entry point is executed installing all specified hooks as described in Section 3.4.6.

[14] http://msdn.microsoft.com/en-us/library/ms684175(v=vs.85).aspx
[15] http://msdn.microsoft.com/en-us/library/ms682437(v=vs.85).aspx

```
3 def inject_dll(h_process, dll_path):
4   # Allocate some space for the DLL path
5   dll_len  = len(dll_path)
6   arg_address = kernel32.VirtualAllocEx( h_process, 0, dll_len,
        VIRTUAL_MEM, PAGE_READWRITE)
7   ...

13  # Write the DLL path into the allocated space
14  written = INT(0)
15  kernel32.WriteProcessMemory(h_process, arg_address, dll_path,
        dll_len, byref(written))
16
17  # Resolve the address of LoadLibraryA
18  h_kernel32 = kernel32.GetModuleHandleA("kernel32.dll")
19  ...

24  h_loadlib  = kernel32.GetProcAddress(h_kernel32,"LoadLibraryA")
25  ...

30  # Create remote thread
31  thread_id = DWORD(0)
32  h_remoteThread = kernel32.CreateRemoteThread(h_process, None, 0,
        h_loadlib, arg_address, 0, byref(thread_id))
33  ...

42  print "Injection successful."
43  return h_remoteThread
```

Listing 3.10: DLL injection

Step 3: Resume the Target Process

After injecting the hook library, we have to wait for the created thread to finish the hook installation process and to resume the suspended main thread subsequently. We use the API function `WaitForSingleObject`[16] in order to wait for the thread to terminate. The interface only receives one argument which is the handle to the remote thread. After that, we can resume the main thread by calling the API function `ResumeThread`[17] with the thread handle to the main thread of the remote process. This handle is provided in the `PROCESS_INFORMATION` structure of the `CreateRemoteProcess` instruction which we have described in Step 1. Having resumed the thread, the target process executes as usual while all hooked API calls are logged until the process terminates. Therefore, the analysis tool has to wait for the remote process to terminate in order to proceed. This is done by usage of `WaitForSingleObject` and the handle to the remote process which is also stored in the `PROCESS_INFORMATION` structure. In case further processes have been injected by the hook library, we have to wait for them to terminate as well.

[16] http://msdn.microsoft.com/en-us/library/ms687032(v=vs.85).aspx
[17] http://msdn.microsoft.com/en-us/library/ms685086(VS.85).aspx

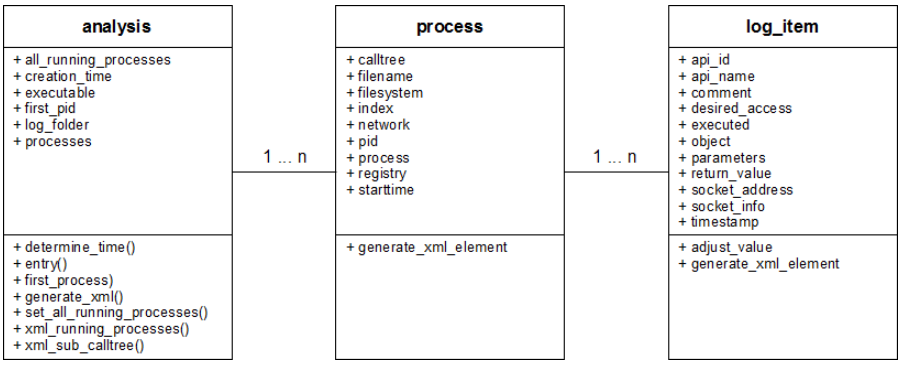

Figure 3.5.: PyBox data analysis classes

3.5.4. XML Report

In the previous section, we have described how the monitored process is created and injected. As soon as all observed process have terminated, the logged data can be retrieved. The data is stored as a Python list of C data structures. How this list is created is part of Section 3.6. The goal of this section is to describe, how we store this data in a form that enables the flexible creation of a structured, understandable XML document and how to create the XML file. For this reason, this section is divided into two parts: the processing of the log data and the generation of the final XML document from this data.

Data Processing

Although we could use the logged data directly and simply display them, we choose another solution. The analysis report is supposed to provide structured information as well as a structured overview of the activities of the observed executable. To do this, we have to distinguish the various log entries into different categories of the monitored APIs: file management, registry management, process management, and networking.

In order to structure the data, we use three different classes: `analysis`, `process`, and `log_item`. Their attributes, methods and relationships are displayed in Figure 3.5. The class `analysis` contains the main attributes of the report, such as the executable file observed by PyBox, the log file as well as the report's creation time. It also contains a dictionary object which includes all observed processes. We can access these processes by using the dictionary with the respective process identification number. The observed processes are represented by the second class `process`. It contains all process-relevant details as attributes such as the process identification number and the image name of the process. Each process object also has four different lists which represent the four different categories of monitored API calls mentioned above. These lists contain the actual log data that have been monitored by the hook library. The log data are represented by the class `log_item`. For each entry in the log, a `log_item` object is created which includes all information associated with the monitored entry.

The class `analysis` provides a method `entry` which allows us to insert log data entries

into the created report. The analysis object then checks the process identification number and function identification number of the log data. The function identification number is the position in the hook settings array which we have described in Section 3.5.2. After the `log_item` object is created, it is inserted into the corresponding `process` object by using the process identification number and into the corresponding category list object by using the function identification number.

XML Generation

Once, the data is processed and all information is represented by entities of the classes described above, we can generate the XML report from this data. Python provides three different alternatives in order to handle XML files: *DOM*, *SAX*, and *ElementTree*. The three alternatives are briefly described in the following.

- **DOM**
  DOM is short for *Document Object Model*. The DOM interface[18] has been standardized by the *World Wide Web Consortium* and provides functionality to parse and edit XML files. For this purpose, DOM loads the entire file into memory and generates a tree structure out of the read data.
- **SAX**
  In contrast to DOM, the *Simple API for XML*[19] (SAX), which is also a standard does not load the entire file into memory, but parses it sequentially. Therefore, it needs much less memory but is less flexible at the same time.
- **ElementTree**
  While the first two alternatives are standards, ElementTree is a Python-specific feature and not available in other languages. It is integrated since Python version 2.5 and offers the functionality to write and parse XML files.

We prefer to stay flexible in order to create a well-structured report. SAX, however, is relatively inflexible due to its sequential approach and therefore rather represents an alternative to read XML files. Hence, we have to choose between DOM and ElementTree in order to implement the XML file generation. Both alternatives are similar and equally suited for the required purpose. For this reason, DOM is chosen here for personal preference.

Since DOM uses a hierarchic tree structure, we can use the classes of the previous section and implement a method, which creats an individual subtree, for each. Thus, we only have to combine the various subtrees to the final XML document. This makes the creation process of the report simple and comprehensive. Hence, both the class `process` and the class `log_item` must provide a method `generate_xml_element`, which creates the respective XML subtree. In order to create the entire XML report, we have to call the `generate_xml` method of the `analysis` class instance, which is depicted in Listing 3.11.

[18] http://www.w3.org/DOM/
[19] http://www.saxproject.org/

```
306 class analysis:
307   ...

391   def generate_xml(self):
392     ...

400     # Create XML document
401     root = dom.Document()
402     tag_analysis = dom.Element("PyBox-Analysis")
403     tag_analysis.setAttribute("Executable", self.executable)
404     tag_analysis.setAttribute("LogFile", log_file)
405     tag_analysis.setAttribute("Time", xml_time_string)
406     tag_analysis.setAttribute("MD5", self.file_info["md5"])
407     tag_analysis.setAttribute("SHA1", self.file_info["sha1"])
408     tag_analysis.setAttribute("FileSize", self.file_info["size"])
409
410     # Insert call tree
411     tag_calltree = dom.Element("calltree")
412     tag_calltree.appendChild(self.xml_sub_calltree(self.first_pid))
413     tag_analysis.appendChild(tag_calltree)
414
415     # Insert all monitored processes including all monitored activity
416     tag_processes = dom.Element("processes")
417     for p in self.processes.values():
418       tag_processes.appendChild(p.generate_xml_element())
419     tag_analysis.appendChild(tag_processes)
420
421     # Insert all running processes
422     tag_analysis.appendChild(self.xml_running_processes())
423     root.appendChild(tag_analysis)
424
425     # Write XML tree to log file
426     with open(log_file, "w") as file:
427       root.writexml(file, "", "\t", "\n", "utf-8")
```

Listing 3.11: XML document generation

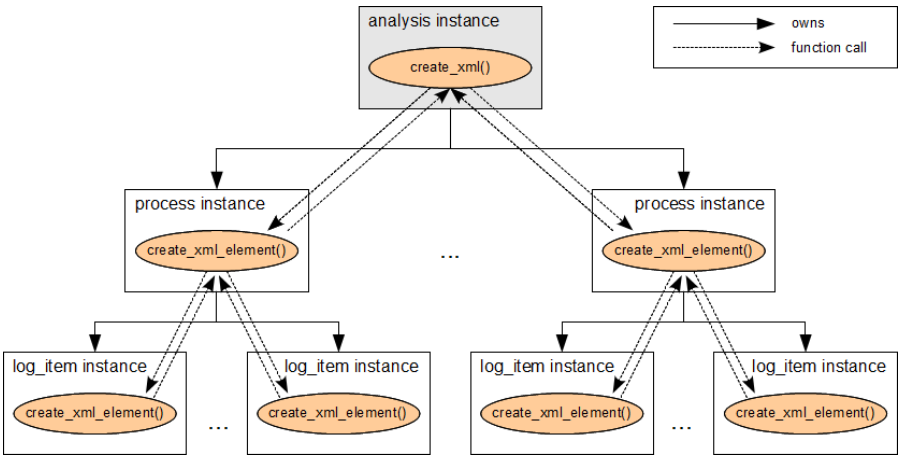

Figure 3.6.: PyBox XML generation procedure

The `create_xml` method of the class `analysis` creates the root tag `<PyBox-Analysis>` (line 402) and inserts its subtags `<calltree>` (line 413), `<processes>` (line 419) as well as `<running_processes>` (line 422). For each process instance obtained by the analysis instance, the `generate_xml_element` method is called and returns its XML subtree. Each process in turn iterates through the `log_item` instances in the lists of every category calling their `generate_xml_element` methods. This way, the entire XML tree is created which can be finally written to a file (line 427). The XML report generation process is depicted simplified in Figure 3.6 regarding the different class instances. The resulting XML structure is outlined in Listing 3.12.

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <PyBox-Analysis Executable="target.exe" LogFile="..." ...>
3   <calltree>
4     ...
5   </calltree>
6   <processes>
7     <process>
8       <filesystem_section>
9         ...log item tags...
10      </filesystem_section>
11      <registry_section>
12        ...log item tags...
13      </registry_section>
14        ...log item tags...
15      <process_section>
16       ...
17      </process_section>
18    </process>
19    ...
20  </processes>
21  <running_processes>
22    ...
23  </running_processes>
24 </PyBox-Analysis>
```

Listing 3.12: XML report structure

3.6. Inter-Process Communication

Thus far, we have described the hooking functionality of the hook library as well as the configuration and data processing functionality of the analysis tool. Obviously, for every analysis at least two processes are executed: the analysis tool as well as the observed target process. As mentioned before, these processes have to communicate and interact with each other. On the one hand, the hook library has to transfer several settings data to the hook libraries inside the target processes specifying the path to the hook library, the process identification number of the analysis tool which has to be hidden, and the hooks to be installed. On the other hand, the hook library has to send all created log entries to the analysis tool in order notify the analysis environment of the occurred events.

However, as described in Section 2.2.1, processes cannot simply access the memory of another process. This is an important security feature of modern operating systems. Instead, we have to use functionality provided by the operating system. For this purpose, inter-process communication mechanisms are used.

Inter-process communication (IPC) comprises all methods which enable communication and data sharing between different threads, processes, and applications. In this section, we take a closer look at IPC. At first, we outline several methods of IPC in Windows and their purposes and derive suitable alternatives for our scenario. Subsequently, we describe the method implemented in PyBox and how it is applied.

3.6.1. IPC Methods

The need to transfer data between processes is not rare. There are various scenarios in which the usage of IPC methods can be applied. Consequently, many methods exist based on different conditions and different needs. When choosing an IPC method, one must be aware of the purpose it has to serve and the conditions it has to meet. This raises some questions:

- Does the communication concern processes on different computers, and if so, do they use the same operating system?
- What actions are performed on the data?
- Do the processes implicitly know the other processes?
- Does performance play a key role?

Based on the answers to these questions, different methods can be applied. Typically, these methods use a client-server model. One process acts as a client requesting a particular service which is offered by another process, the server process. The service can either be a function or data. In the following, several different IPC methods available on Windows operating systems (cf. MSDN [Netb]) are outlined.

- **Dynamic Data Exchange** (DDE) is a protocol used for the communication between applications. The communication is realized via transmission of messages and shared memory in order to share data. DDE can be used for both one-time and continuous communication. It also provides a mechanism to signal updates to the other processes. Yet, it is an older technology and not efficient.
- **Data Copy** can be used to send data from a sender to a receiver. This method is realized using `WM_COPYDATA` messages. In order to apply this method, pointers must not be used. This method is very quick but rather used for one-way communication.
- **Pipes** are basically shared memory used for the communication between a parent process and a child process by redirecting the standard input and output. There are two types of pipes: *anonymous pipes* and *named pipes*. Anonymous pipes are used for the communication between processes on a single computer whereas named pipes are used to transfer data between different computers. This method is a relatively efficient IPC method.

- **Windows Sockets** provide a protocol-independent interface which is used for the communication between applications. It is commonly used for the communication over networks.
- **Remote Procedure Calls** (RPC) allow to call functions of another process and can be used both on a single computer or on networks. It also supports data conversion and different operating systems.
- **File Mapping** is usually a method which loads the content of a file into memory. Communication between different processes can be realized either through the contents of a file or via *named shared memory*. Named shared memory uses the system swapping file. Different processes can identify the memory by its name and access it. At the same time, one has to guarantee the consistency of the data by the means of synchronization. Although this method is very efficient and performant, it is also restricted to the use on a single computer.

For further information about IPC on Windows operating systems, please refer to the Microsoft System Developer Network [Netb].

3.6.2. IPC in PyBox

In PyBox, we require a preferably performant IPC method. Therefore, the added functionality in the target process should have as less of an impact as possible. Furthermore, we only have to communicate between processes which are running on the PyBox's virtual machine. Therefore, the IPC method not necessarily has to support the communication between two or more computers. For these reasons, file mapping through named shared memory is the method of choice in PyBox.

As mentioned in the previous section, IPC uses a client-server model. In PyBox, the analysis tool acts as file mapping server whereas the hook library represents the client. Therefore, the analysis tool must create the file mapping and provide the required data. The hook library's installation process reads the information contained in the file mapping and processes the received data. In order to create a file mapping three steps are necessary:

1. Creation of the file mapping object
2. Mapping of the contents into memory
3. Copying the data to the memory location

The Windows API provides the required API functions `CreateFileMapping`[20] for Step 1 and `MapViewOfFile`[21] for Step 2. The implementation used in the analysis tool is located in the module `mmf_services` of the package `ipc` and outlined in Listing 3.13. `CreateFileMapping` is called with the argument `INVALID_HANDLE_VALUE` (line 15). This causes that instead of a regular file in the filesystem the windows paging file is used to create shared memory. The `MapViewOfFile` function returns a pointer to the buffer of

[20] http://msdn.microsoft.com/en-us/library/aa366537(v=vs.85).aspx
[21] http://msdn.microsoft.com/en-us/library/aa366761(v=vs.85).aspx

the shared memory (line 22). This pointer is used to copy the required data to the shared memory in Step 3 (line 30, 32).

```
4  class file_mapping:
5    ...

13   def create(self, mmf_data, isString=false)
14     self.data = mmf_data
15     self.__hMapObject =
           windll.kernel32.CreateFileMappingA(INVALID_HANDLE_VALUE, None,
           PAGE_READWRITE, 0, self.size, self.name)
16     ...

22     self.pBuf = windll.kernel32.MapViewOfFile(self.__hMapObject,
           FILE_MAP_ALL_ACCESS, 0, 0, self.size)
23     ...

28     memcpy = cdll.msvcrt.memcpy
29     if not isString:
30       memcpy(self.pBuf, byref(self.data), sizeof(self.data))
31     else:
32       memcpy(self.pBuf, self.data, len(self.data))
33     return True
```

Listing 3.13: File mapping creation

The steps of accessing the file mapping in the remote process are quite similar. At first, an appropriate data structure is created to which the transmitted data can be copied. Subsequently, the API function `MapViewOfFile` is used to load the shared memory into the process's address space. Finally, the data can be copied to the created data structure.

Both the analysis tool and the hook library are provided with the name of the file mapping. This way, the analysis process as well as the monitored processes can access the shared memory. We use three different file mappings in order to communicate three different types of data: the common settings, the hook settings, and the log entries. The file mappings, the included data and the purposes of the data are described as follows.

The Common Settings File Mapping

The common settings consist of a C data structure which contains two members. The members are depicted in Table 3.6. The first member `PyBoxPid` contains the process identification number of the analysis tool. This information is required by the hook library in order to hide the analysis tool and prevent its detection. As described in Section 3.4.5, some malware samples contain functionality to detect software which might analyze them or prevent their execution. Therefore, we have to install additional hooks which hook API functions that can be used to inspect all running processes. In order to avoid the detection of the analysis framework, we have implemented a callback which prevents this API from listing our analysis tool process. In order to check if the next process to be listed is the analysis tool's process we have to compare the process identification numbers. For this purpose, this member is required. The second member contained in the common settings file mapping is `PbMonitorPath`. It specifies the file

Member	Type
PyBoxPid	DWORD
PbMonitorPath	char[256]

Table 3.6.: The PyBox file mapping data structure for common settings

Member	Type
intercept	BOOL
preventExecution	BOOL
returnValue	LONG

Table 3.7.: The PyBox file mapping data structure for hook settings

path of the hook library. This information is required by the hook library in order to inject the hook library into processes which might be created by the monitored target process. Thus, this member provides the relevant information in order to monitor the behavior of all processes which are concerned by the execution of the actual target process.

The Hook Settings File Mapping

The hook settings file mapping contains the information which have been specified in the configuration file `hooks.cfg` described in Chapter 3.5.2. In this configuration file, the analyst using PyBox has specified all API functions which have to be hooked as well as whether to use the trampoline functions and which return values have to be returned. These information are mapped by the analysis tool to a file mapping object and thus can be obtained and realized by the hook library. In order to represent all hooks we use a C array of data structures. For each API function which can be hooked a data structure of type `MMF_SETTINGS_ITEM` is created. This data structure is depicted in Table 3.7. The member `intercept` describes whether to hook the function or not. The member `preventExecution` is used in order to specify whether or not the trampoline function is called. If the trampoline function is not called we have to define the return value of the API function. This value is contained in the member `returnValue`. The data structure's position in the array defines the concerned API function. This means that there is a fixed position in the array for each API function which can be hooked by PyBox. The hook library is implemented accordingly mapping the respective configuration to the corresponding hook. With these information read, all specified hooks are installed in the target process.

The Log Entry File Mapping

The log entry file mapping provides the information which has been logged in the callback functions of the hook library. For each hooked API call a new log entry is created. In order to provide the analysis tool with as much information as possible a data structure

Member	Type	Size
function	unsigned char	1
object	wchar_t[256]	512
comment	wchar_t[256]	512
executed	unsigned char	1
pid	DWORD	4
desiredAccess	DWORD	4
param1	unsigned long	8
param2	unsigned long	8
param3	unsigned long	8
param4	unsigned long	8
timestamp	long long	8
Size		1064

Table 3.8.: The PyBox file mapping data structure for log entries

has been chosen to be used in the file mapping containing the members depicted in Table 3.8. In the following, the members of this data structure are described.

- `function` contains the identification number of the hooked API function which has created the log entry.
- `object` specifies the object name on which the API function has been executed.
- `comment` is used as a placeholder for another object name. For instance, the hooks of the registry functions use this member for the value name of a registry key while the registry key is stored in `object`.
- `executed` determines whether or not the trampoline function has been called and thus whether or not the original functionality has been executed. If the value is set to `1` the trampoline function has been called. Otherwise, the value is set to `0`.
- `pid` defines the process identification number of the process in which the corresponding API function has been called.
- `desiredAccess` describes the access attributes used to access the object of the API call.
- `param1`, `param2`, `param3`, and `param4` are placeholders for various data that is transmitted. The contents of these members vary from API function to API function. Their interpretation is handled by the report creation in module `analysis` of the package `report` in the analysis tool which has been described in Chapter 3.5.4. The class `log_item` identifies the meaning of the various values based on the log entry's `function` value.
- `timestamp` specifies the time passed in milliseconds since the hook library has been injected.

In contrast to the first two file mappings outlined above, we have to add further functionality in order to use file mappings for the transmission of log entries. The first

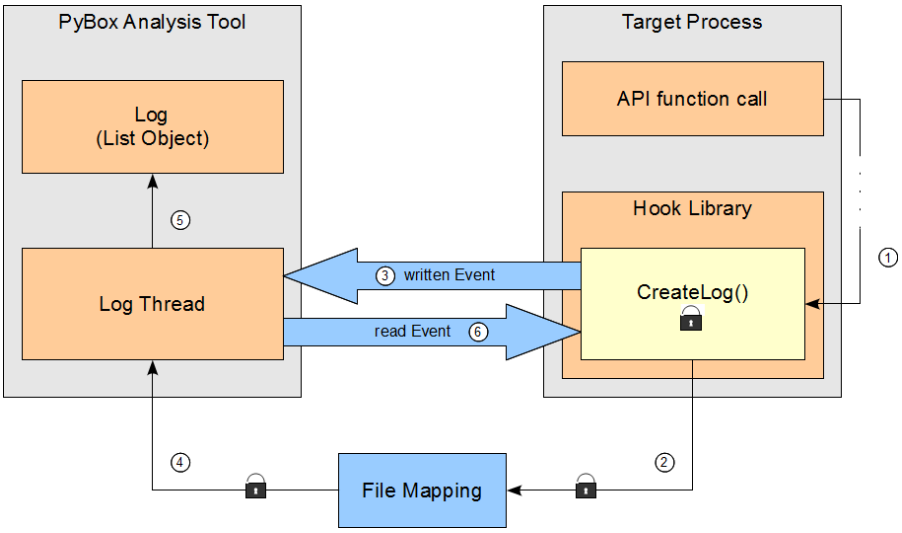

Figure 3.7.: PyBox IPC procedure for log entries

two file mappings are used only once in order to provide the configuration information copied and used by the hook library. Subsequently, these file mappings are of no use for the hook library in the target process anymore. At the same time, they must not be deleted, as the hook library might have to be injected into further processes which might be created by the target process. The hook libraries in these new created processes have to use these file mappings as well.

The log entry file mapping, however, transmits data in the opposite direction, from the hook client to the hook server. Furthermore, its content is permanently changed. This means, once a log entry is created and mapped to the file mapping object, the analysis tool must immediately read and copy the file mapping data. Otherwise, the data is overwritten by new log entries and information would get lost. As both processes have to access the file mapping permanently, means of data synchronization are required. However, the file mapping method, unlike other methods, does not provide the functionality of updating the communication members about the changes which have been caused. Thus, we have to add further functionality in order to implement IPC here. Windows provides so-called *synchronization objects*[22] in order to maintain data consistency of objects which are accessed by one or more threads or processes. Synchronization objects comprise *mutexes*, *semaphores*, *events*, and *waitable timers*. As we have only one file mapping which is concerned by the IPC, there is no need to use semaphores. Instead, we use two mutexes in order to guarantee that the data are only accessed by one process at a time. Furthermore, we have to use two events. One event has to signal to the hook library that the file mapping has been read and that new log entries can be created. The other event has to signal to the analysis tool that a new log entry has been created ensuring that the data are read immediately in order to prevent data loss. The procedure concerning the transmission of log entries is depicted in Figure 3.7.

The picture outlines six steps. Whenever the target process calls a hooked API function, the corresponding callback executes the `createLog` function provided in the hook

[22]http://msdn.microsoft.com/en-us/library/ms686364(v=vs.85).aspx

library. In Step 2 `createLog` waits for the mutex objects which allow to create the actual log entry object, access the file mapping, and to write the log data. After the information has been written, `createLog` releases the file mapping mutex and sets the `written` event in Step 3 which notifies the analysis tool that a new log entry has been created. In Step 4, the log thread acquires the file mapping mutex and copies the file mapping data to a new `log_item` object which is stored in the log list (Step 5). After the log entry has been read and copied, the mutex is released again and the `read` event in Step 6 is set by the log thread notifying the target process to proceed.

In the Listings 3.14 and 3.15, the implementation of the mentioned components is depicted. At first, we describe the log entry creation implementation of the hook library. Afterwards, we examine the log thread implementation applied in the analysis tool.

```
46 int __stdcall createLog(unsigned char function, wchar_t *object,
       wchar_t *comment, const char *buffer, unsigned char executed,
       unsigned int desiredAccess, unsigned long param1, unsigned long
       param2, unsigned long param3, unsigned long param4,
       SOCKET_ADDRESS_INFO socket_address, unsigned long return_value)
47 {
48   LOG newLog;
49   newLog.Function = function;
50
51   // Copy strings
52   wcsncpy_s(newLog.Comment, comment, MAX_STR_BUFFER-1);
53   newLog.Comment[MAX_STR_BUFFER-1] = '\0';
54   wcsncpy_s(newLog.Object, object, MAX_STR_BUFFER-1);
55   newLog.Object[MAX_STR_BUFFER-1] = '\0';
56   strncpy_s(newLog.Buffer, buffer, MAX_STR_BUFFER-1);
57   newLog.Buffer[MAX_STR_BUFFER-1] = '\0';
58
59   // Copy numeric values
60   newLog.Executed = executed;
61   newLog.PID = pid;
62   newLog.Timestamp = getTimeStamp();
63   newLog.DesiredAccess = desiredAccess;
64   newLog.Param1 = param1;
65   newLog.Param2 = param2;
66   newLog.Param3 = param3;
67   newLog.Param4 = param4;
68   newLog.socket_address = socket_address;
69   newLog.ReturnValue = return_value;
70
71   WaitForSingleObject(h_outer_mutex, INFINITE);
72   WaitForSingleObject(h_inner_mutex, INFINITE);
73   memcpy(mmf_log, &newLog, sizeof(newLog));
74   ReleaseMutex(h_inner_mutex);
75   SetEvent(h_event_w);
76   WaitForSingleObject(h_event_r, INFINITE);
77   ResetEvent(h_event_r);
78   ReleaseMutex(h_outer_mutex);
79   return 0;
80 }
```

Listing 3.14: Log entry creation

In order to create a log entry file mapping, we have to initialize a `LOG` data structure `newLog` and insert the corresponding values to be logged into its fields (lines 51 - 69). Having created the log entry data structure, we are ready to copy the information to the file mapping. Before we can actually copy the log entry, we have to acquire the required mutexes first (lines 71, 72). We have already obtained the required handles in the initialization sequence of the hook library. The first mutex referenced by `h_outer_mutex` ensures that no further log entry overwrites the existing one before the analysis tool has read the file mapping. The second mutex referenced by `h_inner_mutex` guarantees that the file mapping is only accessed by one process or thread at the same time. If only the inner mutex was applied, it would be possible that another process or thread would acquire the mutex before the analysis tool, the data would be overwritten and thus be lost. Having copied the log entry data structure to the buffer of the file mapping (line 73), the inner mutex can be released (line 74). Then, the analysis tool is notified that a new log entry has been created by using the `written` event referenced by the handle `h_event_w`. While the log thread of the analysis tool reads the provided data, the target processes waits until the `read` event is triggered referenced by the handle `h_event_r` (line 76). Finally, the outer mutex can be released (line 78) and the target process proceeds.

```
30 def log_thread(p_filemapping, h_mutex, h_event_written, h_event_read):
31     while forrest_gump:
32         # Wait for event
33         if kernel32.WaitForSingleObject(h_event_written, INFINITE) is 0:
34             # Wait for mutex to read log
35             kernel32.ResetEvent(h_event_written)
36             kernel32.WaitForSingleObject(h_mutex, INFINITE)
37             # Read log and append it to the log
38             a_lock.acquire()
39             try:
40                 log_entry = MMF_LOG()
41                 memcpy(byref(log_entry), p_filemapping,
                       sizeof(log_entry))
42                 if (log_entry.Function == 9) or (log_entry.Function ==
                       10):
43                     created_processes.append(log_entry.Param1)
44                 pbmonitor_log.append(log_entry)
45             except:
46                 print "[ERR] An error occurred during reading from the
                       file mapping."
47             a_lock.release()
48             kernel32.ReleaseMutex(h_mutex)
49             kernel32.SetEvent(h_event_read)
```

Listing 3.15: Log thread

The log thread of the analysis tool consists of an infinite loop that runs until no process has to be monitored any more. The thread waits until the `write` event referenced by the handle `h_event_written` is triggered (line 33). Subsequently, the mutex to the inner handle is acquired (line 36) and the log entry contained in the file mapping is copied (line 41) to a data structure which is the equivalent to the `LOG` data structure of the hook library and added to the list `pbmonitor_log` in which all log entries are stored (line

44). The outer handle is only used for the synchronization of the log creation between different target processes and therefore does not concern the analysis tool. Additionally, if the log item contains an entry concerning the `CreateProcess` or `CreateProcessEx` API function, the process identification number of the created process which is stored in the member `param1` is added to the list `created_processes` (lines 42, 43). This list is used to ensure that the analysis tool waits for all monitored processes before it terminates. Having read and copied all relevant data, the mutex is released (line 16) and the `read` event is set in order to notify the target process to proceed (line 49).

3.7. Summary

In this chapter, we have detailed the implementation of PyBox. After the description of several design goals, we have discussed the implementation of the hook library containing the hooking and monitoring functionality. More specifically, we have seen how the inline hooking method as well as callback and trampoline functions are implemented. This has been demonstrated using three different callback functions as examples. Additionally, we have outlined how the hook library can prevent its detection by using hooking. Apart from the hook library, we have detailed the implementation of the analysis tool. The analysis tool serves as the interface for user interaction. An analyst can apply it in order to configure the the hook library. Hence, we have described its setup and configuration files, the creation of the monitored target process with the injected hook library as well as the final report generation. Finally, we have described the implementation of the IPC method used for the communication and interaction of all concerned processes.

Having described the various implementation-specific details, we are now able to evaluate the result of the implementation by testing the framework on various examples. Thus, we can see the features and benefits of the created sandbox.

Chapter 4.

Outlook on Portability of PyBox towards Linux

In recent years, the usage of mobile devices such as PDAs, and smartphones with internet access has grown rapidly. Due to their increasing popularity these devices are also becoming more and more targets of malware attacks. But they don't run the same operating systems as desktop or notebook computers. This would be not efficient regarding issues such as battery consumption. Instead, they incorporate customized operating systems for small mobile devices which are more suited to their needs. Like malware developers, security specialists also have to focus on attacks on systems designed for mobile devices. One of the most popular systems is Google's Android which is based on the Linux kernel 2.6 according to the Android developer guide [Gooa]. Thus, it would be convenient, if we could extend the functionality of PyBox to Linux operating systems.

Throughout this chapter, we will address the possibilities of bringing the PyBox analysis environment to Linux. As we have seen before, dynamically analyzing a malware sample or any other executable requires basic knowledge about the respective operating system such as how applications interact with a system and the layout of executables. Therefore, in the first section, we provide basic information about relevant characteristics of Linux and derive suitable starting points to observe malware samples.

As described in chapter 2.3.2, we use hooking techniques in PyBox in order to reveal an executable's actions. The PyBox analysis tool which injects the customized code into the target process's address space and receives and evaluates the results of the monitoring process is written in Python. Python is available for both Windows and Linux, so this part of the environment can be used and just has to be extended. However, the hooking and code injection process is closely related to system characteristics and therefore has to be reimplemented. Thus, in the second part of this chapter we describe how hooking methods can be applied on a Linux operating system.

Finally, the third section outlines other alternatives of observing executables during run-time.

4.1. Linux Fundamentals

Linux[1] is a free unix-like operating system which was originally developed by Linus Torvalds. Nowadays, it is a collaborative project of countless software developers throughout the world. Although, the name "Linux" actually only refers to the kernel and thus to the core of an operating system, it is usually used for the entirety of various operating systems implementing the Linux kernel. The variants which are usually put to practical use are called *Linux distributions*. There are many different distributions, some of the most popular being Debian[2], Ubuntu[3], and Suse[4]. Due to its open-source kernel, Linux can be run on various hardware such as super computers, desktop computers, notebooks, servers, mobile devices, routers etc.

With regard to the different Linux products and the required information about system functionality for analyzing software, a common basis is needed in order to be able to use system functionality for the purpose of monitoring executables during runtime. Thus, in this section we take a closer look at common Linux system functionality. For this purpose, we explain system calls in Linux, introduce the term *ABI*, outline existing APIs used in Linux, describe the Linux executable file format, and derive similarities and possible starting points to observe malware samples.

System Calls

Virtual memory is separated into two different memory parts in Linux: user space and kernel space, just as in Windows. Only the kernel can access kernel space. User applications can only access user space and must not access kernel space in order to protect the operating system's integrity and stability and prevent misuse. On the other hand, applications must be able to somehow call system services in order to access the system's resources. For this purpose, *system calls* are used. System calls are provided by the operating system and can be called from user space. The user application can cause a software interrupt instruction and pass a number as argument thus triggering a certain process in kernel space associated with this number.

In Linux, the sum of system calls can vary from one machine architecture to another. However, Love [Lov07, p. 3] mentions that more than ninety per cent is implemented in all of them.

Application Binary Interfaces

Similar to an API, *application binary interfaces* (ABI) describe interfaces for the purpose of communication between different pieces of software. In contrast to an API, the ABI is more low-level defining the binary interfaces for the interaction between applications, kernel, and libraries. This means that the code which is compiled for one system will

[1] http://www.linux.org/
[2] http://www.debian.org
[3] http://www.ubuntu.com
[4] http://en.opensuse.org

work on another system without recompilation if it uses the same ABI. Thus, an ABI provides compatibility. Examples of ABI characteristics according to Love [Lov07, p. 5-6] are:

- Calling convention
- Byte order
- Register use
- System call invocation
- Linking
- Library behavior
- Binary object format

According to Love [Lov07, p. 5], at the moment there is no uniform architecture-specific ABI which is valid for multiple operating systems. Instead, each operating system defines its own ABI. Furthermore, there is a different ABI for each architecture on Linux. The ABI is usually realized by the so-called *toolchain*, including system tools such as the compiler and the linker. Thus, application developers usually don't have to care about the ABI as the C compiler and the C library take care in order to meet the ABI's standards. Yet, system programmers have to be aware of the respective architecture's ABI in order to build proper system software.

APIs in Linux

As described in 2.2.3, an API defines an interface which serves the purpose of communication. One piece of software offers a set of interfaces which are usually functions which can be utilized by other software. The provided interfaces can be used in order to make use of the providing software's functionality or resources. Thus, the software which provides an API has to implement it. One example of an API in Linux is the *C standard library*.

Standards are used to define APIs. The two main standards in Linux are the *Portable Operating System Interface (for Unix)* (POSIX) [IEE], the *Single UNIX Specification* (SUS) [Ope]. The POSIX project was created by the *Institute of Electrical and Electronical Engineers*[5] (IEEE) in the mid-1980s with the goal to harmonize Unix system-level interfaces. SUS emerged in 1994. Since version 3, SUS combines various standards including POSIX.1[6].

According to Love [Lov07, p. 6], Linux seeks to comply with POSIX and SUS. Yet, there is no official compliance to these two standards. In spite of several attempts to achieve this, there has not been much success. The difficulties are due to the dynamic nature of and permanent changes to Linux and the globally distributed collective of developers. Furthermore, there are very old parts of the system that already existed before these standards. This is why there are efforts made by some groups to address

[5] http://www.ieee.org
[6] http://www.unix.org/version3/apis.html

these problems by trying to make everything match the official standards in order to finally achieve full compliance. One of these attempts is the so-called *Linux Standard Base*[7] (LSB). The LSB is a collaborative group of several distributions headed by the *Linux Foundation*[8] trying to eliminate differences between the various distributions in order to match standards such as POSIX and SUS.

The C standard library meets these standards. It is one of the most important libraries in Linux. Most parts of the Linux kernel as well as most system calls are written in C. Even if other programming languages are used in order to develop applications, they are nevertheless dependent on this library. Very often, higher-level libraries use function wrappers of exported C library functions in order to simplify the use of system calls or other provided functionality. The most prevalent C standard library in Linux is *GNU libc*[9] (glibc), but there are also different versions. A major point of criticism of the GNU C library has been that it is too slow and too extensive especially regarding smaller devices such as smart phones. This is why on such platforms alternative C libraries are used. For example, Torvalds [Tor02] has pointed out these drawbacks. As an example of an alternative C library, Android implements an own BSD-derived standard C library which is called *Bionic* according to Maia et al. [CM10, p. 67].

Executlable and Linkable File Format

The current file format of executables files in Unix-like operating systems such as Linux is the *executable and linkable file format* (ELF) [TIS95]. It is the Linux equivalent to the Windows PE file format which has been described in Chapter 2.2.4. ELF has replaced several other executable file formats because of its extensibility and flexibility.

There are three main types of object files described by ELF:

- Relocatable files
- Executable files
- Shared object files

While *relocatable files* contain contents which can be linked in order to generate executable files or shared object files, programs are stored in *executable files* and can be executed. *Shared object files* can be linked with other object files in order to create a new object file or can be linked with an executable file and loaded into its process image in order to add functionality to the program. All three types are binary representations of the data and code they contain. This means, that they are already compiled and their code is directly executed by the processor.

Figure 4.1 depicts the structure of an ELF file. It is divided into a an ELF header, the program header table, various sections and segments, and finally a section table. The *ELF header* is at the beginning of each ELF file. It is a C data structure containing the main information about the file and how it is organized and structured. It also marks

[7] http://www.linuxfoundation.org/collaborate/workgroups/lsb
[8] http://www.linuxfoundation.org/
[9] http://www.gnu.org/s/libc/

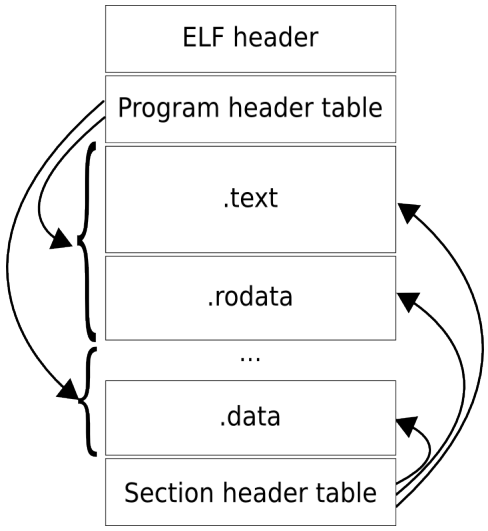

Figure 4.1.: Executable and linkable file format (ELF) structure

itself as an ELF file and describes for which architectures it is designed. Following the ELF header, there is the *program header table*. It is basically an array of data structures each describing a segment or other system information. The program header table is important for the execution of an application as it contains information about the creation of the process image. Subsequently, there are various *segments* described by the program header table. Each segment in turn can contain one or more *sections*. These sections contain the actual content of an object file such as the code to be executed and the data to be used. This means, there are various types of sections. There are for example `.text` sections containing the code to be executed, `.data` sections containing required data, or a `.got` section providing address information about exported functions of a shared object file which has been linked. At the end of an ELF file there is the *section table* which contains information about the file's sections such as their type and size.

4.2. Hooking in Linux

In 2.3.2, we have learned about different hooking and DLL injection methods which can be applied in Windows. Although the concept remains more or less the same, there are some differences when trying to implement such techniques in Linux. As described in chapter 3, we have to interact with the respective operating system and make use of various provided services. There are not necessarily equivalents for all of the API functions which we have used in Windows. Therefore, different ways have to be found in order to implement this functionality. In this section, we describe possible methods of hooking functions and injecting a customized shared object into a target process in Linux.

Hooking Methods

In the following, we learn about different hooking methods which can be applied in Linux. Since the respective implementation of those methods can vary from one Linux distribution or system to another, for example due to memory page protection issues, we will focus on their concepts.

Simple Hook One first simple method which can be used to hook function calls is to analyze the executable process image and manipulate all of these calls in order to redirect them to our customized code. The nature of a function call is displayed in listing 4.1. The call instruction with the opcode `0xE8` is followed by a four byte offset which leads the execution flow to the target function.

```
...
400521: e8 ce ff ff ff     call   4004f4 <some_function>
...
```

Listing 4.1: A function call

So, there is the possibility to examine the whole process image and replace the offsets of all function calls. Obviously, this method is not very efficient because the whole process image has to be parsed. Furthermore, we do not usually want to hook all functions of an executable. Thus, the effort to examine each function call by a whole process scan and just manipulate the offsets of those which we want to hook bears no relation to benefit. This method is also quite resource-intensive and is very likely to be detected. In fact, there are much more appropriate solutions for hooking functions.

GOT Hook An alternative hooking method for functions which are imported from a shared object in Linux is to modify the corresponding *global offset tables* (GOT). This technique was, for example, presented by Damato[Dam] at the *Defcon 18*[10] conference. In order to understand these hooks, we need to understand what the GOT is and how runtime dynamic linking works in Linux. In Listing 4.2 an example of a dynamic linking procedure is outlined.

```
08048464 <main>:
  8048474:   e8 2b ff ff ff     call   80483a4 <newobj@plt>
  ...
0804836e <.plt>:
  ...
  8048374:   e9 25 34 96 05 08 jmp    *0x8049634
  804837a:   68 08 00 00 00     push   $0x8
  8048380:   e9 e0 ff ff ff     jmp    8048374
  ...
```

Listing 4.2: Runtime dynamic linking in Linux

In this example, a regular call instruction calls the imported function. However, the call is not directed immediately to this function. Instead, the call leads to an instruction

[10] http://www.defcon.org/html/defcon-18/dc-18-index.html

block in the *procedure linkage table* (PLT). Usually, this block consists of a jump, a push and again a jump instruction. The second and third instructions are only required for the first time the function is called. The first jump instruction has a pointer as parameter. This pointer points to the corresponding entry in the GOT. The first time the function is called the entry leads the first jump instruction right to its subsequent push instruction in the PLT. Then, the runtime dynamic linker is invoked which fills the right offset to the desired function into the GOT entry. Afterwards, it is possible to jump to the desired function. The second time the function is called the push instruction and the second jump instruction are no longer required because the GOT entry directs the execution flow immediately to the function's address.

This mechanism provides us with the opportunity to hook these functions by manipulating the GOT entries. At this point, one should note, that every object such as every imported shared object as well as the main executable has its own GOT table. Thus, we have to manipulate all of them in order to observe the entire activity of the executable. The benefit of this characteristic is such that we can hook all GOT tables except the one of the shared object which we have ideally injected in order to provide our customized code. Thus, we can still use the functionality of the original functions by just calling them out of our own shared object containing a not manipulated GOT.

Inline Hook We have already covered the concept of inline hooking in detail in 2.3.2. It can also be applied in Linux. Therefore, the address of the target function to be hooked has to be discovered. Then, the first five bytes are copied to a trampoline function and are then overwritten by a jump instruction with the opcode `0xE9` followed by a four byte offset leading to the customized hook function. A simplified implementation is depicted in listing 4.3.

```
void inline_hook( void *original_function, void *hook_function,
  void *trampoline_function)
{
  char *buf;
  unsigned int offset;

  // Backup original function to trampoline function
  memcpy(trampoline_function, target_function, 5);

  // Determine offset to hook function
  offset = (unsigned int)hook_function -
           ((unsigned int)target_function + 5);

  // Overwrite original bytes with jump
  *buf    = (char *)target_function;
  *buf    = 0xE9
  *(buf+1)  = offset & 0xFF;
  *(buf+2)  = (offset >> 8) & 0xFF;
  *(buf+3)  = (offset >> 16) & 0xFF;
  *(buf+4)  = (offset >> 24) & 0xFF;
}
```

Listing 4.3: A simplified inline hooking implementation

In order to make this code work, further instruction are required which have been neglected here. For example, the corresponding memory page has to be set to be writeable, otherwise a memory error will show up. The procedure to do this depends on the used operating system. Another possibility to manipulate a process's memory during runtime for the purpose of inline hooks is to fork a process, create a child process executing the target and use ptrace to edit memory and change the process's execution flow. Ptrace is covered in more detail in 4.3.2. Another issue is to find the memory addresses of the functions involved. For this purpose, there are two system calls. The first system call is *dlopen*[11] which can load libraries and returns a handle to the object containing the wanted function. Then the function's address can be determined by usage of the system call *dlsym*[12].

A popular rootkit technology is to use inline hooks to redirect system calls. The system call table is basically an array of function pointers. Although it is usually not exported there are ways to discover its location according to [WJCN09, p. 550-551]. Consequently, by manipulation of those pointers kernel-level inline hooking can be realized and thus monitoring of system calls.

Injection Technique

As mentioned before, we have to find a way to inject our own customized code into the target process in order to redirect the hook functions to them. As we have seen, in Windows this is rather simple as we can use specific API functions to allocate memory, create a remote thread and load libraries in the target process. In Linux, it is much more difficult. While there are certain services that provide the functionality to load a library, open a process and write to memory, no functionality is offered to create a remote thread and to allocate memory in a remote target process. Furthermore, the `dlopen` function does not reside at the same memory address in all processes like its equivalent `LoadLibraryA` in Windows. The development of a solution to inject shared objects into a remote process is far from being trivial. Yet, there is a solution. In 2001, Clowes[Clo] presented a tool called *injectso* at the Blackhat Briefings in Amsterdam[13]. Injectso can be used to inject shared objects into processes during runtime. It attaches to a process using ptrace, locates the `dlopen` function by reading the process memory and locating the relevant information. Since there is no way to create a new thread in the remote process, injectso stops the main process storing all current registers and an apropriate portion of the stack. Then, it creates a new fake stack frame and sets up the registers so that the process will run dlopen with the customized shared library as argument. Finally, the original registers and the stack are recovered and the process resumes.

[11] http://linux.die.net/man/3/dlopen
[12] http://linux.die.net/man/3/dlsym
[13] http://www.blackhat.com/html/bh-europe-01/bh-europe-01-speakers.html

4.3. Monitoring System Calls

4.3.1. Hooking Targets

In PyBox, we use userland inline hooks in order to analyze the behavior of an executable file. However, we need to know the targets of these hooks in order to apply these methods. In Windows, we use Native API functions as hooking targets because they process the system calls, thus offering a direct interface to the Windows kernel. By this approach, we receive detailed information about the observed executable's execution flow. But if we want to hook system calls in Linux we face some difficulties. As mentioned in 4.1 the manner how system calls are called depends on the corresponding ABI. The ABI in turn varies from one architecture to another. Therefore, unless there is no specific architecture defined for the objects to be observed, the preferred solution is to focus on the system call implementations of the C standard library. In order to apply this solution we have to be aware of the respective C library and its implemented system calls. However, when hooking imported system call functions of the C standard library, there is the risk that system calls might not be registered in case the observed target executable avoids the C standard library and executes the system calls directly.

Thus, the optimal target to be hooked greatly depends on the the specific situation. This means, we first have to define which operating systems, machine architectures and C libraries have to be covered by the hooking and analysis environment before implementing an appropriate solution.

4.3.2. Tools and Alternative Methods to Observe an Executable's Execution Flow

So far, we have focused on hooking methods in order to monitor an executable's behavior. But hooks are not the only options to do this. Linux also provides utilities which enable us to observe and control the execution flow of a process. Two of these utilities are ptrace and strace. In the following, both utilities are briefly described.

ptrace

Ptrace[14] is a system call provided by Linux and other Unix-like operating systems. It is a process tracing tool which allows a parent process to attach to another process as well as to observe and control it. Thus, it can be also be used to intercept system calls both at their entry and exit points. Aside from the interception of system calls, ptrace is primarily used for debugging purposes. The structure of a ptrace instruction is depicted in listing 4.4.

[14] http://www.kernel.org/doc/man-pages/online/pages/man2/ptrace.2.html

```
long ptrace( enum __ptrace_request request,
             pid_t pid,
             void *addr,
             void *data);
```

Listing 4.4: Ptrace call structure

The first argument determines the action to be done by ptrace. The second argument `pid` receives the process identification number of the process which is controlled. The third argument `addr` is used if the ptrace request refers to a specific memory address of the target process and the last argument `data` is a pointer to the data which are used in combination with the defined address.

There are two alternatives to trace a process using ptrace. The first way is to call ptrace with a `PTRACE_ATTACH` request and the target's process identification number as arguments. The second way makes use of the system call `fork` which copies the existing process and creates the additional one as a child process. The child process can then call ptrace with a `PTRACE_TRACEME` request in order to be traced. While being traced by ptrace, the child process stops every time a signal is delivered. During the execution of the child process, the parent process uses a `wait` instruction in order to wait for the child process to stop. If it stops, the parent process executes some arbitrary code and resumes the execution of the child process. Ptrace not only provides the functionality to stop and resume a process but also provides mechanisms by which operations can be executed on the child process in order to modify its data or its execution flow. For example, the requests `PTRACE_POKETEXT`, `PTRACE_POKEDATA`, `PTRACE_PEEKTEXT`, `PTRACE_PEEKDATA` allow read from and write to the child process's memory no matter what page protection the corresponding page is set to. Furthermore, the request `PTRACE_SYSCALL` provides the functionality to observe system calls and the arguments passed to them.

Thus, the ptrace utility poses an alternative to hooking methods in order to observe the execution flow of an executable. But it can also be used within those methods. Injectso, for example, uses ptrace in order to execute the `dlopen` instruction inside the remote process.

Yet, according to Keniston et al. [KMPP07] there are also some disadvantages concerning ptrace.

- Ptrace is not a POSIX system call. The consequence is that its implementation and behavior can be quite different on various operating systems and architectures.
- Applying ptrace on one process is relatively simple, but if we want to observe more than one process or even threads it can be very difficult and cumbersome.
- There is a large overhead if the child's process registers and memory are edited.

strace

All standard Linux distributions provide the command *strace*. Strace provides the functionality to attach to a specific user application and to trace all system calls including

its parameters and its return values. It can also be used for multi-threaded applications. In order to realize its functionality, strace makes use of the ptrace system call and its request `PTRACE_SYSCALL`. Therefore, strace suffers from the same disadvantages as ptrace.

Thus, strace as well as ptrace serve the goal to observe an executable's behavior. Even if the functionality of strace proves insufficient for specific details of the malware analysis process which are needed for PyBox, it is possible to develop an own ptrace-based utility or even hooking environment in order to meet the requirements.

4.4. Summary

During this chapter, we have learned about fundamentals of the Linux system. It is difficult to state common characteristics which can be used for the purpose of analyzing executables during runtime because these characteristics can vary from operating system to operating system and from architecture to architecture. Furthermore, there can be different C standard libraries on which the system depends. Based on these fundamentals, three different methods of implementing hooks have been described. Their exact implementation can depend on the respective Linux system used. Finally, with ptrace and strace two alternatives of observing the execution flow of executables have been outlined.

Chapter 5.

Evaluation

So far, we have described fundamentals of malware and the Windows operating system, we discussed various techniques that enable us to monitor malware samples, and combined these techniques and implemented a sandbox solution that we have named "PyBox". In this chapter, we evaluate the created analysis environment. For this purpose, we test PyBox by performing several analysis processes in order to demonstrate its functionality and features. At first, we outline the applied analysis procedure in Section 5.1. In Section 5.2, we provide different executables, analyze their behavior and evaluate the results. In Section 5.3, we conclude the results of the executed test runs and evaluate PyBox's functionality.

5.1. Analysis Procedure

In this section, we describe the necessary steps applied in the following sections of this chapter in order to perform analyses of given executable samples. As starting point of these analysis processes, we use a VirtualBox VM as sandbox environment. In what follows, we refer to this VM as "PyBox VM". The PyBox VM runs a Microsoft Windows XP operating system with Service Pack 3. It also implements the PyBox analysis tool as well as the hook library which we have both examined in detail in Chapter 3. A snapshot of this system provides a clean system state based on which we can start each analysis process.

The process is divided into six steps:

1. The executable sample to be monitored is copied to the PyBox VM.
2. The PyBox configuration file `pybox.cfg` is configured as follows. The value of `EXE_TARGET` is set to the file path of the target executable file to be monitored. We also set the `TIMEOUT` value to `''120''` in order to avoid the situation of a not terminating process. This way the monitored process is terminated after an time-out interval of two minutes in case it is still running at this point in time. Furthermore, we set the `LOG_FOLDER` to the desired output folder and specify the file path of `hooks.cfg` in `CFG_HOOKS`.
3. In the hook configuration file `hooks.cfg`, all hooks are activated by setting their values `Intercept` to `1`. We do not prevent their execution and therefore set the values `PreventExecution` and `ReturnValue` to `0` for all available API functions.

4. The actual analysis is initiated by running the analysis tool `pybox.py` from the command line and passing the file `pybox.cfg` as an argument.
5. As soon as the the analysis tool has terminated and has output the report to the desired folder, we copy the report to the host machine.
6. The VM is shut down and restored to the snapshot providing the clean state.

5.2. Analyses of Executable Samples

Using the six steps outlined in the previous section, we are able to apply and test the created analysis environment. In this section, we put PyBox to practical use in order to show its features. Thus, we depict what type of information one can expect from such an analysis and how this information can be evaluated. We present some executable samples which are executed and analyzed by PyBox. After each analysis process, we examine the output XML report in detail. At first, we present a special application which has been designed to test the functionality of PyBox. By analyzing the test application and comparing its activities to the result of the analysis process, we can verify the correct functioning of PyBox. Furthermore, we demonstrate what data is logged in case an API function is monitored. Additionally, we analyze two malware samples in order to verify that PyBox reveals malicious behavior and provides useful information for analysts.

5.2.1. PyBox Test Application

The PyBox test application is a command line application which has been written for the purpose of testing PyBox's functionality. It calls various API functions that we try to monitor during the analysis processes. As we want to monitor all activities conducted by an executable, we have to verify that other processes created by the original target process are observed as well. Therefore, the test API calls are distributed among several executables. As in what follows, we take a closer look at the implementation of the test executables. Subsequently, we run a PyBox analysis process on the test environment and compare its result to the implemented test API functions.

Figure 5.1 depicts the design of the test environment. The application consists of four executables. The target executable passed to PyBox is `PyBox-Tester.exe`. All other executables are executed by this program. The test functions implemented in these files can be categorized into four test modules: file management, registry-related, process-related, and network API calls. Thus, every test module covers a certain test scenario. The different scenarios are tested subsequently. All functions we want to monitor are displayed in the different test modules of Figure 5.1.

The test application does not implement any native API calls directly. Instead, the Windows API is used because it is much easier to implement. Since the Windows API implements the native API calls, as we have described in Section 2.2.3, this aspect has no affect on the outcome of the analysis process.

The process running `PyBox-Tester.exe` invokes the different test modules categories

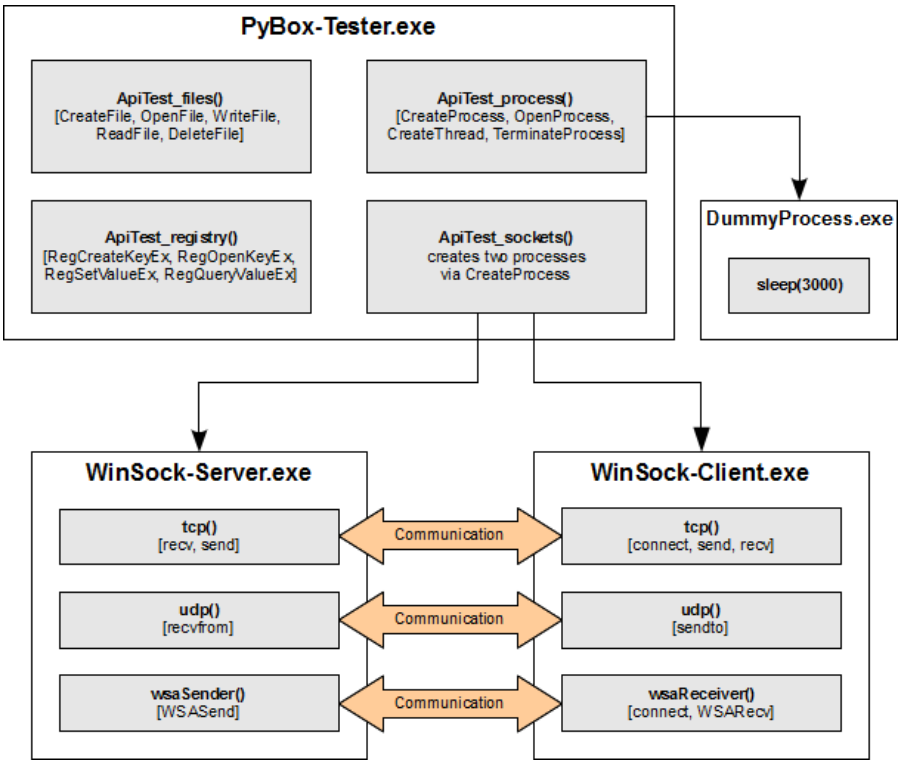

Figure 5.1.: Test application structure

of API functions. At first, `ApiTest_files` is called, which calls the different file management API functions. It creates a file `CreateFileW.txt` using the API `CreateFile`. Subsequently, the module writes a test string to it (`WriteFile`) and reads it out again (`ReadFile`). Finally, the created file is deleted via `DeleteFile`. Secondly, the test module `ApiTest_registry` is called. This module creates (`RegCreateKeyEx`) and opens (`OpenKeyEx`) two registry keys. Subsequently, an integer value is set to one of the created keys (`RegSetValueKeyEx`) and readout again (`RegQueryValueKeyEx`). After the test of all registry-related API functions, the test application calls the third test module `ApiTest_process` executing all process-related API functions. It creates a process running the executable `DummyProcess.exe` which does not implement any functionality. It only serves as an object on which we test the process API functions. Since we have to use the created process, we have to ensure that it does not immediately terminate. Therefore, we cause the process to sleep for three seconds. Within this time interval, we open the process (`OpenProcess`), create a new thread inside via `CreateRemoteThread` and terminate it (`TerminateProcess`). Finally, the test application runs the network test module by calling the function `ApiTest_socket`. Socket API functions, however, serve the purpose of communication between two or more objects. Hence, we do not test their functionality inside this process. Instead, the test application creates two further processes: a server process (`WinSock-Server.exe`) and a client (`WinSock-Client.exe`) process. Both associated executables implement three different test modules which establish three different communication channels between the two processes by applying three different types of socket API functions. The first channel serves the tests of the socket API functions `connect`, `send`, and `recv` provided by `ws2_32.dll` library. The second

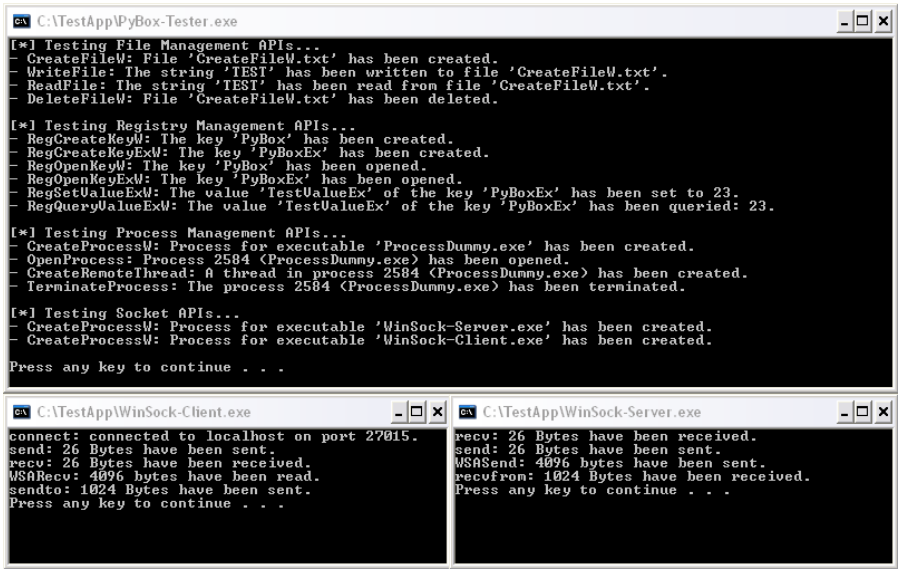

Figure 5.2.: Test application output

channel tests the communication via the socket API functions `sendto` and `recvfrom`. The last communication channel establishes a connection via `connect`, which enables the communication via the API functions `WSASend` and `WSARecv`. In order to verify that all API functions have been executed properly, the command line console displays their results. Figure 5.2 provides a screenshot displaying the output of the test environment.

As depicted in Figure 5.2, all API functions which we intend to monitor have been executed successfully. In order to verify that PyBox keeps track of all relevant function calls, we run a PyBox analysis monitoring the execution of the test environment as described in Section 5.1. The resulting report contains more than thousand lines and thus is far too extensive to be listed entirely here. Therefore, we focus only on the relevant entries concerning the test API functions called by the test environment. We depict different parts of the XML report and explain all relevant entries.

```
2 <PyBox-Analysis Executable="C:\TestApp\PyBox-Tester.exe" FileSize="37888
    " LogFile="...\PyBox-Log_20110425-222852.xml" MD5="..." SHA1="..."
    Time="25.04.2011 22:28:52">
3   <calltree>
4     <process filename="C:\TestApp\PyBox-Tester.exe" index="1" pid="2824"
        starttime="0">
5       <process filename="C:\TestApp\ProcessDummy.exe" index="2" pid="672
          " starttime="26"/>
6       <process filename="C:\TestApp\WinSock-Server.exe" index="3" pid="
          3264" starttime="1057"/>
7       <process filename="C:\TestApp\WinSock-Client.exe" index="4" pid="
          2776" starttime="2089"/>
8     </process>
9   </calltree>
```

Listing 5.1: Monitored call tree

First of all, we take a look at the root tag `<PyBox-Analysis>` in Listing 5.1. It contains the information about log file and the monitored main executable such as its file path, file size, and hash values. Below this tag, the call tree lists all processes that have been created during the analysis process. It provides information about which process has called the other and in what order the processes have been called. Furthermore, it specifies the time interval between injection of the hook library and process creation in milliseconds. Listing 5.1 depicts the root tag and the call tree of the report documenting the activities of the test application.

The call tree lists all created processes of the test environment as well as the correct order that we have described above and depicted in Figure 5.1. At first, `PyBox-Tester.exe` is executed creating the processes `ProcessDummy.exe`, `WinSock-Server.exe`, and finally `WinSock-Client.exe`.

The monitored activities of each process are listed within the tag `<processes> ... </processes>` below the call tree. We examine certain entries in order to check whether or not the executed API calls have been detected.

```
10  <processes>
11    <process Filename="C:\TestApp\PyBox-Tester.exe" Index="1" PID="2824"
          Starttime="0">
12      <filesystem_section>
13        <NtCreateFile CreateDisposition="0x00000005" CreateOptions="0
              x00000060" DesiredAccess="0xc0110080" Executed="YES"
              FileAttributes="0x00000080" Object="C:\TestApp\CreateFileW.
              txt" ReturnValue="0x00000000" ShareAccess="0x00000000"
              Timestamp="4"/>
14        <NtWriteFile Executed="YES" Length="10" Object="TestApp\
              CreateFileW.txt" ReturnValue="0x00000000" Timestamp="5"/>
15        <NtReadFile Executed="YES" Length="10" Object="TestApp\
              CreateFileW.txt" ReturnValue="0xc0000011" Timestamp="5"/>
16        <NtOpenFile DesiredAccess="0x00010080" Executed="YES" Object="C:
              \TestApp\CreateFileW.txt" OpenOptions="0x00204040"
              ReturnValue="0x00000000" ShareAccess="0x00000007" Timestamp=
              "6"/>
```

Listing 5.2: Monitored file management API functions

In Listing 5.2, all entries concerning the called file management API functions can be found. Line 13 reveals that the file `CreateFileW.txt` has been created. Subsequently, 10 bytes are written to the created file (line 14). Afterwards 10 bytes are read (line 15) and a value of `0xc0000011` is returned. This value represents the flag `STATUS_END_OF_FILE` which means that the application could not read all 10 Bytes because it has reached the end of the file. The last monitored function call concerning the observed file is `NtOpenFile` in line 16. The file is opened using the flag `DELETE` (`0x00010000`), i.e. this API function is executed in order to perform our Windows API `DeleteFile` call.

In Listing 5.3, the monitored API call entries of the registry test module are depicted. They are listed in the registry section of the `PyBox-Tester.exe` process. PyBox reveals that the test application has created two keys `PyBox` and `PyBoxEx` (lines 31, 32). Moreover, both keys are opened (lines 33, 34) and a value `TestValueEx` is assigned to the

key `PyBoxEx` (line 35). Finally, an entry `NtQueryValueKey` (line 36) displays that the assigned value has been queried as well.

```
30      <registry_section>
31        <NtCreateKey CreateOptions="0x00000000" DesiredAccess="
              MAXIMUM_ALLOWED" Executed="YES" Object="REGISTRY\MACHINE\
              SOFTWARE\PyBox" ReturnValue="0x00000000" Timestamp="9"/>
32        <NtCreateKey CreateOptions="0x00000000" DesiredAccess="0
              x00020006" Executed="YES" Object="REGISTRY\MACHINE\SOFTWARE\
              PyBoxEx" ReturnValue="0x00000000" Timestamp="9"/>
33        <NtOpenKey DesiredAccess="MAXIMUM_ALLOWED" Executed="YES" Object
              ="REGISTRY\MACHINE\SOFTWARE\PyBox" ReturnValue="0x00000000"
              Timestamp="10"/>
34        <NtOpenKey DesiredAccess="0x000f003f" Executed="YES" Object="
              REGISTRY\MACHINE\SOFTWARE\PyBoxEx" ReturnValue="0x00000000"
              Timestamp="10"/>
35        <NtSetValueKey DesiredAccess="0x00000000" Executed="YES" Object=
              "REGISTRY\MACHINE\SOFTWARE\PyBoxEx" ReturnValue="0x00000000"
               Timestamp="10" ValueName="TestValueEx" ValueType="4"/>
36        <NtQueryValueKey DesiredAccess="0x00000000" Executed="YES"
              KeyValueInformationClass="2" Length="144" Object="REGISTRY\
              MACHINE\SOFTWARE\PyBoxEx" ReturnValue="0x00000000" Timestamp
              ="11" ValueName="TestValueEx"/>
```

Listing 5.3: Monitored registry API functions

The report entries associated with the test module calling all process-related API functions are listed in the XML file's registry section of the `PyBox-Tester.exe` process tag and are depicted in Listing 5.4. PyBox reveals these function calls as well: It has been monitored that the function `NtCreateProcessEx` (line 138) has been called in order to create the process for `DummyProcess.exe`. The new process has the identification number `672`. Remembering this number, we are able to determine the activities which are performed on this process. In the lines 144 and 145, it is documented that the process has been opened and a remote thread has been created via `NtCreateThread`. The attribute `CreateSuspended` tells us that `PyBox-Tester.exe` has created a thread in suspended mode which means that the thread is created, but not yet executed. In the Window XP operating system all threads are created using this flag in order to report the created thread to the Windows subsystem before its execution. Subsequently, the termination of the created process via `NtTerminateProcess` is reported. Furthermore, the creation of the two processes `WinSock-Server.exe` (line 149) and `WinSock.exe` (line 154) is displayed before the process terminates (line 157).

```
137     <process_section>
138       <NtCreateProcessEx DesiredAccess="0x001f0fff" Executed="YES"
              NewPID="672" Object="C:\TestApp\ProcessDummy.exe" ParentPID=
              "2824" ReturnValue="0x00000000" Timestamp="26"/>
139        ...

144       <NtOpenProcess DesiredAccess="GROUP_WRITE_ACCOUNT" Executed="
              YES" Object="" ObjectPID="672" ParentPID="2824" ReturnValue=
              "0x00000000" Timestamp="1024"/>
145       <NtCreateThread CreateSuspended="0x00000001" DesiredAccess="0
```

```
            x001f03ff" Executed="YES" Object="" ObjectPID="672"
            ObjectTID="2440" ReturnValue="0x00000000" Timestamp="1029"/>
146         <NtTerminateProcess Executed="YES" ExitStatus="0" ObjectPID="
            672" ParentPID="2824" ReturnValue="0x00000000" Timestamp="
            1031"/>
147         ...

149         <NtCreateProcessEx DesiredAccess="0x001f0fff" Executed="YES"
            NewPID="3264" Object="C:\TestApp\WinSock-Server.exe"
            ParentPID="2824" ReturnValue="0x00000000" Timestamp="1057"/>
150         ...

154         <NtCreateProcessEx DesiredAccess="0x001f0fff" Executed="YES"
            NewPID="2776" Object="C:\TestApp\WinSock-Client.exe"
            ParentPID="2824" ReturnValue="0x00000000" Timestamp="2089"/>
155         ...

157         <NtTerminateProcess Executed="YES" ExitStatus="0" ObjectPID="0"
             ParentPID="2824" ReturnValue="0x00000000" Timestamp="3112"/
            >
158       </process_section>
```

Listing 5.4: Monitored process API functions

Finally, we have to verify PyBox's monitoring abilities regarding the usage of socket API functions. As mentioned before, there are two processes, `WinSock-Server.exe` and `WinSock.exe`, which communicate with each other using three communication channels. In Listing 5.4, we have noticed that both processes are created. Since both processes have been created by the monitored process `PyBox-Tester.exe`, they have been monitored and are listed in the report as well. Therefore, we also find two tags inside `<processes> ... </processes>` representing them. The report entries concerning the called socket API functions can be found in their network sections. At first, we examine the `WinSock-Server.exe` which is listed in Listing 5.5. Subsequently, we also consider the report's entries concerning the `WinSock-Client.exe` process depicted in Listing 5.6.

The process `WinSock-Server.exe` is a small application opening three different sockets and waiting for incoming connections. The first monitored activity occurs as soon as a the client is connected: The process receives a message via `recv` (line 578) and sends a message in response via `send` (line 581). This is what we have covered in the test module for socket API calls of the test environment. The second communication channel captured in the report is the usage of the API call `WSASend` (line 587): The process listens on a socket and waits until a client has connected. As soon as the connection is established, it sends the depicted test message: "Test buffer sent via WSASend." (line 588). Finally, the report shows that some data is received via the APIfunction `recvfrom` as well. With this logged event, we can tell that the test environment has successfully transmitted messages using three different methods of communication via sockets and all corresponding events have been successfully captured by PyBox.

```
577     <network_section>
578       <recv BufferLength="30" Executed="YES" Flags="0x00000000"
              ReturnValue="26" Socket="280" Timestamp="2146">
579         Test buffer sent via send.
580       </recv>
581       <send BufferLength="26" Executed="YES" Flags="0x00000000"
              ReturnValue="26" Socket="280" Timestamp="2147">
582         Test buffer sent via send.
583       </send>
584       <recv BufferLength="30" Executed="YES" Flags="0x00000000"
              ReturnValue="0" Socket="280" Timestamp="2147">
585         Test buffer sent via send.
586       </recv>
587       <WSASend BytesSent="4096" Executed="YES" Flags="0x00000000"
              ReturnValue="0" Socket="284" Timestamp="2699">
588         Test buffer sent via WSASend.
589       </WSASend>
590       <recvfrom AddressFamily="2" BufferLength="1024" Executed="YES"
              Flags="0x00000000" ReturnValue="1024" Socket="288"
              SocketAddress="127.0.0.1" SocketPort="5637" Timestamp="3255"
              >
591         Test buffer sent via sendto.
592       </recvfrom>
593     </network_section>
```

Listing 5.5: Monitored network API functions of `WinSock-Server.exe`

The counterpart to the server process is the client process. The two processes communicate with each other in order to transmit data. The client's activities are listed in the process's network section in the XML report. This section is displayed in Listing 5.6. As displayed in the other sections of the XML report and described above, we can also derive the network-related activites of `WinSock-Client.exe` from the PyBox report: The client establishes two connections using the `connect` API call (lines 1275, 1285) in order to connect to the respective server process and to transmit data via `send` and `recv` (lines 1276 - 1284) as well as to receive data via `WSARecv`. Additionally, data is sent to the waiting `recvfrom` function in the server process using the `sendto` API call (line 1292).

All log entries contain attributes and content providing information about the sent data. The first 255 bytes of the messages transmitted between the client and the server are displayed between the tags representing the corresponding API call. Furhermore, the single entries provide additional information about the transmitted data such as the length of the buffer, the used socket, as well as certain flags which have been specified. Thus, we can for example see that all processes connect to the socket address "127.0.0.1" which is the IP address of the local system.

```
1274     <network_section>
1275       <connect AddressFamily="2" Executed="YES" ReturnValue="0"
              Socket="308" SocketAddress="127.0.0.1" SocketPort="34665"
              Timestamp="1100"/>
1276       <send BufferLength="26" Executed="YES" Flags="0x00000000"
              ReturnValue="26" Socket="308" Timestamp="1101">
```

```
1277            Test buffer sent via send.
1278          </send>
1279          <recv BufferLength="30" Executed="YES" Flags="0x00000000"
                  ReturnValue="26" Socket="308" Timestamp="1104">
1280            Test buffer sent via send.
1281          </recv>
1282          <recv BufferLength="30" Executed="YES" Flags="0x00000000"
                  ReturnValue="0" Socket="308" Timestamp="1104">
1283            Test buffer sent via send.
1284          </recv>
1285          <connect AddressFamily="2" Executed="YES" ReturnValue="0"
                  Socket="312" SocketAddress="127.0.0.1" SocketPort="34665"
                  Timestamp="1652"/>
1286          <WSARecv BytesReceived="3435973836" Executed="YES" FlagsIn="0
                  x00000000" FlagsOut="0x00000000" ReturnValue="4294967295"
                  Socket="312" Timestamp="1652"> </WSARecv>
1287          <WSARecv BytesReceived="4096" Executed="YES" FlagsIn="0
                  x00000000" FlagsOut="0x00000000" ReturnValue="4294967295"
                  Socket="312" Timestamp="1653">
1288            Test buffer sent via WSASend.
1289          </WSARecv>
1290          <sendto AddressFamily="2" BufferLength="1024" Executed="YES"
                  Flags="0x00000000" ReturnValue="1024" Socket="304"
                  SocketAddress="127.0.0.1" SocketPort="34665" Timestamp="2209
                  ">
1291            Test buffer sent via sendto.
1292          </sendto>
1293        </network_section>
```

Listing 5.6: Monitored network API functions of `WinSock-Client.exe`

Although we have only examined a minor part of the entire XML report, we can conclude, that we have developed a test environment which successfully executes all required API functions. Furthermore, we have verified that PyBox has monitored and logged all required API calls and written the associated information to the XML report.

5.2.2. Analyses of Malware Samples

So far, we have only tested PyBox on the test environment in order to verify the required functionality. The actual purpose of PyBox, however, is to monitor and reveal the behavior of malware samples. Therefore, we have to test it on actual malicious binaries as well in order to complete the evaluation of PyBox. In this section, we present two different malware samples, evaluate the corresponding reports produced by PyBox, and try to derive the malware's behavior.

Malware Sample: Backdoor.Knocker

The first malware sample analyzed in this section can be categorized as back door, trojan horse, or bot. Information about this threat is provided by VirusTotal [Vir], Prevx [Pre], and ThreatExpert [Thr]. ThreatExpert refers to it as *Backdoor.Knocker*. Its purpose

is to frequently send information about the infected system to its home server. This information may include the operating system version, its IP address, and open ports.

In order to draw conclusions about the malware's behavior, we run a PyBox analysis as described in Section 5.1. The corresponding report, again, is far too extensive to depict all monitored activities in this thesis. Therefore, we focus on certain parts which indicate malicious behavior. We divide the report evaluation into different analysis parts according to the four different categories of monitored API calls: file managent, registry, process, and network. The entire report can be found on the CD-ROM attached to this diploma thesis. It should be noted, that the analysis process has terminated the target process after the time-out interval of two minutes.

```
30 <NtCreateFile CreateDisposition="0x00000005" CreateOptions="0x00000060"
       DesiredAccess="0x10100080" Executed="YES" FileAttributes="0
      x00000000" Object="C:\WINDOWS\system32\Hl7fP2.syz" ReturnValue="0
      x00000000" ShareAccess="0x00000003" Timestamp="572"/>
31 <NtWriteFile Executed="YES" Length="5856" Object="WINDOWS\system32\
      Hl7fP2.syz" ReturnValue="0x00000000" Timestamp="575"/>
```

Listing 5.7: Malware sample 1 - file management section - extract 1

The report's call tree shows only one process, that is the binary executed by the analysis tool. Therefore, we only have to focus on this process's activites. Its file section entries reveal that many system files and devices are opened, created, and read. The first entry that we would like to point out is the creation of a file `Hl7fP2.syz` in line 30. The file is created in the Windows system folder `C:\WINDOWS\system32`. The next entry documents the next operation on this file. The process writes content of 5862 bytes to this file. Both entries are depicted in Listing 5.7. In what follows, the file is opened several times and put to further use. Both the creation of a file in the Windows system folder and the odd file name are aspects which indicate malicious behavior. The report of a second analysis run, that can also be found on the attached CD-ROM, reveals that another file name has been used. This tells us that the process dynamically generates the file name, probably due to the fact that it tries to prevent the file from being detected.

```
227 <NtCreateFile CreateDisposition="0x00000005" CreateOptions="0x00000064"
       DesiredAccess="0x40110080" Executed="YES" FileAttributes="0
      x00000020" Object="C:\WINDOWS\system32\cssrss.exe" ReturnValue="0
      x00000000" ShareAccess="0x00000000" Timestamp="3115"/>
228 <NtWriteFile Executed="YES" Length="20928" Object="WINDOWS\system32\
      cssrss.exe" ReturnValue="0x00000000" Timestamp="3116"/>
```

Listing 5.8: Malware sample 1 - file management section - extract 2

In line 227, the report documents that the target process has created another file in the Windows system folder and written data to it. This time, the file has been named `cssrss.exe`. These entries also indicate malicious behavior as the file name is very similar to the file name `csrss.exe`. The latter is the executable of the `Client Server Run-Time Subsystem` (CSRSS) which is an important part of the Windows operating system.

```
1211 <NtOpenKey DesiredAccess="0x00000002" Executed="YES" Object="REGISTRY\
     MACHINE\SYSTEM\ControlSet001\Services\SharedAccess\Parameters\
     FirewallPolicy\StandardProfile" ReturnValue="0x00000000" Timestamp="
     3121"/>
1212 <NtSetValueKey DesiredAccess="0x00000000" Executed="YES" Object="
     REGISTRY\MACHINE\SYSTEM\ControlSet001\Services\SharedAccess\
     Parameters\FirewallPolicy\StandardProfile" ReturnValue="0x00000000"
     Timestamp="3121" ValueName="EnableFirewall" ValueType="4"/>
1213 <NtOpenKey DesiredAccess="0x00000002" Executed="YES" Object="REGISTRY\
     MACHINE\SYSTEM\ControlSet001\Services\SharedAccess\Parameters\
     FirewallPolicy\StandardProfile\AuthorizedApplications\List"
     ReturnValue="0x00000000" Timestamp="3121"/>
1214 <NtSetValueKey DesiredAccess="0x00000000" Executed="YES" Object="
     REGISTRY\MACHINE\SYSTEM\ControlSet001\Services\SharedAccess\
     Parameters\FirewallPolicy\StandardProfile\AuthorizedApplications\
     List" ReturnValue="0x00000000" Timestamp="3122" ValueName="binary"
     ValueType="1"/>
```

Listing 5.9: Malware sample 1 - registry section

The report's registry section also contains many entries which have to considered. The process queries information about system and network services as well as about the operating system such as the system's version. One particularly noticeable part of the registry section is depicted in Listing 5.9. The report reveals that the target process uses registry API functions to change the value `EnableFirewall` (line 1212) and furthermore to add its executable file to the list of authorized applications (line 1214). These entries are clear evidence that the application tries to disable security mechanisms and thus performs malicious behavior unwanted by the infected system's user.

```
1709 <NtTerminateProcess Executed="YES" ExitStatus="30343168" ObjectPID="
     3435973836" ParentPID="3220" ReturnValue="0x00000000" Timestamp="
     3117"/>
1710 <NtTerminateProcess Executed="YES" ExitStatus="4227304" ObjectPID="
     3435973836" ParentPID="3220" ReturnValue="0x00000000" Timestamp="
     3117"/>
1711 <NtTerminateProcess Executed="YES" ExitStatus="3220" ObjectPID="
     3435973836" ParentPID="3220" ReturnValue="0x00000000" Timestamp="
     3118"/>
1712 <NtTerminateProcess Executed="YES" ExitStatus="30408767" ObjectPID="
     3435973836" ParentPID="3220" ReturnValue="0x00000000" Timestamp="
     3118"/>
```

Listing 5.10: Malware sample 1 - process section - Extract 3

Additionally, the report's process section lists unusual behavior, too. As documented in Listing 5.10, the process terminates four different processes. Although the monitored binary seems to query many pieces of information concerning sockets and other networking services the network section does not reveal any occurrences of network activity. A possible explanation for this aspect is that the network connection of the VM used for the analysis runs is disabled. The malware sample may have noticed this aspect and therefore not tried to connect to a remote server.

Malware Sample: Fake Spoolsv.exe

The second malware sample analyzed in this diploma thesis can also be categorized as trojan horse and backdoor. Information regarding this malware is provided by VirusTotal. Symantec [Syma] refers to the threat as *Backdoor.Trojan*. This type tries to provide remote access to the infected machine enabling an attacker to send commands and thus misuse the compromised system.

In order to provide more detailed information about the activities of this malware sample, we run a PyBox analysis. Since the threat's execution enforces a system reboot after a time-interval of 60 seconds, we have to reduce the time-out interval. Hence, we use a time-interval of 20 seconds for this analysis run.

In what follows, we examine various entries of the resulting XML report. As is the case with the first monitored threat, the report is too extensive to be depicted in this thesis. Therefore, we draw the reader's attention to certain particularly noticeable entries that allow us to draw conclusions about the examination object's behavior and indicate malicious code.

```
3 <calltree>
4   <process filename="binary" index="1" pid="3316" starttime="0">
5     <process filename="C:\WINDOWS\spoolsv.exe" index="2" pid="3336"
          starttime="1636"/>
6   </process>
7 </calltree>
```

Listing 5.11: Malware sample 2 - call tree

The first section to be considered is the report's call tree that is outlined in Listing 5.11. We notice that the analyzed binary executes another process `spoolsv.exe` (line 5). The file `spoolsv.exe` is known as the *Spooler Subsystem* of Windows and is responsible for print and fax jobs. The actual file provided by Windows is located in the system folder `C:\WINDOWS\System32`. The executed file, however, is located in the folder `C:\WINDOWS`. This is first evidence for malicious behavior of the target process. This aspect draws our attention to the process section of the process associated with the target executable `binary`, which is depicted in Listing 5.12.

```
620 <NtOpenProcess DesiredAccess="0x001f0fff" Executed="YES" Object="C:\
      WINDOWS\system32\spoolsv.exe" ObjectPID="1384" ParentPID="3316"
      ReturnValue="0x00000000" Timestamp="85"/>
621 <NtTerminateProcess Executed="YES" ExitStatus="0" ObjectPID="1384"
      ParentPID="3316" ReturnValue="0x00000000" Timestamp="87"/>
622 ...

631 <NtCreateProcessEx DesiredAccess="0x001f0fff" Executed="YES" NewPID="
      3336" Object="C:\WINDOWS\spoolsv.exe" ParentPID="3316" ReturnValue="
      0x00000000" Timestamp="1636"/>
```

Listing 5.12: Malware sample 2 - process section of `binary`

We can derive from this extract that the process `binary` terminates the original process `spoolsv.exe` (lines 620, 621) and creates a new process (line 631) using another executable file with the same file name located in the `C:\WINDOWS` folder. If we consider the timestamps of the mentioned entries we realize a significant time span of 1549 milliseconds between these events indicating that a lot of activity has been performed in between their function calls. Therefore, we proceed by taking a closer look at the target process's file management activities.

```
346 <NtCreateFile CreateDisposition="0x00000005" CreateOptions="0x00000064"
      DesiredAccess="0x40110080" Executed="YES" FileAttributes="0
      x00000020" Object="C:\WINDOWS\spoolsv.exe" ReturnValue="0x00000000"
      ShareAccess="0x00000000" Timestamp="1585"/>
347 <NtReadFile Executed="YES" Length="65536" Object="PyBox\Target
      \002049463441289680a35c595686d8784cbd0862-268496\binary" ReturnValue
      ="0x00000000" Timestamp="1589"/>
348 <NtWriteFile Executed="YES" Length="65536" Object="WINDOWS\spoolsv.exe"
      ReturnValue="0x00000000" Timestamp="1591"/>
349 <NtReadFile Executed="YES" Length="65536" Object="PyBox\Target
      \002049463441289680a35c595686d8784cbd0862-268496\binary" ReturnValue
      ="0x00000000" Timestamp="1592"/>
350 <NtWriteFile Executed="YES" Length="65536" Object="WINDOWS\spoolsv.exe"
      ReturnValue="0x00000000" Timestamp="1594"/>
351 <NtReadFile Executed="YES" Length="65536" Object="PyBox\Target
      \002049463441289680a35c595686d8784cbd0862-268496\binary" ReturnValue
      ="0x00000000" Timestamp="1595"/>
352 <NtWriteFile Executed="YES" Length="65536" Object="WINDOWS\spoolsv.exe"
      ReturnValue="0x00000000" Timestamp="1596"/>
353 ...
```

Listing 5.13: Malware sample 2 - file management section of `binary`

In Listing 5.13, the report reveals that the process creates another file `spoolsv.exe` (line 346). The flag `CreatePosition` is set to `0x5`, which represents the creation option `FILE_OVERWRITE_IF`, i.e. the file is overwritten and if it does not exist it is created. Subsequently, it reads data from its executable file and writes them to the created file. Thus, the process tries to hide the activity executed by the new `spoolsv.exe` process by pretending to be a system service.

Akin to this part of the report, it reveals also that four further files are created or overwritten in the Windows system folder showing the same activity pattern as is depicted in Listing 5.13: `explorer.exe`, `vcl32.exe`, `msbot32.exe` and `concp32.exe`.

Knowing that the created `spoolsv.exe` process is designed to perform hidden functionality, we also have to consider its activity. The report's entries regarding this process reveal much registry activity requesting system as well as socket information. For instance, a series of calls queries various catalog entries of the protocol catalog of Windows Sockets (line 109 ff.). Additionally, its file section reveals write access and read access to `lsass` (lines 34, 35). The latter is depicted in Listing 5.14. The *Local Security Authority Subsystem Service* (LSASS) is an important system service required for the authentification of systems in networks. It has also been a popular target of attacks (cf. Microsoft [lsa04]). In particular, the worm *Sasser* [Symb] exploited such vulnerabilities in the past.

```
34 <NtWriteFile Executed="YES" Length="72" Object="lsass" ReturnValue="0
       x00000000" Timestamp="2038"/>
35 <NtReadFile Executed="YES" Length="1024" Object="lsass" ReturnValue="0
       x00000000" Timestamp="2038"/>
```

Listing 5.14: Malware sample 2 - file management section of `binary`

Although we have focussed on small extracts of the analysis process's report, the monitored activity described in this section clearly indicate, that the executable is designed for malicious purposes. For more details on the monitored behavior, please refer to the entire XML report that can be found on the CD-ROM attached to this diploma thesis.

5.3. Summary

In this chapter, we have evaluated the functionality and abilities of PyBox. In Section 5.1, we have demonstrated the analysis process procedure applied in this chapter. Subsequently, we have detailed the implementation of a test environment that served as test target process 5.2.1. Thus, we have been able to verify PyBox's functionality. In Section 5.2.2, we have tested PyBox on two different malware samples and evaluated various sections of the output reports in order to examine the activity that PyBox is able to provide information about.

As a result, we can conclude that the created analysis environment very well entails the ability to monitor the executed API calls and report them in a comprehensible way. Furthermore, we have been able to extract valuable information from the analysis reports concerning the two analyzed malware samples and to derive behavior indicating malicious activity. Therefore, we can conclude that PyBox represents a fully functional analysis environment that can be applied to determine the activity of malware samples.

Chapter 6.

Conclusion and Future Work

The goal of this thesis has been to describe the implementation of a sandbox environment for the analysis of malware samples. In this chapter, we conclude this thesis and present our major findings. In Section 6.1, we provide a summary reviewing all contents of this thesis in order to provide an overview of what we have learned. Limitations regarding the described implementation of PyBox are denoted in Section 6.2. In Section 6.3, we outline several possibilities of how the work described in this thesis can be improved and extended. In Section 6.4, we discuss our overall findings.

6.1. Summary

During this thesis, we have focused on the development of a sandbox solution, which is used to analyze executables such as malware samples. At first, we introduced several fundamentals. We outlined different common malware types and described ways to detect them. Then, we provided basic information about the Windows operating system, which was required in order to implement the PyBox analysis framework. Finally, we described the concepts of API hooking and DLL injection as well as various implementation methods. Additionally, we discussed related work, in particular, CWSandbox and Joe Sandbox as well as a concurrent project which pursues the same goal as this thesis.

After introducing these prerequisites, we detailed the implementation of our sandbox solution. We divided the implementation into three parts: the hook library, the analysis tool, and the means of communication between them. We first explained the realization of the hook library which incorporates the hooking and monitoring functionality while focussing on the realization of the hooking functionality. Then, we studied the main part of the implementation, the analysis tool, which provides means of user interaction. In particular, we learned how the configuration files have to be configured and how the log information is processed and written to a report.

In addition to the implementation of PyBox, we also discussed the possibilities of porting the analysis environment to a Linux operating system. For this purpose, we outlined some fundamentals about the Linux operating system and its different manifestations. Furthermore, we described various techniques of function hooking in Linux. We also covered a possibility to implement library injection in Linux by applying the *injectso* tool and explained how the executable behavior can be monitored. As an alternative to implementing a hook library, we presented powerful tools, which allow one to trace

system calls and, thus, represent another possibility for the analysis of the behavior of an executable during runtime.

6.2. Limitations

In Chapter 5, we have shown that the created analysis environment works well and produces reports which can be applied to derive malware behavior. However, the described sandbox implementation is not a complete and perfect solution. It also entails some limitations that are described in the following.

The first limitation is associated with the hook library's focus on native API functions. As we have described in Chapter 3, we primarily hook native API functions instead of Windows API functions. The advantage is that we can also monitor the activity of malware utilizing the native API. The disadvantage is that the different Windows versions can use very different native API functions in order to accomplish the same purpose. For instance, the native API used to create a new user mode process varies from system to system: We have to use `NtCreateProcess` in In Windows NT and 2000, `NtCreateProcessEx` in Windows XP and 2003, and `NtCreateUserProcess` in Windows Vista and 7, while the corresponding Windows API `CreateProcess` can be utilized throughout all Windows versions. In order to provide compatibility with all different Windows versions, we have to provide callback implementations for each one of them. Thus far, PyBox is designed to be used in Windows XP. In order to offer compatibility with other versions the functionality of the hook library has to be extended.

The second limitation of the described implementation is a problem that occurs when we try to use the applied monitoring methods for the analysis of certain GUI applications such as `notepad.exe`. In such a scenario, the initiated target process immediately terminates after the injection of the hook library. Debugging of this incident reveals that these processes terminate with the Windows error message `CLASS_NOT_FOUND`. The reason for this message is that the initialization process of these GUI applications requires the existence of certain Windows classes. Since we create all monitored target processes in suspended mode in order to install the hook library before the program's execution, this initialization process is not started until the hook library has terminated its installation process. However, during the termination of the latter, the hook library checks whether or not there are any other running threads. As the hook library is the only running thread, the termination sequence deletes all Windows classes. Therefore, when the suspended main thread is resumed it cannot create the application's GUI due to the lack of the deleted classes. Therefore, PyBox cannot analyze applications that require the existence of these classes. However, malware samples usually do not use a GUI. Instead, they prefer to hide their presence. Therefore, this issue usually does not influence the functionality of PyBox.

Another point of criticism that might be established is that the developed solution described in this thesis is rather static and not easy to extend. In order to add further hook functionality, we have to change the code of the hook library by adding additional callback and trampoline functions and then by recompile it.

6.3. Future Work

As we have pointed out in the previous section, PyBox still has some limitations which can be addressed by future work. Various possibilities of extending and improving its functionality exist. As in what follows, we outline a few such approaches, which are beyond the scope of this thesis.

An approach for extending the functionality of PyBox is to add further hooks to the analysis framework. For this purpose, additional callback and trampoline functions have to be created as well as interpreters, which convert the log fields to an appropriate report format. Consequently, the analysis would be able to monitor more activity types of the observed executable and, thus, would provide more information.

Furthermore, the analysis tool's report generation can be extended by mapping provided numeric values such as file creation flags to strings that describe the meaning of the activated flags. Although this enhancement does not provide more information, it would simplify the interpretation of the monitored activity and therefore precipitate the analysis process.

A third approach is to implement a hook library for other operating systems such as Android. In this case, the analysis environment could also analyze malware samples that are designed for the mobile operating system and provide valuable information concerning the detection of malware threats and the protection of mobile devices.

Another approach addressing the limitation regarding the flexibility of PyBox would be to inject a Python interpreter into the remote process as described in the approach of Leder and Plohman [LP10] mentioned in Chapter 2.4.2. Thus, the analysis framework would be entirely Python-based. All callbacks would be handled by Python scripts. Since Python uses an interpreter, these scripts would not have to be compiled and, therefore, could easily be extended. This solution would offer more flexibility and extendability. However, when analyzing executables during runtime, we have to consider the important aspect of performance. For example, malware could notice the slow performance of a process and, as a consequence, might behave differently. Therefore, applying this approach requires us to ensure that the python scripts do not cause considerable drops in performance.

Another approach regarding flexibility and extendibility is to incorporate the functionality of a C compiler. We could use a special Python script and create the hook library's code out of existing code fragments according to the configured settings specified by the analyst. Thus, we would implement a modular solution that automatically compiles a separate, customized hook library that can be injected into the remote process to be monitored. In doing so, we would have a flexible solution, which also considers the performance criterion.

6.4. Conclusion

Ultimately, we conclude that we have provided a secure environment in which executables can be safely analyzed and that we have implemented an analysis framework that is able to monitor the most important API calls of an observed executable together with all relevant arguments. The resulting report provides detailed information in a categorized and neatly arranged way which makes it easy to draw conclusions about the activities of the monitored executable. In spite of several limitations, we have shown that the analysis environment can very well be applied to reveal critical behavior of executables in order to determine whether or not they represent a threat.

Appendix A.

Hooked API Functions

A.1. File System Interfaces

- **NtCreateFile (ntdll.dll):** Creates a new file or opens an existing file.
- **NtOpenFile (ntdll.dll):** Opens an existing file, directory, device, or volume.
- **NtWriteFile (ntdll.dll):** Writes data to an open file.
- **NtReadFile (ntdll.dll):** Reads data from an open file.
- **NtDeleteFile (ntdll.dll):** Deletes the specified file.

A.2. Registry Interfaces

- **NtCreateKey (ntdll.dll):** Creates a new registry key or opens an existing one.
- **NtOpenKey (ntdll.dll):** Opens an existing registry key.
- **NtSetValueKey (ntdll.dll):** Creates or replaces a registry key's value entry.
- **NtQueryValueKey (ntdll.dll):** Returns a value entry for a registry key.

A.3. Process Interfaces

- **NtCreateProcess (ntdll.dll):** Creates a new process (Windows NT, 2000).
- **NtCreateProcessEx (ntdll.dll):** Creates a new process (Windows XP, 2003).
- **NtOpenProcess (ntdll.dll):** Opens an existing process.
- **NtCreateThread (ntdll.dll):** Creates a new thread for an existing process.
- **NtTerminateProcess (ntdll.dll):** Terminates an existing process.

A.4. Network Interfaces

- **connect (wsock32.dll):** Establishes a connection to a specified socket.
- **send (wsock32.dll):** Sends data on a connected socket.
- **recv (wsock32.dll):** Receives data from a connected socket or a bound connectionless socket.
- **connect (ws2_32.dll):** Establishes a connection to a specified socket.
- **send (ws2_32.dll):** Sends data on a connected socket.
- **sendto (ws2_32.dll):** Sends data to a specific destination.
- **recv (ws2_32.dll):** Receives data from a connected socket or a bound connectionless socket.
- **recvfrom (ws2_32.dll):** Receives a datagram and stores the source address.
- **WSAConnect (ws2_32.dll):** Establishes a connection to another socket application, exchanges connect data, and specifies required quality of service.
- **WSASend (ws2_32.dll):** Sends data on a connected socket.
- **WSARecv (ws2_32.dll):** Receives data from a connected socket or a bound connectionless socket.

Bibliography

[Ayc06] John Aycock. *Computer viruses and malware*. Advances in information security ; 22. Springer, New York, NY, 2006.

[BDG11] Pedro Bueno, Toralv Dirro, and Paula Greve. *McAfee Threats Report: Fourth Quarter 2010*. McAfee Labs, 2011.
`http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q4-2010.pdf`.

[BMKK06] Ulrich Bayer, Andreas Moser, Christopher Kruegel, and Engin Kirda. Dynamic analysis of malicious code. *Journal in Computer Virology*, 2:67–77, 2006. 10.1007/s11416-006-0012-2.

[CE10] Francie Coulter and Kim Eichorn. *A Good Decade for Cybercrime*. McAfee Inc., 2010.
`http://www.mcafee.com/us/resources/reports/rp-good-decade-for-cybercrime.pdf`.

[Clo] Shaun Clowes. injectso.
`http://www.securereality.com.au/main.html`, Retrieved March 2011.

[CM10] Lúis Miguel Pinho Cláudio Maia, Lúis Nogueira. Evaluating android os for embedded real-time systems. In *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 63 – 70, Brussels, Belgium, July 2010.
`http://webpages.cister.isep.ipp.pt/~smp/OSPERT2010-Proceedings.pdf`, Retrieved March 2011.

[Dam] Joe Damato. Function hooking for osx and linux.
`http://timetobleed.com/slides-from-defcon-18-function-hooking-for-osx-and-linux/`, Retrieved March 2011.

[Eil05] Eldad Eilam. *Reversing : Secrets of Reverse Engineering*. John Wiley & Sons, 2005.

[Eng07] Markus Engelberth. Apioskop: Api-hooking for intrusion detection. Master's thesis, RWTH Aachen, September 2007.
`http://pi1.informatik.uni-mannheim.de/filepool/theses/diplomarbeit-2007-engelberth.pdf`.

[FAZ11] FAZ. *Merkel: Cyberwar so gefährlich wie klassischer Krieg*. Frankfurter Allgemeine Zeitung GmbH, 2011.
`http://www.faz.net/s/RubDDBDABB9457A437BAA85A49C26FB23A0/Doc~EDB330E8D55AE42CFB27B83C8B9985309~ATpl~Ecommon~Scontent.html`,

Retrieved February 2011.

[FMC10] Nicolas Falliere, Liam O Murchu, and Eric Chien. *W32.Stuxnet Dossier*. Symantec, 2010. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf.

[Gooa] Google. *Android Developer Guide*. http://developer.android.com/guide/index.html, Retrieved March 2011.

[Goob] Google. Google basics. http://www.google.com/support/webmasters/bin/answer.py?hl=en&answer=70897&rd=1, Retrieved April 2011.

[HB08] Greg Hoglund and James Butler. *Rootkits : Subverting the Windows Kernel*. Addison-Wesley, 6. print. edition, 2008.

[IEE] IEEE, The Open Group. *POSIX Certified*. http://posixcertified.ieee.org/, Retrieved March 2011.

[Iva02] Ivo Ivanov. *API hooking revealed*. The Code Project, 2002. http://www.codeproject.com/KB/system/hooksys.aspx.

[joe11] How does joe sandbox work?, 2011. http://www.joesecurity.org/products.php?index=3, Retrieved May 2011.

[KMPP07] Jim Keniston, Ananth Mavinakayanahalli, Prasanna Panchamukhi, and Vara Prasad. *Ptrace, Utrace, Uprobes: Lightweight, Dynamic Tracing of User Apps*. IBM, 2007.

[Lov07] Robert Love. *Linux system programming*. OReilly, Beijing, 1. ed. edition, 2007.

[LP10] Felix Leder and Daniel Plohmann. Pybox - a python approach to sandboxing. Master's thesis, University of Bonn, 2010. http://www.informatik.tu-cottbus.de/~spring2010/content/Spring5_01_Abstract_Plohmann.pdf, Retrieved April 2011.

[lsa04] Microsoft security bulletin ms04-011, 2004. http://www.microsoft.com/technet/security/bulletin/MS04-011.mspx, Retrieved April 2011.

[MCA00] John McHugh, Alan Christie, and Julia Allen. Defending yourself: The role of intrusion detection systems. *IEEE Software*, 17:42–51, 2000.

[Mic10] Microsoft. *Microsoft Portable Executable and Common Object File Format Specification*, September 2010. http://msdn.microsoft.com/en-us/windows/hardware/gg463119.aspx.

[Neta] Microsoft Developer Network. Calling convention - enumeration. http://msdn.microsoft.com/us-us/library/system.runtime.interopservices.callingconvention(v=VS.100).aspx, Retrieved March 2011.

[Netb] Microsoft Developer Network. Interprocess communications. http://msdn.microsoft.com/en-us/library/aa365574(VS.85).aspx, Retrieved March 2011.

[Oll08] Gunter Ollmann. The evolution of commercial malware development kits and colour-by-numbers custom malware. *Computer Fraud & Security*, 2008(9):4 – 7, 2008. http://www.sciencedirect.com/science/article/B6VNT-4THKK2D-6/2/d1b7419077cc7f05baad3076bbbbf146.

[Ope] The Open Group. *The Single UNIX Specification, Version 3*. http://www.unix.org/version3/, Retrieved March 2011.

[Ora11a] Oracle. *Oracle VM VirtualBox Manual*, 2011. http://download.virtualbox.org/virtualbox/UserManual.pdf, Retrieved March 2011.

[Ora11b] Oracle. *Oracle VM VirtualBox Programming Guide and Reference*, 2011. http://download.virtualbox.org/virtualbox/SDKRef.pdf, Retrieved March 2011.

[Pie94] Matt Pietrek. *Peering Inside the PE: A Tour of the Win32 Portable Executable File Format*. Microsoft, March 1994. http://msdn.microsoft.com/en-us/magazine/ms809762.aspx.

[Pie02a] Matt Pietrek. *An In-Depth Look into the Win32 Portable Executable File Format, Part 1*. Microsoft, February 2002. http://msdn.microsoft.com/en-us/magazine/cc301805.aspx.

[Pie02b] Matt Pietrek. *An In-Depth Look into the Win32 Portable Executable File Format, Part 2*. Microsoft, February 2002.

[Pre] Prevx. Cssrss.exe. http://info.prevx.com/aboutprogramtext.asp?PX5=3E583C9DC0F87CBE51EE002CBE6AE800476D2E00, Retrieved April 2011.

[Ric94] Jeffrey Richter. Load your 32-bit dll into another process's address space using injlib. *Microsoft System Journal*, 9(5), May 1994.

[Ric99] Jeffrey M. Richter. *Programming Applications for Microsoft Windows*. Microsoft Press, Redmond, WA, USA, 4th edition, 1999.

[Rob00] John Robbins. *DeBugging Applications*. Microsoft Press, Redmond, WA, USA, 2000.

[Roz03] Danny Rozenblum. Understanding intrusion detection systems, 2003. http://www.sans.org/reading_room/whitepapers/detection/understanding-intrusion-detection-systems_337, Retrieved April 2011.

[Rus06] Mark Russinovich. *Inside Native Applications*. Microsoft TechNet, 2006. http://technet.microsoft.com/en-us/sysinternals/bb897447.

[Sal10] David Salomon. *Elements of computer security*. Undergraduate topics in computer science. Springer, London, 2010.

[Sei09] Justin Seitz. *Gray Hat Python: Python Programming for Hackers and Reverse Engineers*. No Starch Press, San Francisco, CA, USA, 2009.

[Som11] Ravi Somaiya. *Hackers Shut Down Government Sites*. The New York Times, 2011.
http://www.nytimes.com/2011/02/03/world/middleeast/03hackers.html, Retrieved February 2011.

[Syma] Symantec. Backdoor.trojan.
http://www.symantec.com/security_response/writeup.jsp?docid=2001-062614-1754-99, Retrieved April 2011.

[Symb] Symantec. W32.sasser.worm.
http://www.symantec.com/security_response/writeup.jsp?docid=2004-050116-1831-99, Retrieved April 2011.

[Thr] ThreatExpert. Threatexpert report: Trojan.flush.g, hacktool.rootkit, downloader-biu.sys.
http://www.threatexpert.com/report.aspx?md5=3cdb8dc27af1739121b1385f36c40ce9, Retrieved April 2011.

[TIS95] TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*, May 1995.
http://refspecs.freestandards.org/elf/elf.pdf, Retrieved March 2011.

[Tor02] Linus Torvalds. libc-alpha@sources.redhat.com mailing list, 2002.
http://ecos.sourceware.org/ml/libc-alpha/2002-01/msg00079.html.

[Vij10] Jaikumar Vijayan. *WikiLeaks furor spawns rival DDoS battles*. Computerworld Inc., 2010.
http://www.computerworld.com/s/article/9200098/WikiLeaks_furor_spawns_rival_DDoS_battles, Retrieved February 2011.

[Vir] VirusTotal. Virustotal report.
http://www.virustotal.com/file-scan/report.html?id=9e9efb4321ffe9ce9483dc32c37323bada352e2dccc179fcf4ba66dba99fdebf-1233827064, Retrieved April 2011.

[WHF07] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security and Privacy*, 5:32–39, 2007.

[Whi11] Sophos Whitepaper. *Security Threat Report 2011*. Sophos Ltd., 2011.
https://secure.sophos.com/sophos/docs/eng/papers/sophos-security-threat-report-2011-wpna.pdf.

[WJCN09] Zhi Wang, Xuxian Jiang, Weidong Cui, and Peng Ning. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 545–554, New York, NY, USA, 2009. ACM.