TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

Master's Thesis

Understanding and Replaying Network Traffic in Windows XP for Dynamic Malware Analysis

carried out at the
Institute of Computer Aided Automation and
at the Information Systems Institute
Technical University of Vienna

under the guidance of

Doz. Dipl.-Ing. Dr.techn. Christopher Krügel
Univ.Ass. Dipl.-Ing. Dr.techn. Engin Kirda

by

Helmut Petritsch, Bakk. rer. soc. oec.
Bräunlichgasse 7
2700 Wr. Neustadt

Vienna, February 13, 2007

Kurzfassung

Die Analyse von unbekannten, möglicherweise böswilligen, ausführbaren Dateien wird als “Malware Analyse” bezeichnet. Dies ist allerdings immer eine Gratwanderung zwischen einer möglichst hohen Genauigkeit der Analyseergebnisse und der gleichzeitigen Anforderung, dies mit einem möglichst geringem Aufwand zu bewerkstelligen. Um diese Aufgabe möglichst effizient gestalten zu können, werden Hilfsmittel (“Tools”) benötigt, die einerseits eine detaillierte Analyse ermöglichen, andererseits aber nur signifikante Informationen präsentieren, um Analysten rasch einen guten Überblick über die Funktionsweise des Testobjekts bzw. dessen Kernbereiche, also zum Beispiel den Replikationsmechanismus, zu ermöglichen.

Die beiden gebräuchlichsten Methoden - statische und dynamische Analyse - haben beide jeweils ihre Stärken und Schwächen. Eine statische Analyse kann durch “obfuscation techniques” (Methoden, die das Disassemblieren von Maschinencode empfindlich erschweren) nahezu unmöglich gemacht werden. Mit Hilfe der dynamischen Analyse ist es schwer festzustellen, wie sich das Testobjekt unter wechselnden Umständen verhält (z.B. Benutzereingaben, Systemzeit, das Vorhandensein und Nicht-Vorhandensein bzw. der Inhalt von bestimmten Dateien, Verfügbarkeit bzw. Status von Netzwerkressourcen, etc.). Um mit Hilfe der dynamischen Analyse unterschiedliche Ausführungspfade zu extrahieren, muss die Interaktion des Testobjekts mit seiner Umgebung manipuliert werden, womit indirekt das Testobjekt selbst manipuliert wird. Wie diese Interaktion stattfindet ist im Wesentlichen gut dokumentiert - mit einer Ausnahme: der Zugriff auf netzwerkbasierte Ressourcen.

In dieser Diplomarbeit wird beschrieben, wie von Usermode Programmen auf (socketbasierte) Netzwerk Ressourcen auf system-call Ebene zugegriffen wird, wie diese Zugriffe einerseits beobachtet, andererseits manipuliert und - für das Testprogramm nicht erkennbar - imitiert werden können. Außerdem wird beschrieben, welche Möglichkeiten multithreaded Programme zur Synchronisation der Threads haben und wie diese Möglichkeiten ebenfalls imitiert werden können.

Abstract

Malware analysis is the process of extracting the behaviour of an unknown executable. This task is always a trade-off between the effort invested and the accuracy of results. To achieve high efficiency, tools should provide only the relevant actions of the program. The goal is to quickly help the analyst find and understand the core functionality (e.g., how the exploit or the replication mechanism of a virus is implemented).

The two most common techniques for analyzing unknown executables - static and dynamic analysis - have both advantages and drawbacks: a static analyst has to face the fact that there are many obfuscation techniques, making it difficult to extract the core functionality. For dynamic analysis, it is difficult to determine how the executable would behave under different circumstances and in a different environment (e.g., user input, system time, existence or non-existence of certain files and their contents, availability and interaction with network resources). To determine different execution paths with dynamic analysis, the interaction with the environment could be manipulated, and with it the test subject. This interaction is mostly well-documented and relatively easy to track, with one exception: networking.

In this thesis, I describe my research on how user-mode programs under Windows XP use network resources via sockets on the system-call level, how the communication via sockets can be intercepted, manipulated and imitated for a dynamic analysis, and how multi-threaded applications can synchronize their (network) activities.

Acknowledgments

This master's thesis is based on the work of Ulrich Bayer, who has been very supportive and helped me understand and extend his work, even after he finished his diploma and started to work.

Moreover, I want to thank my advisors Engin Kirda, and Christopher Krügel first for the courses “Internet Security” and “Internet Security 2”, which were among the most interesting I took at university and sparked my interest in IT Security. Furthermore, I like to thank them for their constant patience and their support.

Contents

1 Introduction **1**
1.1 Structure of this Thesis 2
1.2 Terminology 2
1.3 Goals 3

2 Basic Concepts **4**
2.1 Malware 4
2.2 Malware analysis 5
2.2.1 Static Analysis 5
2.2.2 Dynamic Analysis 6
2.3 TTAnalyze 7
2.3.1 Monitoring the Dynamic Behaviour 7
2.3.2 System Architecture 8

3 The Windows Operating System **11**
3.1 User Mode vs. Kernel Mode 11
3.2 Architecture Overview 12
3.3 System Calls 13
3.4 Objects and Handles 14
3.5 Useage of System Calls and Object Handles 14
3.5.1 NtCreateFile() 15
3.5.2 NtWriteFile() 16
3.6 Tools for Analyzing Windows 16
3.6.1 WinDbg 16

4 Networking under Windows **21**
4.1 OSI Reference Model 21
4.1.1 Layers 21
4.2 Protocols 24
4.2.1 IP 24
4.2.2 TCP 24
4.2.3 UDP 26
4.2.4 ICMP 26

4.3 Implementation of Networking in User Mode 27
4.3.1 WSAStartup() 28
4.3.2 socket() 28
4.3.3 gethostbyname() 29
4.3.4 connect() 29
4.3.5 send() 30
4.3.6 recv() 30

5 Manipulating the Test Subject 31
5.1 Manipulating System Calls 32
5.1.1 Hooking System Calls 32
5.1.2 Manipulating the Result of System Calls 33
5.1.3 Virtualization of System Calls 33
5.2 Reseting Network Connections to a Previous State 33
5.2.1 Where to Manipulate the System 34
5.2.2 Where to Virtualize the Network in User-Mode 35

6 Networking at System Call Level 37
6.1 System Calls for Networking 37
6.1.1 NtDeviceIoControlFile() 37
6.1.2 NtWaitForSingleObject() 39
6.1.3 NtClose() 40
6.2 Chronology of AFD Control Codes 40
6.2.1 socket() 40
6.2.2 connect() 41
6.2.3 send() and recv() 41
6.2.4 Error handling 41
6.3 IoControlCodes for AFD 42
6.3.1 AFD_BIND (0x12003) 42
6.3.2 AFD_CONNECT (0x12007) 42
6.3.3 AFD_SEND (0x1201F) 43
6.3.4 AFD_RECV (0x12017) 43
6.3.5 AFD_SELECT (0x12024) 44
6.3.6 AFD_UDP_SEND (0x12023) 45
6.3.7 ICMP 46
6.4 Synchronisation 46
6.4.1 Waiting for Event Objects 46
6.4.2 I/O Completion 46
6.4.3 Replaying Synchronisation 48
6.5 Local Procedure Call (LPC) 49

7 Implementation 50
7.1 Hooking Network Access 50
7.1.1 Changes of the Generator 50
7.1.2 Manual Decoding Function Parameters 51
7.1.3 LPC - DNS 54
7.2 Virtualising Network Access 55
7.2.1 Changes of TTAnalyze for Virtualising 56
7.2.2 Record Mode 57
7.2.3 Replay Mode 60
7.2.4 Simulation Mode 62

8 Evaluation 64
8.1 Mode of Operation 64
8.2 Verification 66

9 Future Work and Conclusion 68
9.1 Future Work 68
9.2 Conclusion 68

List of Figures

3.1 System Module Dependencies [1] 12
3.2 Screenshot from WinDbg, debugging the Internet Explorer 17

4.1 Schema of the OSI Reference Model 22
4.2 Implementation of TCP/IP in the OSI Reference Model [2] 25
4.3 The TCP three-way handshake [2] 26

Chapter 1

Introduction

Security is becoming a more and more relevant topic. Even Microsoft has recognised the increasing relevance of security[1]. Campaigns such as "Month of Browser Bugs (MoBB)" [3] demonstrate in an impressive way that software vendors such as Microsoft are not able to keep track with the increasing numbers of exploits. Obviously, not only Microsoft has such problems; the more famous a software becomes, the more interesting target it becomes for crackers[2]. For example, as the popularity of Mozilla Firefox is growing, the more vulnerabilities are published [4], although it seems to be more simple to find vulnerabilities in Microsoft's Internet Explorer - the "MoBB" reported 25 out of 31 bugs for this browser.

Most users feel uncomfortable with this situation because they do not possess the necessary knowledge about how to use resources such as the Internet in a safe way[3] or to recognise an attack they are a target of. Therefore, security aware users use third party products like virus scanners, anti-spyware tools, personal firewalls or tools for detecting potential malicious software, such as Hijackthis [5]. Obviously, scanners for detecting malware need to know the malware samples they have to detect. Technically spoken they need a signature of a malware sample to identify these malware samples with an adequate certainty[4]. Therefore, it is essential for the vendors of such scanners to collect and analyze new malware samples to be able to provide up to date signatures for their customers.

s execution. Furthermore, we wish to reset the test-executable in user-mode only, not in kernel mode (and with it the whole system), mainly because of performance reasons.

The goal of this thesis is to describe, how TTAnalyze was enhanced with the ability to

[1]This can be observed with e.g.,the new operating system Windows Vista: in contrast to Windows XP, in Windows Vista new users are not per default system administrators; furthermore, by the implementation of UAC (User Account Control) for reducing the rights of processes, thus, e.g., the Internet Explorer has less rights than the user executing it

[2]I explicitly use "cracker" and not "hacker", because a "hacker" only tries to use a system in a way that it was not designed for, whereas a "cracker" will use systems in a way they were not designed for to harm others and profit himself

[3]e.g., not to surf as system administrator at unknown web pages using a vulnerable browser

[4]There are approaches (e.g., heuristic analysis) to identify malware without such signatures, but these have currently to much drawbacks (e.g., high rates of false positives) to solve this problem in general [6]. Although, for some areas (e.g, macro viruses) heuristic analysis is essential for modern scanners

virtualize network traffic. Furthermore, some documentation of undocumented windows internals that have been observed during the implementation of these enhancements is provided.

1.1 Structure of this Thesis

Chapter 2, "Basic Concepts", provides basic concepts that are necessary for further reading: the terms malware and malware analysis are introduced. Additionally, a short description of TTAnalyze is presented.

Chapter 3, "The Windows Operating System", covers an introduction to selected topics referring to the Windows operating system such as system calls, objects and handles. With a short example, it is shown how a simple user mode program implements its core functionality with system calls . At the end of this chapter a short introduction to WinDbg, the Windows Debugger, is given.

Chapter 4, "Networking under Windows", presents the basics for networking under Windows. A short description of the OSI model and protocols discussed later in this thesis are given. Again, with a short piece of code an introduction is given how networking is implemented under Windows with the aim of Winsock.

Chapter 5, "Manipulating the Test Subject", discusses the different possibilities, where a test-subject executed in TTAnalyze could be manipulated, so that a changed environment for this test-subject can be virtualized. For networking it is discussed, where in user-mode such a manipulation could take place.

Chapter 6, "Networking as System Call Level", describes how networking is implemented at system call level. This includes a description of the used system call, an example how a request/response is handled down to system call level, and a detailed description of the most relevant control codes used by AFD.

Chapter 7, "Implementation", describes the most interesting parts of the implementation and how specific problems were solved.

1.2 Terminology

Virus The term virus is mostly used as equivalent to Malware, but we use this term like the classic definition by Frederick Cohen.

> "A *computer virus* is a program that can infect other programs by modifying them to include a possibly evolved copy of itself... A virus need not be used for evil purposes." [7]

Malware is a term used general for software that has a functionality that the user is not aware of or does not wish. This term will be discussed detailed in Chapter 2.1. In contrast to a Virus, Malware must not modify or infect other programs, instead it can be, e.g., a stand-alone program.

Rootkit Rootkit technologie can be used by any program, in particular by malicious programs (i.e., by malware in general and therefore by viruses too).

> "Rootkits are not, in and of themselves, malicious. However, rootkits can be used by malicious programs . . . A rootkit is a set of programs and code that allows a permanent or consistent, undetectable presence on a computer." [8]

Test-subject The term test-subject references the executable file, that is executed in and analyzed by TTAnalyze.

Handle is a 32 bit integer that acts as user-mode representation for a kernel-mode object.

1.3 Goals

The basis for this work is TTAnalyze [9], a tool for dynamic malware analysis. TTAnalyze executes and analyses portable executable (PE) files [10] in an emulated environment.

To make dynamic analysis more sophisticated, we wish to extract the behaviour of the executable under different conditions, i.e., how does the executable behave if the in- and output in the broadest sense (e.g., reading from files, network access) change. For this, we manipulate the test subject and test how it behaves under different circumstances. To do many such tests in a relative short amount of time, we need a facility to reset the test-subject to a previous state in it

Chapter 2

Basic Concepts

In this chapter an introduction to topics this thesis is based on is given, such as malware, malware analysis, and TTAnalyze.

2.1 Malware

Malware (short for Malicious Software) is a general term for software with malicious intents, such as misusing resources of an attacked computer for, e.g., sending spam or spreading itself, stealing information such as credit card numbers, etc. Malware can be divided in sub categories such as viruses, worms, spyware, trojan horses, etc, but not all malware samples can be mapped exactly to a single category, because their behaviour and goals are part of several categories. Malware is an increasing threat, the number of new samples that are "released" every day get more and more. Thus, for virus scanner vendors it is more and more difficult, to keep pace with new malware samples.

On the other ("crackers") side, it is relative easy to create malware without a deep or even any understanding of the used technologies. There are many code generators and "toolkits" available, which create new malware without that the "user" of these tools ever has to write even a single line of code. These tools have different levels of sophistication - more advanced ones use technologies like rootkits, self-modifying code, etc.

It is some kind of strange luck that most of the malware samples are created by these toolkits[1]. Many of these malware samples can be associated with a particular toolkit, and thus, it is more easy to create a signature, which can be used by virus scanners to detect this malware.

First of all, it is important to collect samples of a new type of malware and to recognise as early as possible how dangerous and "infective" a new malware sample is. The scope goes from samples that are not able to survive in the wild up to sample that cause an major "break out". For these requirements, the most effective solutions are honey pots. Honey pots simulate vulnerable systems so that malware is able to attack these systems. First, new malware samples can be collected this way. Furthermore, the number of attacks

[1]and not all malware samples are written by high-sophisticated crackers

from a new malware is a good indication about how dangerous this new sample is, and therefore, honey pots can help to install "early-warning-system" for major breakouts.

The next step is to generate a signature of the new malware sample which is needed by virus scanners to detect and eliminate running and/or attacking malware. This task should be finished as quickly as possible, because new appearing malware samples are, in particular at major out-breaks, reproducing themselves aggressively. Therefore, the probability to become a target is high. Additionally, the more PC's are protected, the less the malware spreads.

For the task of creating a signature it is essential to know what the malware sample is doing, e.g., how the replication mechanism is implemented, how the attack on the system is accomplished, which modifications of the system are done etc. This task is named "malware analysis". There are different approaches to determine the behaviour of an unknown executable. These are discussed in the next section.

Obviously, even after a signature has been created, there are more challenges to be solved. These signatures have to be distributed, particular modem users have problems with the increasing traffic of signatures, caused by the increasing numbers of malware. Furthermore, malware scanner vendors have to improve the performance of the scanning algorithms, because the databases are growing faster and faster.

2.2 Malware analysis

As described above, malware analysis is needed to generate signatures, which are needed to detect and eliminate malware. The results of malware analysis can also be used to find and fix vulerabilities of attacked software. The manual analysis of malware is a tedious task, therefore tools exist that help the analyst to do his job faster. The difficulties for such tools is to reduce the information to the most relevant one. These tools can be distinguished by their approach, how the analysis is done: static and dynamic analysis. These two approaches are described in more detail in the next two sections.

2.2.1 Static Analysis

The static analysis determines the behaviour of an executable without executing it. To perform such an analysis, the PE (portable executable) file is needed, so that it can be reverse engineered (i.e., the assembly code is reviewed). One of the most famous tools for doing this is IDA Pro [11].

This technique has some major problems:

- Assembly code is not very readable. There are no syntactic structures like in high-level programing languages such as conditional statements, loops, functions[2], variables, etc.

[2]As long as functions are not exported, which is in particular for malware rarely the case

- It is difficult to determine, which parts of a PE file are used frequently, which are used only rarely (e.g., error handling), or which parts are the most important one, i.e., which contain the replication mechanisms, exploits etc.
- If techniques like self-modifying code or encryption are used, the code is not available from the beginning but has to be discovered in a tedious way.
- Techniques such as code obfuscation make it nearly impossible to reassemble the complete executable in one single step. Code obfuscation inserts e.g., junk data, and many conditional and unconditional jumps which cannot be resolved without keeping track of the runtime status of the executable.

2.2.2 Dynamic Analysis

For a Dynamic analysis the test-subject is executed and it is tried to determine how the running executable affects its environment, i.e., which actions are executed. The problem of dynamic analysis is, how this is done. There are mainly two approaches:

- The status of a system is saved, before the test-subject is executed. After the execution, the actual status of the system is compared with the saved one. Therefore, with this approach only the changes in the system can be determined, and not how the changes have been achieved. Consider a virus that exploits an unknown vulnerability: with this analysis approach it is impossible to discovered how this exploit is working. Furthermore, there are performance restrictions, which result from a high effort for saving and comparing complete system stati.
- The test-subject is executed in a (mostly emulated) system, and all actions are monitored. This is the approached used by TTAnalyze, so it is discussed later in more detail.

Executing unknown executables in an emulated environment is not a new idea: This is the technique behind the the most powerful weapon of virus scanners: emulation [6]. Virus scanners use a “lightweight system” because they have stringent performance restrictions. Obviously, this approach in general has some problems too:

- It is impossible, to create an emulated environment that behaves exactly like a real system. In the emulated systems used by virus scanners it is more easy for the executed subject to detect that it is running in an emulated environment than in TTAnalyze. Obviously, even in a more advanced, full fledged system such as QEMU it is possible for the test-subject to detect such an emulated environment.
- With this approach, only one possible execution path can be analysed. If some parts of the malware are only executed under certain circumstances (e.g. the replication mechanism is only started at the first of every month), they cannot be analyzed.

2.3 TTAnalyze

As already mentioned, this thesis is based on TTAnalyze, which was developed at Secure Systems Lab at the TU Vienna, mainly by Ulrich Bayer. For a better understanding of this thesis, a short description of this tool is given, so that the reader is able to understand the extensions to this tool[3].

TTAnalyze analyzes PE files with a dynamic approach. The test-subjects are executed, but not in a "real" system. This would be a dangerous task, because the malware could infect the test system or use the test system for spreading. TTAnalyze uses an emulated system for the execution, where the test-subject is executed by a virtual processor in a virtual system. This virtual system can be controlled and reset to the original state after the analysis. Thus, every new test-subject is executed in a "clean" system.

Obviously, the execution in an emulated system is not as efficient as on a real processor, but in the emulation we have full control over the system and in particular the test-subject. The system can be stopped at any state of execution to take a deeper look at the current state of the system. For this, the memory and the CPU states can be read, and even a manipulation of the current state is possible. Obviously, emulation does not have only advantages. As already mentioned, the execution in an emulated system is much more time consuming than in a real system. The emulation software used by TTAnalyze is QEMU, which reduces the performance of execution approximately by a factor 10. Furthermore, an emulated system will never behave as a real system. It would be difficult to correctly implement all operation codes a CPU is supporting.

Emulation is not a new idea. Virus and malware scanners use this technique for a so called heuristic analysis, to get a basic understanding what an executable is doing. Obviously, out of performance restrictions, they cannot execute it completely (e.g., until it is terminating) in a full system. For such an emulation there a lightweight systems used that implement a rudimentary subsystem.

2.3.1 Monitoring the Dynamic Behaviour

With emulation we have a powerful instrument to execute any executable in a secured environment. For a dynamic analysis, we determine how the executable behaves and how it interacts with its environment. In TTAnalyze, all actions are monitored via a defined set of function calls. For finding this set of functions so that all actions can be monitored, there are two contrary approaches[4] - hooking "high level" vs. "low level" functions:

Hooking documented API functions

The first possibility would be, to hook the exported programming API by Windows, which would include many functions. The main advantage for this high level approach is that

[3]For a detailed description, download the paper from http://seclab.tuwien.ac.at/publications.html

[4]It is theoretically possible, to find approaches between these two, but they would only have more disadvantages of the two quoted ones and less advantages

the function that would have to be hooked are well documented. Obviously, there are fundamental disadvantages:

- Not all system calls provided by the Windows Kernel and therefore available to user mode programs are exported by the documented Windows API, e.g., the system calls for LPC[5] are not directly exported (although they are used indirectly by some API libraries, e.g., Winsock).
- Malware does not have to use high level functions. If the test-subject is using more low-level functions, it could bypass the hooking mechanism. Additionally, if the test-subject uses system calls that are not exported by high-level functions, it would bypass the hooking mechanisms too.
- There are many APIs with many functions available for user-mode programs. Monitoring all these functions would result in a high coding effort.

Hooking Low Level functions

To avoid the disadvantages of the approach discussed above, more low-level layered functions could be monitored. The lowest level in user-mode that can be hooked are the functions exported by `ntdll.dll`. This DLL is a layer for system-calls, to provide system-calls as "normal" function calls and therefore making them available in languages such as C/C++ without inline assembler. Thus, by hooking functions exported by `ntdll.dll` we are almost hooking the system calls itself.

Obviously, hooking functions at this low level has one important disadvantage: the functions from `ntdll.dll` are not documented officially. For most functions, including e.g., reading and writing from and to files, editing the registry, starting processes, etc., there exists some unofficial documentation [12]. Unfortunately, for other essential tasks such as networking, there does not exist any documentation about how it is implemented with system calls.

2.3.2 System Architecture

TTAnalyze is made up of several modules. QEMU is responsible for the system emulation, InsideTM implements the communication between the virtual system and the system around the virtual system, the Generator creates code, which is responsible for reading the function parameters from the stack, and the Analysis Framework interprets the informations, gained by the rest of the modules.

Qemu

QEMU is a *machine emulator: it can run an unmodified target operating system (such as Windows or Linux) and all its applications in a virtual machine* [13]. For TTAnalyze there

[5]Local Procedure Call for interprocess communication. This technique is discussed in section 6.5

have been introduced some modifications such as a packet filter (so that it is possible to control which target should be available for the test-subject, because we do not wish to spread malware in an uncontrolled way), and a call back mechanism to TTAnalyze. QEMU is used in the same process as TTAnalyze, so it has been transformed from an executable to a shared library (DLL). TTAnalyze uses functions exported by this DLL to determine the behaviour of the virtual system.

QEMU boots from a virtual hard disk that is saved as file on the workstation executing TTAnalyze. For a quick startup the virtual operating system is not booted for every analysis, instead the state of a booted system is saved in a snapshot (which represents the current state of the RAM of the virtual system, the CPU state, and modifications to the hard disk), which can be loaded quickly.

InsideTM

InsideTM stands for "Inside The Matrix" and refers the parts of TTAnalyze running inside the emulated system. InsideTM is a bridge between the the emulated system and the parts outside of the emulated system.

InsideTM consists of two parts. The first part is a RPC Server waiting for requests from TTAnalyze in an endless loop. This RPC connection is used for, e.g., file up- and downloads, starting processes inside the emulated system, etc. Therefore, the RPC Server is responsible for loading and starting the test subject.

The second part is a driver that runs inside the kernel of the emulated system and therefore has access to all resources. During execution, the analysis is restricted to the test subject - all other processes are not monitored. The differentiation between different processes should be done quickly, because it has to be done often (with every translation block). For the distinction of the target process out of all other processes, the PDBR (Page Directory Base Register) is used. The PDBR contains the address of the page directory for each process in the kernel, thus, it is unique for each process. The considerable advantage of this technique is that the page directory base address is saved in the first 20 bits of the `CR3` register (thus, the page directory must be aligned to 4KB boundaries) [14], so it can be accessed quickly.

After starting the test subject, only the process ID of the process is known (which is unique too, but it would require a more complex algorithm to determine the process ID of the currently running process). The driver is responsible for getting the PDBR from the process ID.

Generator

Reading the function parameters of a hooked function from the stack is simple, as long as only simple data types (e.g., integers, floats, etc.) are used. This is not always the case, because there are pointers, structures, which may recursively contain pointers, structures and so on. The generator is a framework that helps to avoid writing much code resolving these references.

As input the Generator needs the function declarations and a definition of all structures that are contained in these declarations. Therefore, the Generator is executed during build-time and generates code that is compiled to run during the execution of TTAnalyze. The generated code reads the function parameters with its whole structures (as long as these these can be defined statically) from the virtual system and provides these informations to the analyzing part. The Analysis-Framework is notified about hooked function calls and all function parameters are submitted.

Analysis-Framework

The Analysis-Framework is the core part of TTAnalyze. It keeps track of all functions that are called and generates out of this information a report. Therefore, all the other parts only exist to support this module.

The Analysis-Framework is divided into several parts, whereas every part is responsible for a defined field of activity, e.g., the File-Analyzer for all file activities, the Network-Analyzer for all activities corresponding to network, etc. These analyzers request notifications of function calls they need to keep track of, e.g., the File-Analyzer of `NtCreateFile()`, `NtWriteFile()`, etc., the Network-Analyzer of `NtCreateFile()`, `NtDeviceIoControl-File()`, etc. At the analysis end (i.e., if the test subject terminates or a defined timeout occurs) the Analyzers are requested to generate a report out of their collected data.

Chapter 3

The Windows Operating System

Giving a full introduction to the Windows Operating System is not possible on these few sides, but we will give a short instruction to those parts that will be needed to understand the rest of this thesis (there are many good books offering official and unofficial documentation [1, 12, 15, 16]).

3.1 User Mode vs. Kernel Mode

Like in most operating systems, applications run in a lower privileged mode than the operating system itself. Windows is using two different processor access modes (although most processors windows is running on support more than these two), to protect the critical operating system data. These two access modes are called user mode and kernel mode. As the name reveals, the Windows kernel runs in kernel mode, managing all resources, granting or rejecting access to these resources for user mode programs. The kernel is responsible for managing all resources such as memory, access to network and files, etc. The managing of devices such as network adapters, graphic cards, etc. is done by so called device drivers. These have to run in kernel mode, because they access the devices, which is not granted from user mode. Not even device drivers access the devices directly, this is done via the HAL (Hardware Abstraction Layer).

Programs are mainly running in user mode. If they need access to any resource, they have to call the system to provide the needed resources. For such an system-call, the CPU switches to the more privileged level, so that the requested operation can be executed. For switching to the more privileged kernel mode, a well defined call-gate has to be passed, which controls the actions to be executed. The kernel itself checks, if the calling program is allowed to execute the requested action, thus, there is no way to switch to the more privileged level bypassing this check by the kernel.

3.2 Architecture Overview

Windows was designed originally for running different subsystem, i.e., POSIX, OS/2 (removed with Windows 2000), and Windows. The access of system resource is layered *through one or more subsystem dynamic-link libraries (DLLs). The role of the subsystem DLLs is to translate a documented function into the appropriate internal (and generally undocumented) Windows system service calls* [15]. These system service calls are implemented in the `ntdll.dll`, and there is some unofficial documentation for most of the functions, exported by this DLL [12]. There are some system calls that are not handled by the `ntdll.dll`, but all of these are for graphical programming purposes. Thus, for analyzing the behaviour of a malware these are not interesting.

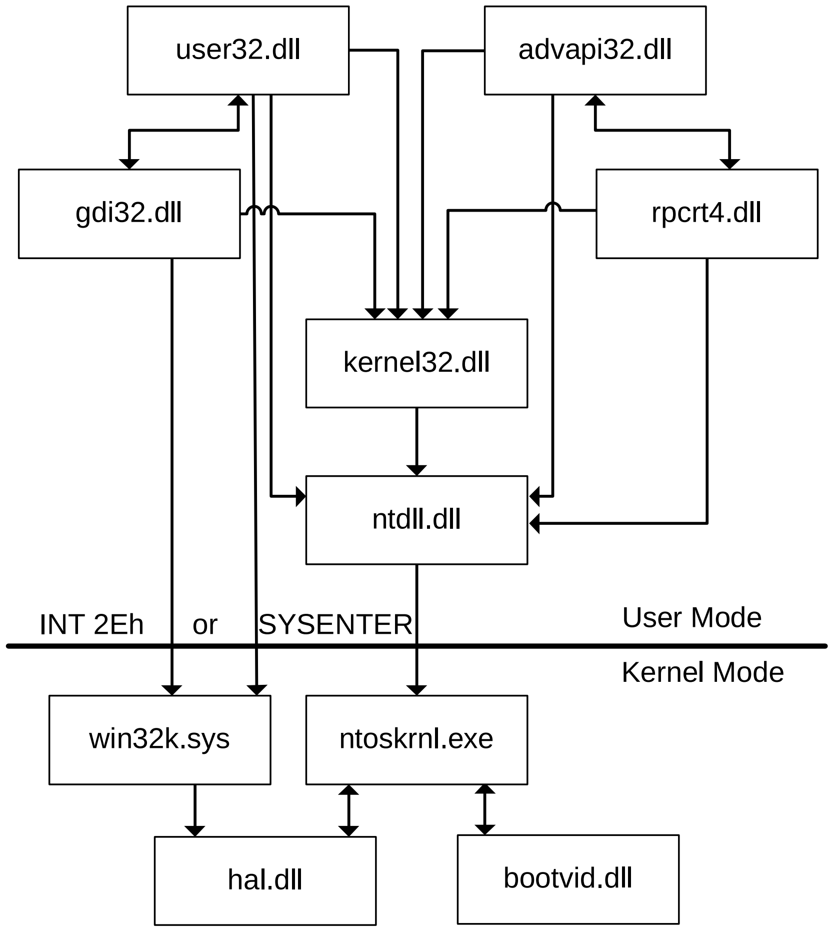


Figure 3.1: System Module Dependencies [1]

The DLLs `kernel32.dll`, `advapi32.dll`, `user32.dll`, and `gdi32.dll` are the core Windows subsystem, the exported functions are well documented, e.g., at the MSDN web pages [17]. Microsoft tries to keep these interfaces as constant as possible, so that it should be as simple as possible to transform Windows applications from one version to another (or even from one service pack to another). In more low-level DLLs undocumented changes are possible and therefore, these should not be used by user-mode programs.

Function calls accessing files, threads, networking etc. are passed into the kernel via `ntdll.dll`, for graphics there are faster solutions, thus, `gdi32.dll` is able to call the `win32k.sys` directly, without using the general solution over `ntdll.dll`.

The `ntdll.dll` is the most interesting one for our purposes. This DLL implements the system-calls and out of reasons discussed later, this is the best interface for placing hooks monitoring the behaviour of user-mode programs. There is no official documentation for this DLL, but [12] provides some documentation for the `ntdll.dll` under Windows 2000. For the main parts, this documentation can be used for Windows XP too. As mentioned above, for low-level DLL's such as `ntdll.dll` there is no guarantee that the interface keeps constant between operation system versions and even service packs and that the behaviour of all functions keeps the same.

`ntoskrnl.exe` contains the main part of the operating system: the executive, which is responsible for memory management, process and thread management, security, I/O, networking, interprocess communication and the kernel, containing the low-level functions for e.g. thread scheduling, interrupt and exception dispatching, synchronization etc.

Furthermore, `ntoskrnl.exe` is responsible for loading device drivers, managing the access to the used hardware. This hardware is not accessed directly, but through a additional layer, called the HAL (Hardware Abstraction Layer), implemented in the `hal.dll`.

3.3 System Calls

The term system-call has been used in the preceding sections, without defining it in more detail. To execute such a system-call, there are two possibilities:

- Executing a software interrupt: `INT 2Eh`. This is a relative slow approach, used on older CPUs that do not support the second approach
- Executing the `SYSENTER` instruction. This approach is efficient, because there is no interrupt that has to be executed and handled.

In both cases, the `eax` register is filled with a number, representing the function to execute[1]. This number is an index in the Interrupt Descriptor Table (IDT), containing function pointers to the corresponding kernel mode functions, which are called after the switch to kernel-mode. The kernel copies the function parameters from the user-mode stack to only from kernel-mode accessible memory (thus, the memory the kernel is working on cannot be manipulated from user-mode, which could be exploited from malicious code).

Such a system-call is now explained with a short example function from `ntdll.dll` - `NtDeviceIoControlFile()`, which will be discussed in more detail later. Reassembling this function reveals the following assembler code:

[1]This numbers depend on the used system (Windows 2000, XP) and on the build version

```
  ntdll!NtDeviceIoControlFile
1   mov      eax,42h
2   mov      edx,offset SharedUserData!SystemCallStub (7ffe0300)
3   call     dword ptr [edx]
4   ret      28h
```

Listing 3.1: Reassembled function `ntdll!NtDeviceIoControlFile()`

In line 1, the number for the called function is loaded to the `eax` register, which is `0x42` for `NtDeviceIoControlFile()`. In line 2, the address of the system-call stub is loaded to the `edx` register. This function pointer is saved in the symbol `SharedUserData!SystemCallStub`, which is in this case at address `0x7FFE0300`. This variable is filled at system startup, depending on the system Windows is running - for systems supporting the `SYSENTER` instruction it is a pointer to `ntdll!KiFastSystemCall`.

```
  ntdll!KiFastSystemCall
  mov      edx,esp
  sysenter
```

Listing 3.2: Reassembled function `ntdll!KiFastSystemCall()`

This function only copies the stack pointer to the `edx` register and executes the `SYSENTER` instruction, which executes the requested function as described above in the kernel.

3.4 Objects and Handles

Objects are implemented as statically defined structures in the kernel, represent different types of runtime objects, such as threads, files, and communication ports. These objects are accessible only from kernel-mode, user-mode programs cannot use them directly. As representation for these objects, user-mode programs receive handles, which are 32 bit integers. These handles are unique for a process (they are unique for all handles, not only for one type of handle). Thus, the handles for e.g., a thread and a file will never have the same value. Handles are received by user-mode programs via an accordant functions, such as `NtCreateFile()`. There are two “special handles” that can be used without creating an object: `0xFFFFFFF` represents the currently running process, and `0xFFFFFFFE` represents the currently running thread. If one thread wishes to supply a handle from itself, it can use `NtDuplicateObject()`, using `0xFFFFFFFE` as source handle.

The function `NtClose()` can be used to close handles. The impact of this system call depends on the type of the closed handle: if the handle represents a file, this file is closed; if the handle represents a thread, `NtClose()` will only make the handle itself invalid, the thread is not terminated (for this `NtTerminateThread()` can be used).

3.5 Useage of System Calls and Object Handles

For a better understanding, the use of handles is demonstrated with a simple piece of code. The function `fileFoo()` opens a file `foo.txt` and writes some bytes into this file.

```
1  void fileFoo() {
2    FILE *file = fopen ( "C:\\foo.txt", "w") ;
3    fprintf ( file, "Hello World!\n" );
4    fclose ( file );
5  }
```

Listing 3.3: Writing to a file

In line 2, the file is created. The `fopen()` is linked to and executed by `kernel32!CreateFileA()`. To create a file, the system call `NtCreateFile()` is executed.

3.5.1 NtCreateFile()

```
1  NTSATUS NtCreateFile(
2    OUT PHANDLE FileHandle,
3    IN  ACCESS_MASK DesiredAccess,
4    IN  POBJECT_ATTRIBUTES ObjectAttributes,
5    OUT PIO_STATUS_BLOCK IoStatusBlock,
6    IN  PLARGE_INTEGER AllocationSize OPTIONAL,
7    IN  ULONG FileAttributes,
8    IN  ULONG ShareAccess,
9    IN  ULONG CreateDisposition,
10   IN  ULONG CreateOptions,
11   IN  PVOID EaBuffer OPTIONAL,
12   IN  ULONG EaLength
13 );
```

Listing 3.4: ntdll!NtCreateFile()

The `OUT` in Line 2 indicates that this parameter is used as out-parameter, thus, the function will set a value. `PHANDLE` is a pointer to a handle (whereas a handle is as already mentioned a 32 bit integer). Therefore, if the function returns and signals success, `FileHandle` will point to a handle, representing the created file (whereas "create" not necessarily means that the file is created - it could have been opened with this function too). For our function `fileFoo()` let the handle be e.g., `0x7E8`.

In Line 3, `DesiredAccess` is a bit mask, defining the access, such as `FILE_READ_ACCESS 0x1`, `FILE_WRITE_ACCESS 0x2`, `FILE_APPEND_DATA 0x4`, and `FILE_EXECUTE 0x20` etc.

In Line 4, `ObjectAttributes` points to a structure, describing the requested object in more detail. Among other things, this structure contains an unicode string `ObjectName`, which is in our case `"C:\foo.txt"`. The rest of the parameters are for the understanding of our current example not relevant. Some of these are described later in the context of other functions, see Section 6.1.1.

In Line 4 of our function `fileFoo()`, the obligatory "Hello World" is written to the first line of our file. The `fprintf()` is executed by `kernel32!WriteFile()`, which itself uses the system-call `NtWriteFile()`.

3.5.2 NtWriteFile()

`NtWriteFile()` looks similar to another function, `NtDeviceIoControlCode()`, which will be discussed later in more detail. For now have a look at the parameters that are interesting for the current example. For the parameter `FileHandle` in Line 2, the handle received by `NtCreateFile()` is used, for our example `0x7E8`. The string "Hello World" is standing in the buffer, referenced by `Buffer`, the length of the buffer is defined by the parameter `Length` in Line 9.

```
1  NTSTATUS NtWriteFile(
2     IN  HANDLE FileHandle,
3     IN  HANDLE Event OPTIONAL,
4     IN  PIO_APC_ROUTINE ApcRoutine OPTIONAL,
5     IN  PVOID ApcContext OPTIONAL,
6     OUT PIO_STATUS_BLOCK IoStatusBlock,
7     IN  PVOID Buffer,
8     IN  ULONG Length,
9     IN  PLARGE_INTEGER ByteOffset OPTIONAL,
10    IN  PULONG Key OPTIONAL
11 );
```

Listing 3.5: ntdll!NtWriteFile()

3.6 Tools for Analyzing Windows

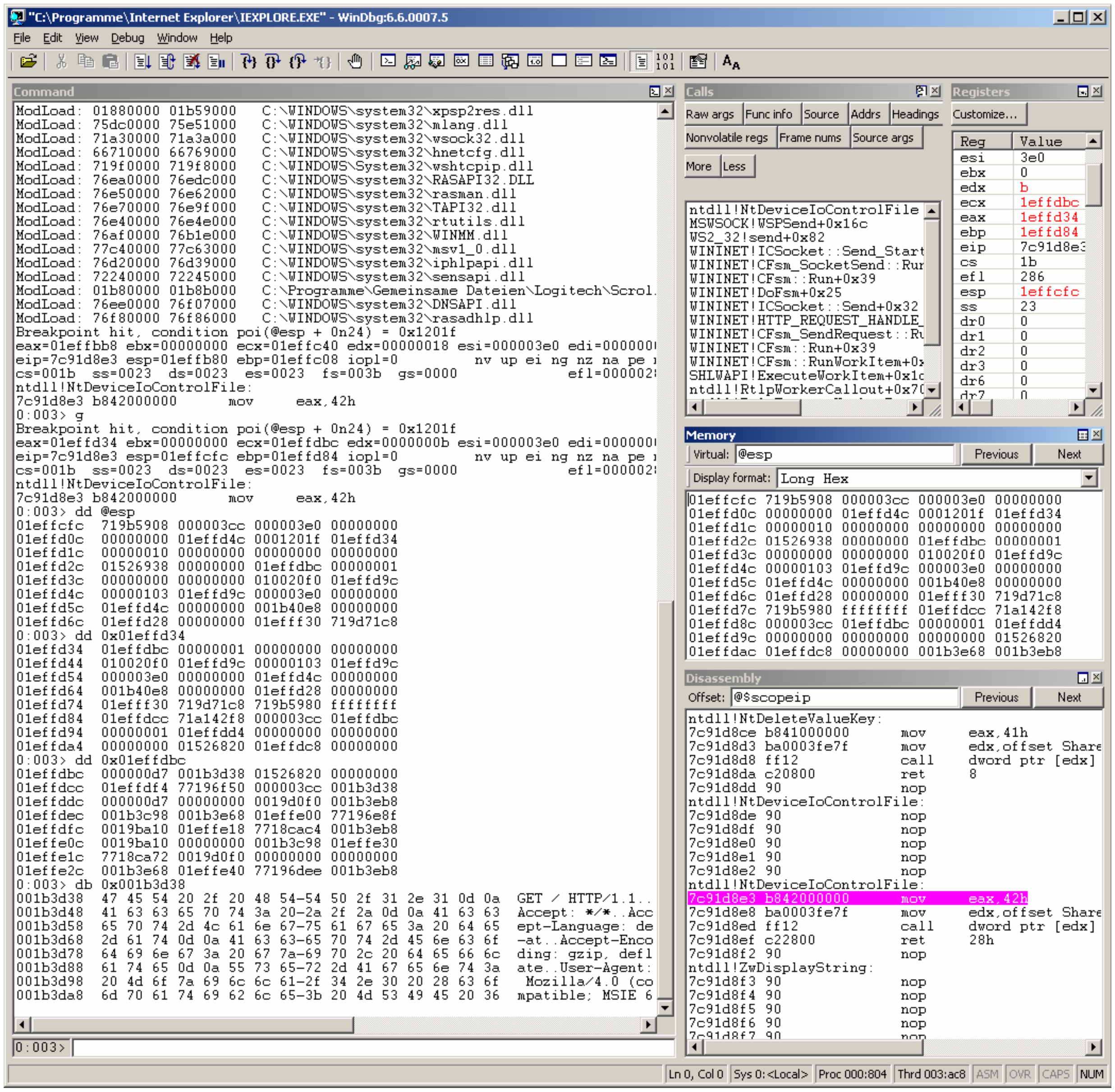
The most important tool for this thesis was, beyond TTAnalyze, the Windows Debugger. There are alternatives, such as SoftICE, but WinDbg [18] has been a good tool for our purposes.

3.6.1 WinDbg

The Windows Debugger can be used for user-mode and for kernel-debugging session. Today it is not possible to debug both user-mode and kernel-mode and it is currently unlcear if it will be supported in future. This tool is updated frequently to consider new released Windows operating system versions (e.g., for a new Service Pack).

User Mode Debugging

The screenshot of WinDbg in Figure 3.2 shows a debugging session for a TCP send, using the Function `NtDeviceIoControlFile()` with the the AFD code `AFD_SEND (0x1201F)`. This example will be discussed in detail in section 6.3.3.


"C:\Programme\Internet Explorer\IEXPLORE.EXE" - WinDbg:6.6.0007.5
File Edit View Debug Window Help
Command
Calls
Raw args Func info Source Addrs Headings
Nonvolatile regs Frame nums Source args
More Less
Registers
Customize...
Memory
Virtual: @esp
Previous Next
Display format: Long Hex
Disassembly
Offset: @$scopeip
Previous Next
Ln 0, Col 0 Sys 0:<Local> Proc 000:804 Thrd 003:ac8 ASM OVR CAPS NUM

```
ModLoad: 01880000 01b59000   C:\WINDOWS\system32\xpsp2res.dll
ModLoad: 75dc0000 75e51000   C:\WINDOWS\system32\mlang.dll
ModLoad: 71a30000 71a3a000   C:\WINDOWS\system32\wsock32.dll
ModLoad: 66710000 66769000   C:\WINDOWS\system32\hnetcfg.dll
ModLoad: 719f0000 719f8000   C:\WINDOWS\System32\wshtcpip.dll
ModLoad: 76ea0000 76edc000   C:\WINDOWS\system32\RASAPI32.DLL
ModLoad: 76e50000 76e62000   C:\WINDOWS\system32\rasman.dll
ModLoad: 76e70000 76e9f000   C:\WINDOWS\system32\TAPI32.dll
ModLoad: 76e40000 76e4e000   C:\WINDOWS\system32\rtutils.dll
ModLoad: 76af0000 76b1e000   C:\WINDOWS\system32\WINMM.dll
ModLoad: 77c40000 77c63000   C:\WINDOWS\system32\msv1_0.dll
ModLoad: 76d20000 76d39000   C:\WINDOWS\system32\iphlpapi.dll
ModLoad: 72240000 72245000   C:\WINDOWS\system32\sensapi.dll
ModLoad: 01b80000 01b8b000   C:\Programme\Gemeinsame Dateien\Logitech\Scrol.
ModLoad: 76ee0000 76f07000   C:\WINDOWS\system32\DNSAPI.dll
ModLoad: 76f80000 76f86000   C:\WINDOWS\system32\rasadhlp.dll
Breakpoint hit, condition poi(@esp + 0n24) = 0x1201f
eax=01effbb8 ebx=00000000 ecx=01effc40 edx=00000018 esi=000003e0 edi=000000
eip=7c91d8e3 esp=01effb80 ebp=01effc08 iopl=0         nv up ei ng nz na pe
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00002
ntdll!NtDeviceIoControlFile:
7c91d8e3 b842000000       mov     eax,42h
0:003> g
Breakpoint hit, condition poi(@esp + 0n24) = 0x1201f
eax=01effd34 ebx=00000000 ecx=01effdbc edx=0000000b esi=000003e0 edi=000000
eip=7c91d8e3 esp=01effcfc ebp=01effd84 iopl=0         nv up ei ng nz na pe
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00002
ntdll!NtDeviceIoControlFile:
7c91d8e3 b842000000       mov     eax,42h
0:003> dd @esp
01effcfc  719b5908 000003cc 000003e0 00000000
01effd0c  00000000 01effd4c 0001201f 01effd34
01effd1c  00000010 00000000 00000000 00000000
01effd2c  01526938 00000000 01effdbc 00000001
01effd3c  00000000 00000000 010020f0 01effd9c
01effd4c  00000103 01effd9c 000003e0 00000000
01effd5c  01effd4c 00000000 001b40e8 00000000
01effd6c  01effd28 00000000 01efff30 719d71c8
0:003> dd 0x01effd34
01effd34  01effdbc 00000001 00000000 00000000
01effd44  010020f0 01effd9c 00000103 01effd9c
01effd54  000003e0 00000000 01effd4c 00000000
01effd64  001b40e8 00000000 01effd28 00000000
01effd74  01efff30 719d71c8 719b5980 ffffffff
01effd84  01effdcc 71a142f8 000003cc 01effdbc
01effd94  00000001 01effdd4 00000000 00000000
01effda4  00000000 01526820 01effdc8 00000000
0:003> dd 0x01effdbc
01effdbc  000000d7 001b3d38 01526820 00000000
01effdcc  01effdf4 77196f50 000003cc 001b3d38
01effddc  000000d7 00000000 0019d0f0 001b3eb8
01effdec  001b3c98 001b3e68 01effe00 77196e8f
01effdfc  0019ba10 01effe18 7718cac4 001b3eb8
01effe0c  0019ba10 00000000 001b3c98 01effe30
01effe1c  7718ca72 0019d0f0 00000000 00000000
01effe2c  001b3e68 01effe40 77196dee 001b3eb8
0:003> db 0x001b3d38
001b3d38  47 45 54 20 2f 20 48 54-54 50 2f 31 2e 31 0d 0a  GET / HTTP/1.1..
001b3d48  41 63 63 65 70 74 3a 20-2a 2f 2a 0d 0a 41 63 63  Accept: */*..Acc
001b3d58  65 70 74 2d 4c 61 6e 67-75 61 67 65 3a 20 64 65  ept-Language: de
001b3d68  2d 61 74 0d 0a 41 63 63-65 70 74 2d 45 6e 63 6f  -at..Accept-Enco
001b3d78  64 69 6e 67 3a 20 67 7a-69 70 2c 20 64 65 66 6c  ding: gzip, defl
001b3d88  61 74 65 0d 0a 55 73 65-72 2d 41 67 65 6e 74 3a  ate..User-Agent:
001b3d98  20 4d 6f 7a 69 6c 6c 61-2f 34 2e 30 20 28 63 6f   Mozilla/4.0 (co
001b3da8  6d 70 61 74 69 62 6c 65-3b 20 4d 53 49 45 20 36  mpatible; MSIE 6

0:003>
```

```
ntdll!NtDeviceIoControlFile
MSWSOCK!WSPSend+0x16c
WS2_32!send+0x82
WININET!ICSocket::Send_Start
WININET!CFsm_SocketSend::Rur
WININET!CFsm::Run+0x39
WININET!DoFsm+0x25
WININET!ICSocket::Send+0x32
WININET!HTTP_REQUEST_HANDLE_
WININET!CFsm_SendRequest::Ru
WININET!CFsm::Run+0x39
WININET!CFsm::RunWorkItem+0x
SHLWAPI!ExecuteWorkItem+0x1c
ntdll!RtlpWorkerCallout+0x70
```

```
Reg   Value
esi   3e0
ebx   0
edx   b
ecx   1effdbc
eax   1effd34
ebp   1effd84
eip   7c91d8e3
cs    1b
efl   286
esp   1effcfc
ss    23
dr0   0
dr1   0
dr2   0
dr3   0
dr6   0
dr7   0
```

```
01effcfc 719b5908 000003cc 000003e0 00000000
01effd0c 00000000 01effd4c 0001201f 01effd34
01effd1c 00000010 00000000 00000000 00000000
01effd2c 01526938 00000000 01effdbc 00000001
01effd3c 00000000 00000000 010020f0 01effd9c
01effd4c 00000103 01effd9c 000003e0 00000000
01effd5c 01effd4c 00000000 001b40e8 00000000
01effd6c 01effd28 00000000 01efff30 719d71c8
01effd7c 719b5980 ffffffff 01effdcc 71a142f8
01effd8c 000003cc 01effdbc 00000001 01effdd4
01effd9c 00000000 00000000 00000000 01526820
01effdac 01effdc8 00000000 001b3e68 001b3eb8
```

```
ntdll!NtDeleteValueKey:
7c91d8ce b841000000       mov     eax,41h
7c91d8d3 ba0003fe7f       mov     edx,offset Share
7c91d8d8 ff12             call    dword ptr [edx]
7c91d8da c20800           ret     8
7c91d8dd 90               nop
ntdll!NtDeviceIoControlFile:
7c91d8de 90               nop
7c91d8df 90               nop
7c91d8e0 90               nop
7c91d8e1 90               nop
7c91d8e2 90               nop
ntdll!NtDeviceIoControlFile:
7c91d8e3 b842000000       mov     eax,42h
7c91d8e8 ba0003fe7f       mov     edx,offset Share
7c91d8ed ff12             call    dword ptr [edx]
7c91d8ef c22800           ret     28h
7c91d8f2 90               nop
ntdll!ZwDisplayString:
7c91d8f3 90               nop
7c91d8f4 90               nop
7c91d8f5 90               nop
7c91d8f6 90               nop
```

Figure 3.2: Screenshot from WinDbg, debugging the Internet Explorer

Kernel Mode Debugging

Microsoft offers symbol files for its debugger, which offer informations about structures that are not documented in an official way. We give a short example how to work with WinDbg, and how Microsoft offers informations about officially undocumented structures. We show, how to find the place, where the PDBR (described in section 2.3.2) is saved for a specific process.

In Listing 3.6 it can be seen, how the command `dt` (display type) dumps the undocumented `EPROCESS` structure. This information provides a good understanding of the internal used structures of Windows.

```
0: kd> dt nt!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x06c ProcessLock      : _EX_PUSH_LOCK
   +0x070 CreateTime       : _LARGE_INTEGER
   +0x078 ExitTime         : _LARGE_INTEGER
   +0x080 RundownProtect   : _EX_RUNDOWN_REF
   +0x084 UniqueProcessId  : Ptr32 Void
   +0x088 ActiveProcessLinks : _LIST_ENTRY
   +0x090 QuotaUsage       : [3] Uint4B
   +0x09c QuotaPeak        : [3] Uint4B
   +0x0a8 CommitCharge     : Uint4B
   +0x0ac PeakVirtualSize  : Uint4B
   +0x0b0 VirtualSize      : Uint4B
   +0x0b4 SessionProcessLinks : _LIST_ENTRY
   +0x0bc DebugPort        : Ptr32 Void
   +0x0c0 ExceptionPort    : Ptr32 Void
   +0x0c4 ObjectTable      : Ptr32 _HANDLE_TABLE
[...]
```

Listing 3.6: Parts of the `EPROCESS` structure

To get the PDBR, we need to dump the `_KPROCESS` structure:

```
0: kd> dt nt!_KPROCESS
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 ProfileListHead  : _LIST_ENTRY
   +0x018 DirectoryTableBase : [2] Uint4B
   +0x020 LdtDescriptor    : _KGDTENTRY
   +0x028 Int21Descriptor  : _KIDTENTRY
   +0x030 IopmOffset       : Uint2B
   +0x032 Iopl             : UChar
   +0x033 Unused           : UChar
   +0x034 ActiveProcessors : Uint4B
   +0x038 KernelTime       : Uint4B
   +0x03c UserTime         : Uint4B
   +0x040 ReadyListHead    : _LIST_ENTRY
   +0x048 SwapListEntry    : _SINGLE_LIST_ENTRY
   +0x04c VdmTrapcHandler  : Ptr32 Void
   +0x050 ThreadListHead   : _LIST_ENTRY
   +0x058 ProcessLock      : Uint4B
   +0x05c Affinity         : Uint4B
   +0x060 StackCount       : Uint2B
   +0x062 BasePriority     : Char
   +0x063 ThreadQuantum    : Char
   +0x064 AutoAlignment    : UChar
   +0x065 State            : UChar
   +0x066 ThreadSeed       : UChar
   +0x067 DisableBoost     : UChar
   +0x068 PowerState       : UChar
   +0x069 DisableQuantum   : UChar
   +0x06a IdealNode        : UChar
   +0x06b Flags            : _KEXECUTE_OPTIONS
   +0x06b ExecuteOptions   : UChar
```

Listing 3.7: The `KPROCESS` structure

The `KPROCESS` structure is `0x6c` bytes long (the last element starts at offset `0x6b` and is one byte long), the element after the `Pcb` in the `EPROCESS` structure starts at offset `0x6c`, therefore, dumping the `KPROCESS` structure obviously shows the beginning of the `EPROCESS` structure.

At offset `+0x018` the entry `DirectoryTableBase` can be seen. This looks like what we are looking for. To be sure we start a kernel debugging session. To verify that the `DirectoryTableBase` contains the value that will be loaded into the `CR3` register (remember, we wish to identify a process in a efficient manner), we need a user-mode process running into a breakpoint, finding the `EPROCESS` structure for this process in memory and compare the `DirectoryTableBase` with the `CR3` register. The values should be equal at least for the 20 most significant bits[2].

For kernel debugging a second workstation is needed[3], which has to be connected via e.g., a 1394er cable. We need a user mode process that e.g., opens a file. We set a breakpoint `bp nt!NtCreateFile` and let the test process open a file. To assure that really the expected test process has triggered the breakpoint[4], we let the debugger display some information of the current process. For this, we use the command `!process`. In Listing 3.8 we see parts of the output `!process` is delivering.

```
0: kd> !process
PROCESS 8533fb38  SessionId: 0  Cid: 0620    Peb: 7ffd5000
```

[2]They will be equal for the complete DWORD, because during a process switch the `DirectoryTableBase` is loaded directly in the `CR3 register`

[3]With WinDbg it is possible to start a kernel debugging session on the same host, but in this case obviously no breakpoints can be set - it is impossible to halt the system for a breakpoint and simultaneous explore the system with the debugger

[4]For kernel debugging the complete system is debugged, and therefore all running processes could execute a `nt!NtCreateFile` and run into the corresponding breakpoint

```
                                                   ParentCid: 00a4
    DirBase: 06c40280  ObjectTable: e25ecae0  HandleCount: 368.
    Image: winamp.exe
    VadRoot 8523fe50 Vads 223 Clone 0 Private 4225. Modified 5564.
                                                    Locked 87.
    DeviceMap e178fa80
[...]
```

Listing 3.8: The `!process` command

Note that our test-executable hit the breakpoint. Thus, we need to find the `EPROCESS` structure for this process in memory - this is the value right after `PROCESS` - `8533fb38`. Dumping the memory reveals the following:

```
0: kd> dd 0x8533fb38
8533fb38  001b0003 00000000 8533fb40 8533fb40
8533fb48  8533fb48 8533fb48 06c40280 0003d743
8533fb58  00000000 00000000 00000000 00000000
8533fb68  000020ac 00000003 00000032 00000035
8533fb78  8533fb78 8533fb78 00000000 00000000
8533fb88  85fcad48 860f71d0 00000000 00000003
8533fb98  06080008 00000000 32000000 00000000
8533fba8  406392c4 01c71eb4 00000000 00000000
```

Listing 3.9: Dumping the EPROCESS structure

As mentioned above, the `EPROCESS` structure starts with no offset with the `KPROCESS` structure. At offset `+0x018` we hope to find our PBDR, which has a value of `06c40280`, if we look at the value of the `CR3` register, we see that it has the same value. Therefore, we found the place where the PDBR is saved in the Windows kernel internal structures. The driver of InsideTM does exactly this at system runtime.

Chapter 4

Networking under Windows

At the beginning of the Internet, Microsoft believed that networking was not important (in contrast to e.g., BSD etc.). Therefore, a non-Microsoft browser - Mosaic (i.e., Netscape) - was dominant. Microsoft developed the Internet Explorer relatively late (introduced it with Windows 95). Today, networking is an essential part of Windows.

In this chapter the general basics about networking are discussed, starting with the OSI Reference Model over the most used protocols for the daily use of the Internet, TCP, UDP, and ICMP. In the following it is shown how networking is implemented with Winsock and which functions for a simple request / response are used.

4.1 OSI Reference Model

The basis for the most used protocols is the OSI Reference Model (Open Systems Interconnection Basic Reference Model). It is a abstract description of how networking should be implemented to achieve platform independent communication channels. The OSI Model was standardised in 1983 by the International Standards Organization (ISO).

The OSI Model describes a set of layers, whereas all are responsible for a concrete task, but not all indications are used in the concrete implementations.

4.1.1 Layers

The Data to be transported over the network are handled down from Layer 7 (Application Layer) to Layer 1 (Physical Layer), and are transported over the physical medium and are transformed in reverse order back to Layer 7, where the data (bits - therefore zeros and ones) can be used, depending on the concrete implemented protocol, e.g., as stream.

Layer 1 - Physical Layer

The Physical Layer is the physical infrastructure of a network, i.e., cables or another transmission medium such as the air for Wireless LAN. Layer 1 receives the binary data from higher layers and sends these over the physical medium.

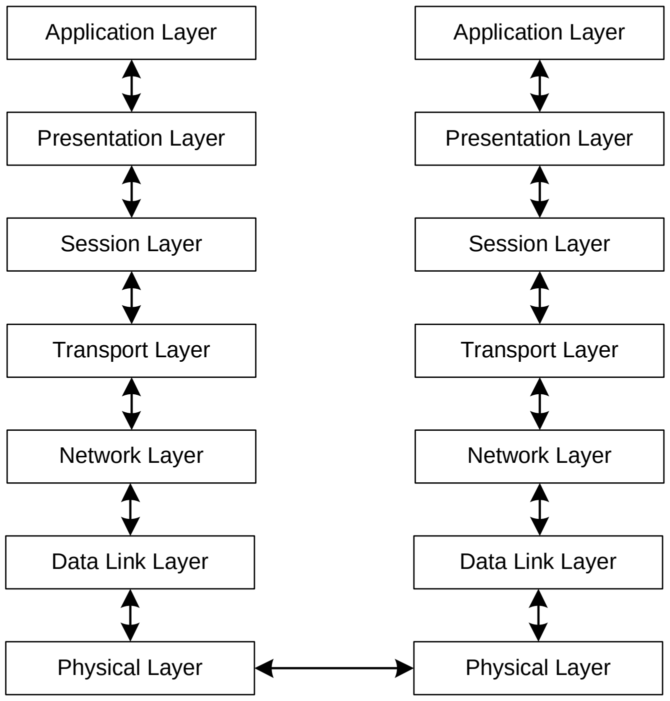


Figure 4.1: Schema of the OSI Reference Model

Layer 2 - Data Link Layer

The Data Link Layer is responsible for sending the data over the Physical Layer and is the only layer that works both in software and hardware. For this, the data accepted from higher layers are sent in frames over the Physical Layer. Therefore, it provides the functionality to transfer data between two physical network devices, which are unique by their physical address. The most famous example for the Data Link Layer is Ethernet, the physical addresses for Ethernet are the so called MAC (Media Access Control) addresses. The physical address is embedded by the manufacturer in the network devices, so the mac address is unique on the network (every manufacturer has its own address space). The Data Link Layer is sometimes called the Physical Address Layer, to regard the meaning of the physical addresses.

At this Layer, data can only be sent between two physical connected devices. If more than two devices are connected to a physical medium, the network device is able to separate out unwanted data: data, which are not addressed to the physical address that the network device is representing, are discarded. Switches are working at this layer: they remember at which ports which MAC addresses are connected and transmit packets only to these ports (in contrast to hubs, which spread the whole traffic to all ports, which reduces the capacity of the network).

The data accepted from higher layers are divided into parts called frames, which contain, e.g., for Ethernet a payload from 46 up to 1500 bytes. For Ethernet, the frame contains a header, which contains informations such as source and destination MAC addresses and a checksum, for detecting errors. The frames passed down to the physical layer are interpreted as pure binaries and sent over the physical device. The receiving host checks, if the destination MAC address is equals to its own address, and if not, the frame is ignored. If the MAC address matches, the header is removed and the rest of the data are passed to the next higher layer.

Layer 3 - Network Layer

The Network Layer is entirely implemented in software, the hardware access is encapsulated completely by the Data Link Layer. The Network Layer works with logical addresses, which could be named software addresses too. Out of this, they do not need to be random (like MAC addresses, which have to be unique), and can be defined by the user, e.g., in a hierarchical form like IP addresses. Therefore, with the abilities of the Network Layer, network traffic can be directed to a destination, of which the physical address is not known and which does not have to be on the same (physical) network. With the ability of hierarchical addresses it is possible to build large networks. These can be connected via routers, which are smart devices that work on Layer 3. These devices do not blindly repeat packets at Layer 1, instead they (try to) route the packets directly to their destination. This results in much less collisions (on the Physical Layer) as if all packets would be send broadcast.

Similar to the Data Link Layer, the data accepted from the higher layer are divided into portions, called packets. These packets start with a header, containing the logical source and destination address. The most famous implementation of the Network Layer is IP (Internet Protocol), the corresponding addresses are IP addresses.

Layer 4 - Transport Layer

The Transport Layer is responsible for transferring the data transparent for higher layers. It implements reliability (if provided by the implementing protocol), e.g., with a checksum over the payload (the data to be transported). State- and connection oriented protocols keep track of packets and retransmit packets that have been lost. Furthermore, protocols at Layer 4 can implement ports, which allow more than one connection and therefore more than one application to use a network device for several connections.

Layer 5 - Session Layer

The Session Layer is not used in most today dominating protocols (i.e., TCP, UDP, etc.). The functionality that should be implement by Layer 5 is mostly implemented in Layer 7, e.g., with http session cookies.

Layer 6 - Presentation Layer

The Presentation Layer is also commonly not used. It transforms data from one format to another, e.g., an ASCII-coded file to an unicode file or transforming XML files to another format. Like the Session Layer, the Presentation Layer is rather theoretical, because most of these transformations are done at Layer 7.

Layer 7 - Application Layer

The upperst layer is the Application Layer, where protocols define a way, how to access resources through the web, e.g., http, ftp, ssh, etc.

4.2 Protocols

The concrete implementation of the OSI Reference Model are several protocols. In this thesis, the focus is set to the protocols already mentioned: TCP, UDP and ICMP. These are the protocols that are used in most Internet applications and therefore by malware too.

4.2.1 IP

The Internet Protocol is the implementation of OSI Layer 3, Network Layer. It provides logical, "software" addresses with 32 bits (IPv4)[19] and 128 bits (IPv6)[20], whereas IPv4 is used today mostly, although IPv6 will be the protocol in future[1]. As shown in Figure 4.2, routers work at Network Layer and therefore at IP level.

4.2.2 TCP

TCP (Transmission Control Protocol) [21] is build on IP, the resulting combination of protocols is referenced as TCP/IP Stack. TCP is a stateful, connection oriented protocol, guaranteeing reliable in-order delivery of data from the sender to the receiver (or, if a transport is not possible the sender is notified about a failure). TCP extends IP with the ability of ports.

Connection oriented protocol means that for sending data over TCP a connection has to be created. Stateful means that this connection is always in a definite state, so that error can be detected, if the connection endpoints are not in the same state. These errors provide the ability to detect packet lost and therefore are the basic for retransmitting packets, until they arrive at the receiver.

In TCP (and, as we will see in UDP too) the endpoints of connections are represented by sockets. Therefore, to create a connection a socket has to be created first. Such a socket is connected to another socket, waiting for connection. The waiting socket is for usual implemented as server socket, waiting for several incoming connections. The socket initialising the connection is referenced as client socket.

TCP is the "core protocol" for the Internet today. It is the basic for many Layer 7 protocols such as http, pop3, imap, ssh, etc.

[1]The address space with 32 bits will become to small in near future - although not everybody has a computer and a public IP, there are already more people on the world than IPv4 addresses could provide

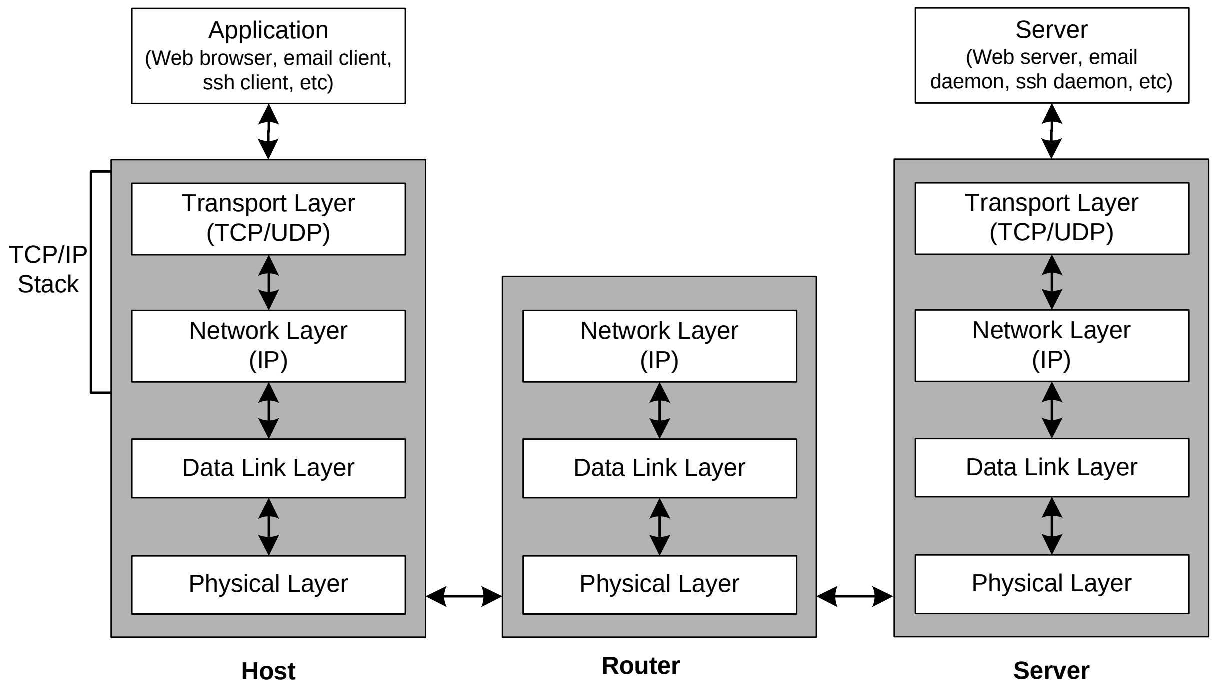


Figure 4.2: Implementation of TCP/IP in the OSI Reference Model [2]

Now we describe, which Control Bits TCP uses, and how the connection is established from the view of the network. We later describe how the connection establishment and TCP networking in general are implemented in software.

Control Bits

Control Bits in TCP are used to describe the according packet in more detail. Those ones needed to discuss the connection establishment and those, which will be referenced later, are described now.

- `ACK` signals that the acknowledgement field (a 32 bit field in the TCP header) is significant, therefore that the corresponding packet is used to acknowledge a packet sent earlier.
- `SYN` requests a (re)synchronization of the sequence numbers - it is used during e.g., connection establishment. As the acknowledgement field, the sequence number is a 32 bit field in the TCP header.
- `RST` signals that the connection has to be reseted, due to error or other interruption.
- `FIN` is used by the sender, to signal that the connection can be closed.

Connection establishment

For connecting two TCP sockets, the so called Three-Way Handshake is executed. It is called Three-Way Handshake, because there are three packets sent over the network, setting up the connection. As you can see in Figure 4.3, the client sends a packet containing an initial sequence number ISN C (therefore the `SYN` bit is set), whereas this value is randomly created by the TCP layer.

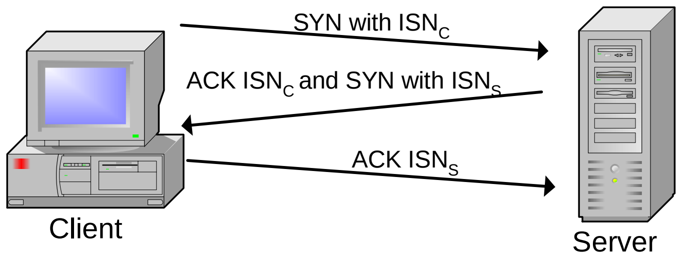


Figure 4.3: The TCP three-way handshake [2]

The server replies (if the corresponding port is opened) with the initial sequence number from the client (ISN C) in the acknowledgement field, and generates itself an initial sequence number ISN S and sets the sequence number field to this value. Therefore, in the second packet the `SYN` and the `ACK` flags are set.

The client then acknowledges the sequence number ISN S from the server by sending the third and last packet of the Three-Way Handshake with the `ACK` flag, whereas the acknowledgement field contains the sequence number of the server, ISN S.

4.2.3 UDP

UDP (User Datagram Protocol) [22] is, like TCP, build on IP and extends IP with the ability of ports. In contrast to TCP, it is connection less and not reliable. The main advantage is that no connection has to be established and no state information has to be saved. Therefore, the transfer of UDP packets is much faster than with TCP. UDP is used for applications, which do not need a guarantee for receiving all packets, but need the packets transmitted as quickly as possible, e.g., streaming applications for multimedia data. It is used for e.g., Internet telephony protocols such as VoIP.

4.2.4 ICMP

The Internet Control Message Protocol [23] is also based on IP and serves as protocol for submitting error- and information messages. In contrast to TCP or UDP, it is not used by users directly, but implicit by the devices (hosts, routers, etc.) connected to the network for exchanging status information, such as Time To Live (TTL) exceeded (code 11). The only exception for this rule is the “ping” request, mostly available by a tool named like the request. It checks the availability of a specific IP address on the network.

4.3 Implementation of Networking in User Mode

For the implementation of applications using TCP or UDP network access the programmer does not have to know anything about TCP flags and the like. For this, there exist APIs that encapsulate the creation of connections and the sending of data.

The networking API for TCP and UDP under Windows is the Winsock (Windows Sockets) library. Version 1.0 was Microsoft's implementation of BSD Sockets, the standard API for UNIX systems since the 1980s. The current version added, among others things, features for asynchronous I/O that offers better performance and scalability [15].

Winsock describes the API for networking and this interface is mainly implemented in the API DLL `WS2_32.dll`. Obviously, this DLL does not implement the whole user mode part, it uses itself helper DLLs. `mswsock.dll` acts as transport service provider, and this library uses Winsock Helper libraries for the accordingly protocols (e.g., `wshtcpip.dll` for the TCP/IP Stack).

As already mentioned, the communication endpoints of TCP and UDP connections[2] are sockets. For TCP, these sockets have to be connected to another socket, before any data can be sent over it. For UDP, the socket has only to be created, after that the data can be sent to any UDP server socket, without the need of connecting it.

Listing 4.1 shows a simple piece of code, doing a send / receive over TCP. In the next subsections the used functions and their parameters are discussed. In further chapters it will be shown, how these functions are implemented, which system calls are used, and how multiple connections running in multiple threads can be synchronised.

```
void foo() {
   WSADATA wsaData;
   WSAStartup( MAKEWORD(2,2), &wsaData );
   SOCKET m_socket = socket( AF_INET, SOCK_STREAM,
                                     IPPROTO_TCP );
   sockaddr_in service; /* setup service */
   service.sin_family = AF_INET;
   service.sin_addr.s_addr = *(unsigned long*)
      gethostbyname( "seclab.tuwien.ac.at" )->h_addr_list[0];
   service.sin_port = htons( 80 );
   connect( m_socket, (SOCKADDR*) &service,
                                 sizeof(service) );
   int bytesSent = send(m_socket, sendStr.c_str(),
                                sendStr.size(), 0);
   char buff[4096];
   int recvBytes = recv(m_socket, buff, 4096, 0);
}
```

Listing 4.1: A simple send / receive with Winsock2

[2]UDP is connection less, but like TCP, for UDP there has to be a client sending UDP packets and a server waiting for UDP packets

4.3.1 WSAStartup()

For using the Winsock library, first an initialization function has to be called: `WSAStartup()`. This function initiates the Winsock library for further use.

The declaration in Line 2 from Listening 4.1 creates the structure `WSADATA`, which is an out parameter[3] of `WSAStartup()`. This structure contains informations such as the used version numbers, etc.

```
int WSAStartup(
  WORD wVersionRequested,
  LPWSADATA lpWSAData
);
```

Listing 4.2: `WSAStartup()`

The parameter `wVersionRequested` defines the highest version the user[4] supports. The major version is defined by the low-order byte, the minor version by the high-order byte. The `MAKEWORD` macro is in `windef.h`, and receives two bytes and returns a `WORD`. The second parameter `lpWSAData` is a pointer to the already mentioned structure `WSADATA`, which receives informations such as the used version.

4.3.2 socket()

As already mentioned in section 4.2.2, sockets are the representation of communication endpoints. For high level programing languages an object representing such an endpoint is needed. All further actions are executed for this object.

```
SOCKET socket(
  int af,
  int type,
  int protocol
);
```

Listing 4.3: `socket()`

For creating a socket with Winsock, the function `socket()` is called. This function returns `SOCKET`, a handle for the socket object saved in the kernel. Therefore, the object is not accessible directly, the user mode program only receives a handle for this object. For executing functions for this socket, in user mode this handle has to be used.

The first parameter `af` defines the address family of the requested socket. For creating an IPv4 socket, `AF_INET` is used. Obviously, there are many other possibility, such as `AF_INET6` for IPv6, `AF_NETBIOS`, etc. The second parameter `type` defines the type of the new socket. For the creation of a TCP socket `SOCK_STREAM` is used, which requests a sequenced, reliable, two-way, connection-based byte stream. Other values are e.g., `SOCK_DGRAM` for

[3]Which means that the called function will manipulate the assigned function parameter.

[4]In this context, the user is the programmer or the program, using the Winsock library

connectionless, unreliable datagrams (e.g., used for UDP), or `SOCK_RAW` for a raw socket[5]. `protocol` defines the protocol to be used, thus, for a TCP connection `IPPROTO_TCP` is used, for UDP sockets `IPPROTO_UDP` is used.

4.3.3 gethostbyname()

From Line 5 to Line 8 the server endpoint of the connection is defined. `sockaddr_in` represents a socket address in Internet style, defined in `winsock.h`. As for the client socket, the address family for the server socket is defined as `AF_INET`. In Line 7, the destination address of the connection is defined, or how the server socket can be located. In our example, the IP address of the destination is not known, only the domain name. For this, we resolve the domain name to an IP address, using the Domain Name System (DNS) [24]. The application of the DNS protocol is implemented by Winsock.

`gethostbyname()`[6] receives a pointer to a null-terminated ANSI string and returns a pointer to a `hostent` structure, which contains an array of addresses. As shown later, the resolving of the domain name is not executed by the current process, but by `services.exe`. The interprocess communication necessary for this task is described in Section 6.5.

4.3.4 connect()

In Line 9 the local socket is connected to the server socket. This is done with aim of the `connect()` function.

```
int connect(
  SOCKET s,
  const struct sockaddr* name,
  int namelen
);
```

Listing 4.4: `connect()`

As first parameter, the previously created socket is passed in. `name` defines the server socket to which the the local socket (referenced by the first parameter) has to be connected to. The third parameter `namelen` defines the length of the `name`.

If the connection establishment is successful, `connect()` returns zero. Otherwise, it returns `SOCKET_ERROR`, and a detailed error code can be obtained by calling `WSAGetLast-Error()`.

[5]For a raw socket no headers at OSI Layer 4 will be generated, the user has to write them himself. There are also possibilities to manipulate even the IP header

[6]This function is declared deprecated, but it is more simple to use than the substitute function `getaddrinfo()`, and for our simple example it is sufficient

4.3.5 send()

```
int send(
  SOCKET s,
  const char* buf,
  int len,
  int flags
);
```

Listing 4.5: `send()`

The `send()` in Line 10 will send the `sendStr` over the connected socket to the destination server. As we connected to port 80, `sendStr` may contain a http request. We use the now connected socket as first parameter `s`, and assign a pointer to the `sendStr` as second parameter `buf`, and `len` defines the length of this buffer. The flags parameter can be used to influences the way, the call is made - but this feature is not used.

The return value defines how many bytes have been sent if no error occurred, otherwise the return value is `SOCKET_ERROR`. Like for `connect()` a specific error can be obtained by calling `WSAGetLastError()`.

4.3.6 recv()

```
int recv(
  SOCKET s,
  char* buf,
  int len,
  int flags
);
```

Listing 4.6: `recv()`

A receive works from sight of the user similar to a send. The user has to allocate memory, where the received data is written to. The pointer to this buffer is assigned as second parameter `buf` to `recv`, `len` defines the length of the available buffer. The `flags` can be used to enforce Winsock to wait, until the complete buffer is filled (or the connection was closed). The return value defines the number of bytes received, or `SOCKET_ERROR` if an error occurred.

Chapter 5

Manipulating the Test Subject

The goal of this theses is to make dynamic analysis more sophisticated by extracting the behaviour of the test subject under different circumstances. For this, the test subject has to be executed under different conditions and then the differences between these executions have to be determined. Obviously, this naive approach has two problems:

- For every possible state an analysis process has to be executed.
- It would be necessary to generate many virtual systems, being in different states.

The second problem can be solved by manipulating the function calls which interact with the environment (these provide the informations about the state of the system and its environment for the test-subject). The functions responsible for this interaction have to be determined, hooked, and, if required, manipulated.

The decision, which functions have to be hooked is relative simple. In Windows every user-mode program has only one facility to interact with its environment: via the kernel and therefore via system calls, which are responsible for the interaction with the kernel. Thus, by hooking and manipulating system calls, it is possible to simulate every state of a system imaginable, without the (expensive) need, to create images of these virtual system states.

The first problem - one execution for every possible system state - is more complex. The number of theoretical possible system states is immense. The performance requirements are not stringent, but executing the test subject with all these system states would be a life-task!

A first step to a solution could be to restrict the system states to those ones, which directly influence the test-subject. This indicates that the decisions which system states should be tested is more easy to solve during execution: at run time it is known, which system calls are executed and therefore which system calls should be manipulated to influence the test subject. This leads to the idea, to choose a complete dynamic approach - the decision which system calls have to be manipulated is felt at runtime. If a execution path is not interesting, this path is left and the test-subject is set back to a previously saved state. For this a facility is needed, which is able to save states and reset the test subject to such a saved state.

In the next section it will be discussed, how system calls can be manipulated to influence the test-subject. In the following sections it is shown, why for reseting the test-subject to a previous state network connections are the main problem and how a solution for this problem could look like.

5.1 Manipulating System Calls

As discussed in the introduction to this chapter, the manipulation and simulation of system-calls can be used to simulation different system states. The first decision presented in the introduction has been to hook the system calls. A further challenge is to manipulate and simulate the execution of system calls.

5.1.1 Hooking System Calls

As shown in chapter 3.3, system calls are not like other function calls in user-mode. System calls are executed as software interrupt `INT 2Eh` or, on newer hardware, as explicit `sysenter` instruction. Therefore, for hooking system calls with TTAnalyze, it would be possible to analyze the assembler code executed by the virtual CPU. Obviously, this approach has two drawbacks:

- Additionally to the hooking of user mode functions in TTAnalyze, there would be a further analysis effort, which would reduce the performance.
- The function to be called (e.g., `NtCreateFile()`, `NtWriteFile()`, etc.) are referenced by a number in the `eax` register. This number is randomly generated at the build of Windows and therefore could change with every Windows version and with every service pack. This would reduce the flexibility of the solution.

To avoid these drawbacks, the functions of the unofficially undocumented DLL `ntdll.dll` are hooked. This DLL contains the last user mode code that is executed before the system switches to kernel mode. A malware author has therefore the theoretical chance to bypass this monitoring technique, but it would be difficult to write malware that is run-able on multiple systems (e.g., Windows 2000, Windows XP, with different Service Packs). Beside the changing values for the `eax` register, he would have to consider changing interfaces at this level, which are not reported by Microsoft. Using undocumented API functions therefore decreases the chance that malware is successful on multiple system.

Although the interface of the `ntdll.dll` is not documented by Microsoft, there exists a book offering unofficial documentation [12]. This book describes most of the functions we need and provides for most functions a detailed description of all function parameters and the used structures. These definitions are needed by the Generator.

5.1.2 Manipulating the Result of System Calls

With the knowledge presented till now, it is possible to hook system calls. For manipulating the result of system calls, a technique already implemented in TTAnalyze can be used: for every call of a hooked functions, TTAnalyze remembers (for every thread) the return address, which is saved on the stack. In the moment the function is started to be executed (i.e., the instruction pointer, saved in the `eip` register, points to the function entry point), the stack pointer (saved in the `esp` register) points to the return address (this value is called the saved instruction pointer). If a thread executes code at the return address, the function returned and the system and with it the function result can be manipulated as needed. E.g., the function result can be changed to any value, by overwriting the `eax` register, any buffers in the system (on the stack or on the heap) can be manipulated.

5.1.3 Virtualization of System Calls

There will be cases (discussed in Section 6.4.3), where the execution of system call is not wished but a result is needed as if the function has been executed. Thus, the execution of system calls has to be imitated. For this, a deep understanding of the imitated function is required, because not only a few bytes in the system we are interested in have to be changed (like for the manipulation of the result of a system call), but for virtualization the complete behaviour of the simulated system call has to be replayed.

To replay a function, the system is manipulated in the moment when the function is called. First, the execution of the function has to be avoided and the execution has to be redirected back to the calling function. For this, the return address is read from the stack and written to the `eip` register (instruction pointer). As already mentioned, the return address is saved at the stack and the stack pointer directly points to it at this moment.

After setting the `esp` register, the second task is to "clear" the stack as if the returning function would have done it. This means that the return address and the function parameters have to be popped from the stack, as they have been pushed on the stack for the function call. To avoid the execution of additional code (which would have to be injected into the virtual system) the stack pointer (saved in the `esp` register) is manipulated directly. For this, the number of the function parameters for the virtualized function has to be known.

The generation of the function result is the same task as described in section 5.1.2, but the result has not only to be manipulated - the complete behaviour has to be replayed.

5.2 Reseting Network Connections to a Previous State

In the introduction to this chapter, we decided that we wish to have a facility to save states of the current execution of a test-subject, and we noted that a main problem in doing this are network connections. In this section it is discussed, why this is a problem and how it could be solved.

The general idea of reseting a test-subject to a previous state should be clear. For a reset, all the variables inside the program are set to a previously saved state. If we not all the variables that are available inside a program are known (which will be mostly the case), it is also possible to reset the complete allocated memory of this program to a previous state. Obviously, the state of a program is not only defined by its internal state - the environment of the program (e.g., the system time, files on the hard disk) are directly or indirectly altered.

Network connections are a special case, because a network connection (e.g., a TCP connection) always has a unique state:

- at the Transport Layer (OSI Layer 4 (*Open Systems Interconnection Reference Model*) [25]) the sequence and acknowledgment numbers are incremented with every request
- at the Application Layer (Layer 7), depending on the used protocol (e.g., FTP, SMTP, etc.)

Because of this, it is almost impossible to reset the system to a previous state, leaving the environment (servers, etc.) untouched: the sequence / acknowledgment numbers would not match, the TCP connection would break and even if this is not the case, the connected server may be confused about unspecified behaviour of the client and abort the connection / session; the test-subject would behave in a different way, because its environment does not behave as usual. This, of course, is exactly, what we would not like to occur.

Thus, we have a problem at the boundaries between the program and its used resources. If we manipulate the program, we have to manipulate the system around the program too. How and where this could be done, is discussed in the next sections.

5.2.1 Where to Manipulate the System

If we wish to reset network connections to a previous state, we have to make sure that the system is in a consistent state, i.e., all shared objects (e.g., sockets) are perceived in the same way by user mode and kernel mode. If the test-subjects were reset without considering the environment, the system would not be in such a consistent state. As a result, the test subject would behave in a way that is different from a real system. To ensure that the system is in a consistent state, we have to manipulate it. There are three ways of doing this:

- The user-mode part of our program could be reset and the endpoints of the established connections (i.e., the servers, etc.) could be controlled. However, in this case, we face consistency problems at OSI Layer 7 (e.g., suppose that a test-subject starts creating a SMTP session, is reset and sends the EHLO a second time - the server will return an error). This problem could be solved by implementing more tolerant versions of all protocols that the test subject is using, which would result in a significant coding effort. Beside this, the internal state of the sockets in the kernel may cause problems.

- To avoid the different states in user-mode and the kernel, the complete system (i.e., the complete Windows) could be reset. Obviously, this would induce more problems than it could solve: consistency problems at OSI Layer 4 and 7 (additionally to all Layer 7 protocols, our own fault tolerant TCP/IP stack would have to be implemented), as well as considerable restrictions on our performance (states with more than 256MB would have to be saved).

- Finally, only the user-mode part of our program is reset, and the program state is separated from the OS view, so that the kernel is not aware of the state of the sockets used. Consequently, there cannot be any inconsistencies. To achieve this, the complete network interaction between a user-mode process and the kernel has to be imitated. In other words, network connections have to be virtualized.

5.2.2 Where to Virtualize the Network in User-Mode

For the third approach, we have to decide where in user-mode to hook and replay the interaction of the test subject with its environment. One possibility would have been the Winsock2 [26] interface, which is well documented [27]. Thus, it should be easy to generate the behaviour of its functions. However, this option has some drawbacks:

- Not all programs will use the Winsock2 lib. Many will do so, but we wish to have a solution that is as general as possible.

- The Winsock2 library has many functions that would have to be hooked and replayed. This would result in much tedious and error-prone coding work.

- The Winsock2 library is "high level". We would have to solve problems such as synchronisation that cannot be solved with the abilities of Winsock2[1]. So we would have to operate at different levels and that would be error prone too.

- We do not have a "real" handle for our simulated socket of which the kernel is aware of (the handle returned by a simulated `socket()`[27] function). If the system returns a handle that is identical to our "faked" one, this may result in an error.

Given the considerations above, we have decided to choose a more low level location to place our hooks and simulate the network functionality. This location should be as low level as possible, but not inside the kernel (we do not wish to reset anything inside the kernel). This removes the possibility of the documented TDI (Transport Driver Interface) [28], which runs inside the kernel. With these requirements, we have to choose the system call interface (implemented inside the `ntdll.dll`) to place our hooks (like most of the functions hooked by TTAnalyze).

[1]As we will see later, to replay synchronisation we use functions from `ntdll.dll` and there are no corresponding functions in the Winsock2 library

Thus, we wish to virtualize the network at system call level. As described in section 5.1.3, the imitation of system calls is not easy, because we have to understand exactly, which function call has which effects and how we could imitate these affects. In the next section it is described in detail, how networking is implemented at system call level and what is needed to know to imitate all these system calls.

Chapter 6

Networking at System Call Level

Till now, we showed why we wish to be able to virtualize network connections and have presented all the basics to understand the details presented now. Thus, this is the main chapter of this thesis and will describe our work in most detail.

First, the system calls that are used to implement networking under Windows are presented and described detailed. Following, the code sample from Listing 4.1 is discussed again, but this time the focus are the used system calls. Then, the main Control Codes used by AFD are discussed. In the last sections, Synchronisation and Local Procedure Calls are discussed.

6.1 System Calls for Networking

As discussed in the previous sections, we aim to virtualize network calls at the system call level. In this chapter, the system calls which are used by Winsock to implement its functionality are analyzed. One of the functions that are needed have already been discussed: `NtCreateFile()`. This function will not be mentioned in this chapter any more.

6.1.1 NtDeviceIoControlFile()

The complete I/O commands for networking except the creation and the closing of the sockets (i.e., connect, send, receive, etc.) are passed into the kernel via only one function[1]: `ntdll!NtDeviceIoControlFile()` [12], using an interface called AFD (Ancillary Function Driver for WinSock). Unfortunately, this interface is undocumented. The AFD driver *is responsible for the connection and buffer management needed to provide a sockets-style interface to an application* [29].

```
NTSTATUS NtDeviceIoControlFile(
  IN  HANDLE FileHandle,
```

[1]This may result from the fact that Microsoft estimated networking in the beginning not as important as e.g., UNIX

```
  IN  HANDLE Event OPTIONAL,
  IN  PIO_APC_ROUTINE ApcRoutine OPTIONAL,
  IN  PVOID ApcContext OPTIONAL,
  OUT PIO_STATUS_BLOCK IoStatusBlock,
  IN  ULONG IoControlCode,
  IN  PVOID InputBuffer OPTIONAL,
  IN  ULONG InputBufferLength,
  OUT PVOID OutputBuffer OPTIONAL,
  IN  ULONG OutputBufferLength
);
```

Listing 6.1: `ntdll.dll NtDeviceIoControlFile()`

FileHandle

The first parameter is a handle, representing the socket. This handle must have been created previously by calling `NtCreateFile()`. Furthermore, there are other types of handles (which do not represent a socket) that can be used to accomplish some network tasks. These tasks are always asynchronous tasks - for details see `AFD_CONNECT`, `AFD_SELECT`.

Event

An optional Event is passed in, if it is necessary that the requested operation is completed before the program can proceed. If `NtDeviceIoControlFile()` returns `STATUS_PENDING`, the requested action is not finished already, but the action was started successfully. A thread can wait for the completion of its request by calling `NtWaitForSingleObject()`, using this event. Obviously, this is not a must, there are other synchronisation methods without using events (see Section **??**).

ApcRoutine

The third parameter is always `NULL` for network calls.

ApcContext

is used for asynchronous completion messages. This will be discussed in more detail in I/O Completion (Section 6.4.2).

IoStatusBlock

The fifth parameter, a pointer to a caller-allocated structure IO_STATUS_BLOCK, is especially important for functions returning `STATUS_PENDING`. It is a special memory area that remains valid until the requested action is completed, which may be longer than `NtDeviceIoControlFile()` needs to execute. I.e., when `NtWaitForSingleObject()` (called to wait for the completion of a started task) returns, the `IO_STATUS_BLOCK` contains information about the status of the initial `NtDeviceIoControlCode()` call.

```
typedef struct _IO_STATUS_BLOCK {
        NTSTATUS status;
        ULONG    information;
} IO_STATUS_BLOCK, *PIO_STATUS_BLOCK;
```

Listing 6.2: `I/O status block`

The field `status` contains values such as `STATUS_SUCCESS`, `STATUS_CONNECTION_RESET`. Depending on the IoControlCode and the `status` (i.e., if the function call was successful), `information` contains some more information about the result, e.g., for an `AFD_SEND` how many bytes have been sent.

IoControlCode

is for networking one of the AFD codes. It defines the action to be executed (e.g., connect, send, receive, etc.). Among other things, the size and existence of the input- and output buffers can be determined by the IoControlCode. The AFD IoControlCodes are described in more detail in the next section.

Inputbuffer

The `InputBuffer` is a pointer to a caller allocated memory. It is used to submit more information, such as the buffer to be sent for a TCP send. The pointer may be null, but only if the parameter `InputBufferLength` indicates a buffer size of 0.

Outputbuffer

Like the `InputBuffer`, the `OutputBuffer` is a pointer to a caller allocated memory[2]. Therefore, even the existence and the size of the output buffer are determined by the caller and the caller has therefore to know, how many bytes of output buffer are needed by the kernel. There are even cases, where `InputBuffer` and `OutputBuffer` point to the same memory.

The `OutputBuffer` is used by the kernel so submit informations back to the caller, that cannot be transmitted over the return value (which only indicates if the function was successful or if an error occured). Just like for the `InputBuffer`, `OutputBuffer` may be null, if `OutputBufferLength` indicates a buffer length of 0.

6.1.2 NtWaitForSingleObject()

This function has already been mentioned as being a synchronisation function. It receives a handle for an event (i.e., a handle for an object, created with `NtCreateEvent()`). The event should be in a non-signaled state, otherwise the function returns immediately. `Alertable`

[2]For usual the kernel does not allocate memory for a process except functions like `malloc()`

defines, if the event can be signaled from user mode (i.e., if functions such as `ZwSetEvent()` can release the thread from blocking). The "big brother" `NtWaitForMultipleObjects()` receives one or more event handles and blocks as long as all of them are getting signaled.

```
NTSTATUS NtWaitForSingleObject(
  IN  HANDLE Handle,
  IN  BOOLEAN Alertable,
  IN  PLARGE_INTEGER Timeout OPTIONAL
);
```

Listing 6.3: `ntdll.dll NtWaitForSingleObject()`

With this function, Winsock is able to let a thread block as long as it takes for the action to be executed. The question is, how Winsock knows if the operation has been successful and if not, which error has occurred. That is what the IoStatusBlock is for: The result of the operation requested by `NtDeviceIoControlFile()` is written to this structure.

6.1.3 NtClose()

```
NTSTATUS NtClose(
  IN  HANDLE Handle
);
```

Listing 6.4: `ntdll.dll NtClose()`

`NtClose()` is used, to close handles so that these handles are not valid any more (and the kernel can use this handle for an newly opened object). The closing of the handle does not necessarily mean that the object that is references by the handle is closed and deleted. A `NtClose()` executed for a file handle will result in other actions executed inside the kernel than a `NtClose()` executed for a handle that references a thread (the thread is not terminated by a `NtClose()`).

6.2 Chronology of AFD Control Codes

Now the most important system calls for networking are known, therefore, the code from Listing 4.1 can be discussed at system-call level.

6.2.1 socket()

The first three lines from Listening 4.1 are less interesting - they are needed to setup the Winsock2 environment and do not cause any system call. In Line 4, the socket is created. In behave of this function call, the necessary helper functions are loaded (if this is not already done). For the creation of the socket, the first system call is executed: `ntdll.dll!NtCreateFile()` is called with ObjectName `\Device\Afd\Endpoint`, the returned file handle is the representation for the created socket, and it is the return value from `socket()`.

On behalf of `socket()`, the first I/O operations are done via `NtDeviceIoControlFile()`: `AFD_GET_INFO (0x1207B)` and `AFD_SET_CONTEXT (0x12047)`. These change the status of the handle in the system, thus, they are less interesting for replaying.

6.2.2 connect()

From Line 5 to 8, we omit the details of setting up our connection, they only work in user mode to set up the definition of the server endpoint. The interesting part of our connection establishment occurs in Line 9: `connect()` first binds the socket (`AFD_BIND (0x12003)`) to a local address[3], performs some administrative tasks (`AFD_SET_CONTEXT` and `AFD_GET_TDI_HANDLES (0x12037)`, and then connects via `AFD_CONNECT (0x12007)`. Here, we have for the first time a return value that is not `STATUS_SUCCESS`: `STATUS_PENDING`. This means that the action was started successful, but is not completed yet. Our simple program is single threaded, so Winsock has to wait until the operation is complete. To achieve this, Winsock calls `NtWaitForSingleObject()`, passes in the event handle from the prior function call (`NtDeviceIoControlFile()` with control code `AFD_CONNECT`). This function will block, until the primarily task (the connection of the local port with the server port) is completed.

After returning from `NtWaitForSingleObject()`, we have again two administrative calls, `AFD_GET_TDI_HANDLES` and `AFD_SET_INFO`.

6.2.3 send() and recv()

The `send()` in Line 10 will send the `sendStr` over our connected socket, causing an `AFD_SEND`, which returns the count of bytes sent in the IoStatusBlock. For usual, this function returns directly and no further synchronisation is necessary. The `recv()` in Line 12 causes an `AFD_RECV`, which receives the caller-allocated buffer where the received data are written to. This function for usual returns `STATUS_PENDING`. As with `AFD_CONNECT`, the completion of this operation is waited for by calling `NtWaitForSingleObject()`. If this function returns, the input buffer of the prior function is not valid any more, only the target buffer, where the received bytes are written to. The number of bytes that have been received is stored in the IoStatusBlock.

6.2.4 Error handling

Errors appearing during execution are reported over the return value of the `NtDeviceIoControlFile()` or, if `ZwWaitForSingleObject()` is used, with the `status` field of the IoStatusBlock. The errors returned by these functions are not the same like the error code received by `WSAGetLastError()`. For example, if a socket cannot be connected, `NtDeviceIoControlFile()` returns `STATUS_CONNECTION_REFUSED (0xC0000236)`, `WSAGet-`

[3]This is done by Winsock automatically, if the user does not do it

`LastError()` uses `ntdll.dll!RtlGetLastWin32Error()` and returns `WSAECONNREFUSED` (`0x274D`).

6.3 IoControlCodes for AFD

In this section, we explore the main IoControlCodes in more detail. We will not discuss all existing control codes, but show, by means of the most important ones, their general behaviour so that it should be easy to determine how other control codes work.

We did not encode the complete AFD structure. We restricted our work to the responses - the reason is that we "only" wish to replay the traffic. So for IoControlCodes such as AFD_SET_CONTEXT, which receives 248 bytes of input buffer, we did not decode the input buffer - it is sufficient to know that after a successful call, the output buffer of 16 bytes is filled with zeros.

6.3.1 AFD_BIND (0x12003)

The bind function associates a local address with a socket. [27] Input- and output buffer are the same memory and 26 bytes long. At offset `0xE` of the input buffer the IP can be found (if `WS2_32!bind()` is not explicitly called, it is executed on behalf of `WS2_32!connect()` and as IP 127.0.0.1 or 0.0.0.0 is used).

With offset `0xC` in the response buffer there is a two byte port number (in network byte order), starting at a non-zero value (mostly slightly larger than `0x400`) that is incremented with every bind. This count is also returned in `AFD_GET_SOCKNAME`. This control code typically does not block.

6.3.2 AFD_CONNECT (0x12007)

For connection establishments, there are two possibilities. A socket can be connected directly, passing in the socket handle to `NtDeviceIoControlFile()` as `FileHandle`. This function usually returns `STATUS_PENDING`. To wait until the connection is established, `NtWaitForSingleObject()` is called.

Alternatively, the connection can be established in an asynchronous way. For this, a separate file has to be created: `\Device\Afd\AsyncConnectHlp`. This is used as function parameter `FileHandle`, the handle for the socket to connect is passed in via the input buffer (as third DWORD - otherwise it is `NULL`). The return value is `STATUS_PENDING` too. For how the application is notified about the completion of the connection establishment, see I/O Completion. Alternatively, to check if the socket is connected, an `AFD_SELECT` can be executed (i.e., check if the socket is write able).

For both options, port and IP Address are found with offset `0x14` and `0x16` in the input buffer.

6.3.3 AFD_SEND (0x1201F)

`AFD_SEND` expects as `FileHandle` a connected socket (so this function is available only for TCP), 16 bytes of input buffer, and no output buffer. The first DWORD of the input buffer is a pointer to a `AFD_WSABUF` that declares the size of the buffer to send and a pointer to the buffer. This function usually blocks until the bytes have been sent; the count of bytes sent is written to the `information` field of the IoStatusBlock.

```
typedef struct _AFD_WSABUF {
    UINT  len;
    PCHAR buf;
} AFD_WSABUF, *PAFD_WSABUF;
```

Listing 6.5: `struct for AFD buffers`

This is the example shown in Figure 3.2, showing a debugging session with WinDbg. For analyzing a TCP send, a breakpoint has to be set for `ntdll!NtDeviceIoControlFile` with condition `poi(@esp + 0n24) = 0x1201f`. This command reads the DWORD from the address which is referenced by the stack pointer plus 24 bytes (the sixth parameter of the function) and compares it with `0x1201f`.

The command `dd @esp` prints the memory where the stack pointer points to. The value `0x719b5908` is the saved instruction pointer, which points to code in the mswsock library, as you can see from the stack trace, shown in the window "Calls" right on the top of the window.

The 7th parameter is the pointer to the input buffer, its content is dumped with the command `dd 0x1effd34`. As you can see from the following memory dumps, the send buffer can be found as described above.

6.3.4 AFD_RECV (0x12017)

As `AFD_SEND`, `AFD_RECV` expects a connected socket as `FileHandle`, no output buffer and 16 bytes of input buffer:

```
typedef struct _AFD_RECV_INFO {
    PAFD_WSABUF     BufferArray;
    ULONG           BufferCount;
    ULONG           AfdFlags;
    ULONG           TdiFlags;
} AFD_RECV_INFO , *PAFD_RECV_INFO ;
```

Listing 6.6: `input buffer for AFD_RECV`

As for `AFD_SEND`, the first DWORD is a pointer to a `AFD_WSABUF` struct, whereas the `BufferCount` with our test subjects always was `0x1`[4]. The `AfdFlags` were always zero, for the `TdiFlags` we observed two possibilities:

[4]`WSARecv()` and `WSASend()` from Winsock2 offer the possibility, to pass in several buffers. These functions are used e.g., with the WSAAsyncSelect Model [30]

- `TDI_RECEIVE_NORMAL (0x20)` is used for the normal receive: The kernel fills the buffer referenced by `BufferArray` and returns the size of the received bytes in the IoStatusBlock

- `TDI_RECEIVE_NORMAL & TDI_RECEIVE_PEEK (0xA0)` are used to get the status of the device, the input buffer is always only one byte long, as return values we have observed `STATUS_CONNECTION_RESET` and `STATUS_DEVICE_NOT_READY`

6.3.5 AFD_SELECT (0x12024)

This control code is not used by our example function `foo()`, but it is used frequently. One example for using an `AFD_SELECT` control code is `WS2_32!select()`:

```
int  select(
   int  nfds,      //ignored
   fd_set*  readfds,
   fd_set*  writefds,
   fd_set*  exceptfds,
   const  struct  timeval*  timeout
);
```

Listing 6.7: `WS2_32!select()`

The select function determines the status of one or more sockets, waiting if necessary, to perform synchronous I/O [27]. `fd_set` is a structure, containing one or more handles; handles in `readfds`/`writefds` are checked for read- and write-ability, handles in `exceptfds` are checked for connection errors and OOB data [31]. The return value determines the total number of socket handles that are ready and contained in the `fd_set` structures.

Similarly to the connect, for select there are two types of handles that can be used for `FileHandle`:

- A handle to a socket (AFD endpoint), e.g., for `WS2_32!select()` the first handle from the `fd_set` structures

- For asynchronous select (e.g., if the WSAAsyncSelect Model [30] is used), a file handle with ObjectName `\Device\Afd\AsyncSelectHlp` is passed, for details see IoCompletion

Input- and output buffer are always the same. The size depends on the type of the `FileHandle`[5] and how many handles are passed in. The buffer is divided in two parts: a static part, which is always present (16 bytes long), and a dynamic part, which grows with the number of handles passed in (12 bytes for each handle).

For the static part, the first two DWORDs are flags for the kernel, which stay unmodified, so we do not have to replay them. The third DWORD is a counter of how many

[5]For an asynchronous select, there is one more DWORD in the buffer, which was never relevant with respect to our test subjects

handles are in the rest of the buffer. The fourth DWORD may be zero or any other value (because the buffer is mostly allocated from the stack - especially if used by Winsock - and is not set to zero). We replaced this and other values (e.g., see the "not relevant" comment in Listing 6.8) with zeros during execution which did not have any effect, so they do not seem to be relevant.

The second part is an array of structures:

```
typedef struct _AFDSELECT_DYNAMICPART {
        ULONG handle;
        ULONG eventFlags;
        ULONG reserved;    //not relevant
} AFDSELECT_DYNAMICPART;
```

Listing 6.8: `struct for the dynamic part of AFD_SELECT`

where `handle` is a handle for a file from type `\Device\Afd\Endpoint`, `eventFlags` defines which (combination of) events have to be checked for this handle, where possible values are `0x1` for receive, `0x2` for OOB receive, `0x4` for send, `0x8` for disconnect, `0x10` for abort, `0x20` for close, `0x40` for connect, `0x80` for accept, `0x100` for connection failures, etc. One handle can be passed in several times, e.g., if `select()` from Winsock is used and a handle is contained in `writefds` and `exceptfs`, it will be passed in twice, with `eventFlags` `0x4` and `0x102`.

In contrast to other IoControlCodes, for `AFD_SELECT` the (output) buffer keeps valid until `ZwWaitForSingleObject()` has returned with `STATUS_SUCCESS`. If the request itself was successful, the `information` of the IoStatusBlock is `0x1C`, otherwise it is `0x10`. For the reply, the setup of the buffer is the same: The third DWORD of the buffer is a count, how many handles are contained in the buffer. The `eventFlags` contains the occurred event. The replied event may not always be requested, for example, OOB receive is replied with `0x1`, `0x39` may replied with `0x80`.

6.3.6 AFD_UDP_SEND (0x12023)

Sending UDP packets requires a similar setup as with TCP connection (obviously, no connect is necessary). The creation of a socket results in the creation of file with ObjectName `\Device\Afd\Endpoint` and the execution of `AFD_GET_INFO` and `AFD_SET_CONTEXT`. As with TCP sockets, if no bind is done by the application, this is automatically executed by Winsock. For TCP, on behalf of the connect, for UDP this is done on behalf of `sendto()`. After the bind, some administrative tasks are executed: `AFD_GET_TDI_HANDLES`, `AFD_SET_INFO`, and `AFD_GET_SOCK_NAME`.

The real send works, as expected for a stateless protocol, with a single function call `AFD_UDP_SEND`. As for TCP send, the input buffer contains a pointer to a `AFD_WSABUF`, at offset `0x34` is a pointer to a buffer that contains at offset `0x8` the destination port and at offset `0xA` the destination IP. The IoStatusBlock contains, as for TCP send, the number of the bytes sent.

6.3.7 ICMP

ICMP requests are completely different to TCP/UDP network calls. `ping.exe` creates a file with ObjectName `\Device\Ip` and executes `NtDeviceIoControlFile()` with IoControlCode `0x120000`, whereas the IP is the first DWORD in the input buffer.

6.4 Synchronisation

The major problem for replaying network connections is to emulate the synchronisation of threads. There are different kinds of issues:

- The operations that block and release threads are executed in the kernel. Our objective is to leave the kernel untouched, thus, the kernel does not know anything about the status of our simulated socket. Therefore, it is not possible to let a function such as `TCP_SEND` execute to achieve a blocking thread.
- There are many possibilities to implement synchronisation. Microsoft's Internet Explorer and Mozilla Firefox[6] implement their own method with local TCP sockets to signal blocking threads, Opera uses the available routines of the Winsock library (WSAAyncSelect approach [30]). Thus, it is difficult to provide a general solution to this problem.

Before the solutions we found are discussed, the main mechanisms that the Windows kernel provides for synchronisation are shown.

6.4.1 Waiting for Event Objects

The main aspects of waiting for event objects have already been discussed. For the sake of completeness, let us briefly recapitulate them: I/O operations for network operations are always started with `NtDeviceIoControlFile()`, which may return `STATUS_PENDING` before the operations are complete. To wait for the completion of one or more actions, `NtWaitForSingleObject()` (or `NtWaitForMultipleObjects()`) has to be called, passing in the event handle(s) used with the corresponding `NtDeviceIoControlFile()` function call(s). As long as these functions return `STATUS_TIMEOUT` and not `STATUS_SUCCESS`, the call has to be repeated.

6.4.2 I/O Completion

Concepts:

I/O Completion ports are an instrument to implement high-performance server applications [29, 32] (but they are adequate for any multi-threaded application where asynchronous

[6]For the research work applications have been selected whose behaviour we could verify easily

I/O operations are required[7]). For such applications, it is optimal to run as few threads as possible (and therefore, to have a minimum of context switches), but also to run at least a minimum of threads (to have a full load of all CPUs). Consider a web server that processes every incoming request with a dedicated thread. The Windows kernel provides with I/O Completion a solution, where the kernel itself keeps track of running and waiting threads. Threads waiting to process an incoming request are blocked as long as at least the minimum of threads are running (i.e., a thread is released, if e.g., for eight CPUs only seven threads are running).

For this, a completion port is associated with a server socket, incoming requests cause a message to the completion port, from where they can be obtained by calling `ZwRemoveIoCompletion()`. By blocking this function call, the windows kernel achieves the behaviour described above.

Practical aspects:

First, a new port has to be created with `ZwCreateIoCompletion()`, receiving among other parameters the maximum number of threads allowed to run. In the kernel, there it is a queue on which completion packets can be pushed (`KeInsertQueue()` in the kernel, `ZwSetIoCompletion()` in user mode) or popped from (`KeRemoveQueue()` and `ZwRemoveIoCompletion()`). If no completion packet on the queue is available, or the maximum of running threads is reached, threads calling `ZwRemoveIoCompletion()` will block; they are released in first in - last out order.

```
NTSTATUS ZwRemoveIoCompletion (
  IN  HANDLE IoCompletionHandle,
  OUT PULONG CompletionKey,
  OUT PULONG CompletionValue,
  OUT PIO_STATUS_BLOCK IoStatusBlock,
  IN  PLARGE_INTEGER Timeout OPTIONAL
);
```

Listing 6.9: `ntdll!ZwRemoveIoCompletion`

Handles for which asynchronous actions can be executed are associated with a port using the function `NtSetInformationFile()`, the InformationClass `FileCompletionInformation (0x1E)` is set. For completed asynchronous I/O operations a completion packet is queued to the port, whereas the key is determined by second DWORD in the input buffer of `NtSetInformationFile()`, the `CompletionValue` is the `ApcContext` from the corresponding `NtDeviceIoControlFile()`, and the `IoStatusBlock` contains as usual the information about the executed action.

For AFD, note that if any `\Device\Afd\AsyncXXXHlp` file is created, an I/O completion port is created (and saved in `ntdll!WorkerCompletionPort()`). The CompletionKeys are function pointers and set depending on the type of asynchronous action they are used for,

[7]These ports can also be used for synchronization of threads without ever consuming the abilities for I/O operations. Microsoft's Internet Explorer uses I/O Completion ports partially in this way

e.g., for `\Device\Afd\AsyncConnectHlp` the key is a pointer to `mswsock!SockAsyncConnectCompletion()`[8]. These functions are executed after `ZwRemoveIoCompletion()`, whereas the `CompletionValue` is a function parameter.

6.4.3 Replaying Synchronisation

In order to replay replaying synchronisation, the following cases have to be distinguished:

- if a `ZwWaitForSingleObject()` call has to block, it is passed into the kernel. To release it, we insert a function `ZwPulseEvent()`, which releases the thread and sets the event to unsignaled state (if we only release the thread, the next `ZwWaitForSingleObject()` with this event would return immediately).
- `NtDeviceIoControlFile()` function calls that have to block, are not passed into the kernel. They are substituted by `ZwWaitForSingleObject()` (how this is done, see below), as event parameter the event from the original function is used. (Up to now, we have not found a function that blocks without having an event passed in.) The thread is released the same way as with "original" `ZwWaitForSingleObject()` functions.
- For `ZwRemoveIoCompletion()` calls the kernel will not insert any completion messages on the queue, so we have to do it manually. This is done with `ZwSetIoCompletion()`, we only have to decide, when to do this. If we need the call non-blocking, we insert the set call before the remove is executed. If we need it blocking, we insert the set call at another thread, when we need the remove to be released.

Substitution of function calls

is needed to replace a function call temporarily with another one: For the caller it should look as if the original function was executed (e.g., `NtDeviceIoControlFile()`), but we let another function execute and pass into the kernel (e.g., `ZwWaitForSingleObject()`). To achieve this, we manipulate the system at the moment, when the instruction pointer is at the first instruction of the function to be substituted: We push the function parameters on the stack, reset the instruction pointer; as saved instruction pointer we use the current one: if for the corresponding thread the instruction pointer is back at our replaced function, we know that we can replay the result of the substituted function and let this function return.

Inserting function calls

works similarly to the substitution of function calls. The main difference is the saved instruction pointer and the fact that we do not need to memorise anything about the inserted function: as with substitution, we push the function parameters on the stack and reset the instruction pointer, for the saved instruction pointer we use the actual one. If the inserted function is processed, the execution processes as if nothing happened.

[8]This function is not exported, but can be found in the symbol files for WinDbg

6.5 Local Procedure Call (LPC)

Resolving domain names [24] for example with `gethostbyname()` from Winsock, is not performed by the program itself instead, such queries are executed by `services.exe` to cache domain names system wide. The interprocess communication is done with an undocumented interface called LPC, "Local Procedure Call" [16].

The communication with LPC runs over ports. Typically, the server application creates a named port, and the client application connects to this port via the appropriate name. For DNS Queries, Winsock connects with `ZwConnectPort()` to the port `\RPC Control\DNSResolver`. The communication goes over the function `ZwRequestWaitReplyPort()` (send a request and wait/block for the reply).

We did not reverse engineer most of the requests - there is a lot of traffic, before the real DNS query is executed. The actual query is executed if the value of the third DWORD in the request buffer is `0x90241`. The domain name is delivered as unicode string at offset `0x38`, the length of the string is found at offset `0x34`.

For the response there are two possibilities:

- If the least significant byte of the third DWORD is `0x1`, then the response is directly in the response buffer with offset `0x24`.
- If this byte is `0x4`, a second call to `ZwRequestWaitReplyPort()` is necessary. In the input buffer at offset `0x28` there is a pointer to a buffer and a length declaration. This buffer contains the response in the same format as with the direct variant.

In this buffer, with offset of `0xB`, an array of 32 byte long block starts. The resolved IP Addresses are written in blocks that start with a DWORD of `0x4001`. The second DWORD is always `0x9`, except the first, which is `0x2009`. The resolved IP is found at offset `0x14`

Chapter 7

Implementation

In this chapter the most interesting and significant implementation details are described. The first part starts with details about the changes for the Generator, describe the implementation of the manual decoding of function parameters and give a short instruction to the decoding of Local Procedure Calls and DNS queries. The second part describes the implementation of the virtualizing of the Network.

7.1 Hooking Network Access

As described in section 6, accessing the network is done via `NtDeviceIoControlFile()`. The IoControlCode defines the action that is executed, and the size and existence of the input and output buffers. Obviously, there is a problem with the Generator, one of the sub systems of TTAnalyze. As described in 2.3.2, the Generator is only able to generate code for function parameters, which can be defined statically. For AFD this is not the case, because the structures of the input buffers vary with the IoControlCode, and even these structures can not always be defined in a static way (e.g., if `BufferCount` for `AFD_RECV` (see Listing 6.6 is larger than one). For hooking network access, we have to change the Generator, so that it is able to read the function parameters from functions, which cannot be defined in a statically way.

7.1.1 Changes of the Generator

For changing the Generator out of reasons described above, there are mainly two possible solutions:

- The behaviour of the Generator (reading only static structures form the virtual memory) is modified to a dynamic behaviour, thus, its behaviour has to depend on certain function parameters that it has already read from the virtual system. Therefore, it has to be a way found, to define under which circumstances the generator should behave in which way.

- To avoid the need of such a definition, reading of non-static function parameters is implemented by hand and the Generator is modified in such a way that for certain functions (which are rare) the code generated by the Generator is calling the manual implemented code.

The first solution is certainly the more elegant one, but this solution would result in much coding work. Till now, the Generator has to read and parse the definitions of the function it has to hook. These function definitions are defined in a classic C/C++ style, so for parsing existing tools are used [33]. For reading non-static data from the virtual stack, it would be necessary, to find a syntax that is able to describe the functionality we need[1], and extending the existing code and tools with the abilities, to read and handle these additional syntax.

After some evaluation we found that it is more efficient to choose the second approach[2]. Thus, the Generator now checks if currently the function `NtDeviceIoControlFile()` is generated. If yes, some code is inserted that checks if `(IoControlCode & 0xFFFFF000) == 0x12000`[3]. If this is the case, the generated code calls a function that contains the manual implemented code, reading the function parameters corresponding to the IoControlCode from the stack.

7.1.2 Manual Decoding Function Parameters

As described in the preceding section, the decoding of function parameters that cannot be defined statically is done manually. The Generator has been modified, so that for certain functions (in the case of networking `NtDeviceIoControlFile()`) under certain conditions a manual written function is called. These functions are part of several classes, whereas every class is responsible for a specific AFD IoControlCode. E.g., the class `TcpConnectDecoder` is responsible for the AFD Code `AFD_CONNECT (0x12007)`. All these classes are build after the same scheme and have to fulfill following requirements:

- Every class is responsible for one AFD Code and every object (instance of such a class) is responsible for managing one specific function call.
- All these classes have to be thread safe: especially network applications are often multi threaded and may execute several commands of the same type parallel.

[1]This would include that the function definitions include some sort if `if`, `else` and loops to express different scenarios of possible combinations of function parameters

[2]It was difficult to find out which control code is responsible for which action, and to reverse engineer the used structures, whereas the file `include/drivers/afd/shared.h` from ReactOS Source [34] delivered some useful hints. The manual implementation for reading these parameters from the virtual stack has been a relatively small effort

[3]If this condition is true, the handle used for `NtDeviceIoControlFile()` as `FileHandle` has with our test-subjects always been from type `\Device\Afd\XXX`, therefore, it has always been an AFD IoControl-Code

- Every function has to implement two static functions: `functionWasCall(Bit32u threadId)` and `functionHasReturned(Bit32u threadId)`.

To demonstrate these requirements in more details, let us discuss a stripped-down definition of the class `TcpConnectDecoder`.

```
1  class TcpConnectDecoder {
2  private:
3      static map<Bit32u, TcpConnectDecoder*>
                                     tcpConnectDecodersdecoders;
4      unsigned __int32 DeviceHandle_in;
5      void functionWasCalled();
6      void functionHasReturned();
7      TcpConnectDecoder(Bit32u threadId);
8  public:
9      static TcpConnectDecoder* functionWasCalled(Bit32u threadId);
10     static void functionHasReturned(Bit32u threadId);
11 }
```

Listing 7.1: Definition `TcpConnectDecoder`

In Line 9 and 10, the definitions of the two functions every of these classes has to implement can be seen.

The map in Line 3 is defined private static, so that the two public functions are able to access it. This map contains (referenced by the unique `threadId`[4]), all current available TcpConnectDecoder Objects.

Line 4 contains an example definition, which data members such an class could contain. E.g., the current class contains additional data members like `EventHandle_in`, `IPAddress`, `Port`, and `AsyncDeviceHandle`, etc.

In Line 5 and 6 there are the non-static, private functions `functionWasCalled()` and `functionHasReturned()`. These functions do not need the parameter threadId, because the constructor in Line 7 makes the threadId available to the object as class variable.

To demonstrate the usage of this calls, let us take a look at the implementation of the static function `functionWasCalled(Bit32u threadId)`:

```
1  TcpConnectDecoder* TcpConnectDecoder::
                           functionWasCalled(Bit32u threadId) {
2      assert(tcpConnectDecodersdecoders[threadId] == 0);
3      TcpConnectDecoder* decoder = new TcpConnectDecoder(threadId);
4      decoders[threadId] = decoder;
5      decoder->functionWasCalled();
6      return decoder;
```

[4]As `threadId` not the ID referenced by Windows is used, but the address of the Thread Environment Block (TEB) in kernel memory. This address is unique and therefore can be used to distinguish the different threads of a process. This is done to avoid an additional memory read from the virtual system out of performance reasons

```
7 }
```

Listing 7.2: `TcpConnectDecoder::functionWasCalled(Bit32u threadId)`

In Line 2, the `assert()` statement assures that there is no function call waiting for returning (if a function is returning, the corresponding object is deleted after the necessary decoding work is done). If this assertion fails, this would indicate an implementation error. The only reason that a function did not return before it is called as second time (in the same thread) may result from the fact that the function is calling itself in a recursive manner. The function we hook is equate with a system-call, but a system-call cannot call itself recursively. Therefore if this function is called a second time in the same thread, we have to search an implementation error[5].

In Line 3 and 4 a new TcpConnectDecoder object is created. This object receives the information, for which `threadId` it is responsible. This new object is inserted into the existing map. Then, the private function `functionWasCalled()` is called, which is doing the main work: this function reads the function parameters from the stack and informs the object that requested a notification.

The implementation of these private functions depend on the IoControlCode that is executed. To show the general design of such a function, we explore the function `functionWasCalled()` from the class TcpConnectDecoder in more detail.

```
void TcpConnectDecoder::functionWasCalled()
{
    VirtualAddress tmpAddr;
    Bit32u esp = vSys->getESP();
    deviceHandle_in = *reinterpret_cast<unsigned __int32*>
                         (virtualSys->readMemory(esp + 4, 4));
    tmpAddr = *reinterpret_cast<VirtualAddress*>
                         (virtualSys->readMemory(esp + 28,4));
    if (CNetworkAnalyzer::isHandleAsyncConnectHlp(
                                                 DeviceHandle_in))
        ASyncDeviceHandle = *reinterpret_cast<unsigned __int32*>
                         (virtualSys->readMemory(tmpAddr+8,4));
    port = *reinterpret_cast<unsigned __int16*>
                         (virtualSys->readMemory(tmpAddr+20,2));
    IPAddress = *reinterpret_cast<unsigned __int32*>
                         (virtualSys->readMemory(tmpAddr+22,4));
    if (aInfo->getOverwriteIP())
    {
        unsigned __int32 targetIP = aInfo->getOverwriteIP();
        if(IPAddress != targetIP)
            virtualSys->writeMemory(tmpAddr+22, 4,
                reinterpret_cast<unsigned __int8*> &targetIP);
    }
    /* notification of registered functions */
```

[5]Obviously, this assertion never failed

```
19 }
```

Listing 7.3: `TcpConnectDecoder::functionWasCalled()`

This function needs two temporarily variables; `tmpAddr` contains temporary addresses in the virtual system and therefore something like a pointer that points to data in the virtual system; the variable `esp` is equivalent with the `esp` register, which represents the stack pointer. The stack pointer always points to the last used address on the stack, and therefore points (as this is the function begin) to the saved instruction pointer on the stack. After the saved instruction pointer ("after" means up to higher addresses) are the function parameters following. (See Figure 3.2 to get an understanding of how the stack looks if a function is currently started to be executed. This Figure is explained in more detail in Section 6.3.3.) Therefore, in Line 6 we read the first function parameter from the stack, which is the device handle representing the socket[6].

As you can see in Listing 6.1 the 7th parameter of `NtDeviceIoControlFile()` is a pointer to the input buffer. In Line 7 of Listing 7.3 this pointer is read from the virtual system (again, all function parameters are 4 bytes long, plus the saved instruction pointer indicates that we have to read the 4 bytes, beginning at `esp` plus 28 bytes).

In the Lines 8 and 9, first the CNetworkAnalyzer (responsible for analysing the complete network related function calls and therefore hooking `NtCreateFile()` which defines the type of an AFD Handle) is queried if the current used device handle is from type `\Device\Afd\AsyncConnectHlp`. If this is the case, the handle for the `\Device\Afd\Endpoint` with offset `0x8` is read from the input buffer (see `AFD_CONNECT` for details why this is done). In the Lines 10 and 11, the port and the destination address are read from the input buffer.

The code from Lines 12 to 17 are responsible for implementing the option `--overwrite-ip`: The class `CAnalysisInformation` is queried for an IP-address that has to be used to overwrite the destination of the TCP connection. If the return value is 0, this option is not in use, otherwise the IP is overwritten in the virtual memory.

For the function return, `functionHasReturned(Bit32u threadId)` is called, which asserts that for the `threadId` a corresponding object is saved in the map. For this object the private function `functionHasReturned` is called. This function does, like for a function call, the necessary work (e.g., overwrites back the primary IP address, if it has been overwritten at the function call).

7.1.3 LPC - DNS

As described in Section 6.5, Local Procedure Calls are an undocumented interface for inter-process communication. It is relative easy to keep track of all requests and responses that are executed using this interface. Obviously, there are two main problems if the complete interaction between two processes has to be monitored and interpreted automatically.

[6]The function `readMemory()` expects as function parameters the address from where the memory has to be read and a count, how many bytes have to be read from the system. `esp + 4` results from the fact that the saved instruction pointer is 4 bytes long, just like the device handle

- With every request / response it is possible to exchange a small amount of data (496 bytes)[7]. This may not be enough for all processes using LPC, therefore, processes can use sections to exchange data. These sections are memory areas, that are mapped in the address space of both processes. These areas are permanently valid in both processes, so it is difficult to keep track of data exchanges. As long as these data areas are only used after a corresponding request or response, the sections could be dumped and the changes could be logged. Furthermore, it is possible that some kind of polling is used. In this case, the executed code has to be analyzed at assembler code level, to recognise read and writes at the corresponding memory areas.
- Just like LPC, the protocols used to exchange data over LPC are undocumented or even a self-contained implementation by a malware author. In both cases it is difficult to analyze and evaluate the content and generate a meaningful and significant report about the executed interprocess communication.

These problems have been encountered during the analysis of LPC. For DNS queries no sections are used (if the response is longer than 496 bytes, a second request / response pair is executed), but it is difficult to understand the complete traffic that is exchanged between the client process and `services.exe`. Thus, a LPC Analyzer has been implemented, which is able to keep track of all registered server- and client ports and the data traffic. Obviously, the analysis of the data has to be done manually. For the DNS traffic, it is possible to recognise queried domain names and manipulate the response (this is done, if the option `--overwrite-ip` is used).

7.2 Virtualising Network Access

The functionality described in section 7.1 is only able to monitor network access[8]. Our goal described in the Introduction and more detailed in Chapter 5 is to virtualize network connections. Therefore, the first step has been to find out how in Windows network access is working. The second step has been to test the accumulated knowledge with many different test-executables and to verify that we are able to capture all of the network traffic. For this, we did not need to understand all of the control codes and all of its responses - it was sufficient to decode `AFD_CONNECT`, `AFD_SEND`, `AFD_RECV`, etc.

The third, more difficult task has been to understand the response of all control codes used, to be able to virtualize the network access[9]. The forth, and most challenging task is to replay the synchronisation of multiple threads. Before these problems are discussed, we have to look at some changes of TTAnalyze.

[7]With every request/response a `PORT_MESSAGE` is transmitted between the two processes. This `PORT_MESSAGE` is 512 bytes long, and 16 bytes are used for a header

[8]If it is not necessary to know which process causes which network traffic, it is more easy to capture the network traffic with a network sniffer, such as Ethereal [35]

[9]We do not need to understand all of the input, as long as we are able to generate a correct response

7.2.1 Changes of TTAnalyze for Virtualising

As described in section 5.1.3, for Virtualising the Network Access we need to imitate the execution of system calls. For this, we have to manipulate the stack pointer (`esp` register), the stack itself and parts of the virtual memory in general, as well as the instruction pointer (`eip` register). The first two tasks can be accomplished without any changes to TTAnalyze, but the third task, manipulating the instruction pointer, cannot be implemented with the existing system. This results out of two problems:

- The instruction pointer is not accessible from TTAnalyze. This results from the architecture of QEMU, where the instruction pointer is tightly coupled with the translation block. Therefore, the instruction pointer cannot be changed in every part of the execution without considering the current translation block.
- The callback mechanism of TTAnalyze manipulates the translation blocks. Therefore, if the notification of the execution of, e.g., `NtDeviceIoControlFile()` is received, the corresponding translation block is already executing and it is difficult to intercept the execution of this translation block. In this situation, the actual instruction pointer is saved in a non-public variable and therefore cannot be manipulated.

Out of this, the callback mechanism for function calls that may be intercepted has to be changed in a way that we are able to change the instruction pointer. For this, the QEMU code has to be changed on a location before a translation block is executed and where it is possible to manipulate the instruction pointer without that it is necessary to consider anything about translation blocks. The best place for this manipulation is in the file `cpu-exec.c` before the translation block for the next execution is searched and executed (in line 435 for version 0.7.1).

```
unsigned __int32 tmpVar = 0;
if ( (env->cr[3]& 0xFFFFF000) == mq_ts_pagedir_addr
                              && entryPointReached ) {
    if ( env->eip == ntDevIoEntryPoint ) {
        cpu_memory_rw_debug(env, env->regs[R_ESP] + 24,
                             (uint8_t*) &tmpVar, 4, 0);
        if ( (tmpVar & 0xFFFFFF00) == 0x12000 ) {
            int funcResult = afdNetworkReplay();
            if (funcResult < 0xFFFFFFFF)
                pc = funcResult;
       }
    }
    else if ( env->eip == zwWaitFSingleObjEntryPoint) {
        int replay = syncReplay();
        if (replay)
            pc = replay;
    }
    else if ( env->eip == zwRemoveIoComplEntryPoint ) {
```

```
17          int replay = removeIoCompCallback();
18          if (replay)
19             pc = replay;
20    }
21    else if ( (tmpVar = isInsertedWaitFunc()) )
22          pc = tmpVar;
23 }
```

Listing 7.4: Changes to `cpu-exec.c`

Listing 7.4 shows the complete changes to `cpu-exec.c`, but for now only the Lines 1 to 10 are discussed, Lines 11 to 23 are needed for tasks described later.

In Line 2 it is checked, if the process of the test-subject is running (with the aim of the `cr3` register, described in section 2.3.2) and if the entry point of the process has already been reached (and therefore, if the analysis is already running). Variables such as `mq_ts_pagedir_addr`, `entryPointReached`, or `ntDevIoEntryPoint` are defined in the `qemu.dll` and set during startup from TTAnalyze.

The code from Lines 3 and 4 is responsible to check, if currently the function `NtDevice-IoControlFile()` is called (if the instruction pointer points to the entry point of this function), and if yes, to read the sixth parameter from the stack - the IoControlCode. If the IoControlCode is an AFD control code (a logical and with `0xFFFFFF00` is `0x12000`), TTAnalyze is called back. TTAnalyze returns 0, if no modification of the system has to be executed and a non-zero value, if the system has to be manipulated. The return value is the instruction pointer, where the execution has to proceed. This value is saved in `pc` (program counter), which is a variable from QEMU representing the current instruction pointer[10].

Obviously, the function `afdNetworkReplay()` has to be implemented in TTAnalyze.

7.2.2 Record Mode

For virtualizing the network access, there are two methods implemented, whereas these both can be mixed up without any effort.

- Record and Replay Mode: For this virtualizing technique, in a first run a record of the interaction between user-mode and kernel-mode is created (i.e., all significant function calls and their parameters are saved and written to a file). In the second run, these data are used to simulate the network traffic.
- Simulation Mode: In this mode, the complete interaction of user-mode and kernel-mode is simulated. This means that the user has to know, when things have to happen. This option can be used for simulating a changed environment.

In this section, we describe the Record Mode of the first technique. In the next sections the Replay Mode and the Simulation Mode are described.

[10]All other changes are accomplished in TTAnalyze

For Record Mode, the functions are hooked as usual (the variable `ntDevIoEntryPoint` from Listing 7.4 is set to zero, so the code from this Listing will never be executed), and for the relevant functions the significant data (e.g., function parameters) are saved in a file. This is done by the class `CNetworkRecord`, which encapsulates the complete record and loading functionality. Before this class and its function is discussed in more detail, let us review the functions that are hooked.

Functions recorded for Replay

For virtualizing the network access, it is not only necessary to record the function calls of `NtDeviceIoControlFile()`, it is also needed to record the functions that are used for synchronisation.

First, the decision which `NtDeviceIoControlFile()` function calls have to be recorded is relative easy - all function calls with a corresponding IoControlCode are recorded. Furthermore, we keep track of all AFD devices (a file with an ObjectName of `\Device\Afd\XXX`) by hooking the function `NtCreateFile()`. With our test-subjects we did not find any sample, where the results of both techniques did not match.

The first function for synchronization is `NtWaitForSingleObject()`, see Listing 6.3. All functions that receive an event-handle that has been used for a call of `NtDeviceIoControlFile()` for an AFD device are recorded too.

For more advanced synchronization technologies (I/O Completion, see chapter 6.4.2) we have to hook the functions `NtCreateIoCompletion()`, `NtSetIoCompletion()`, `NtRemoveIoCompletion()`, and `NtSetInformationFile()`.

`NtCreateIoCompletion()` is needed to keep track of all handles for I/O Completion Ports. As described in section 6.4.2, I/O Completion Ports are associated with AFD devices via a call of `NtSetInformationFile()`. Therefore, we record all function calls of `NtSetIoCompletion()` and `NtRemoveIoCompletion()` that use a port handle that has been associated an AFD device. Obviously, if an I/O Completion Port is closed with `NtClose()` we remove this handle from the list of handles to record.

Which data to record

Untill now we have seen, which functions we are recording, but we did not discuss which data we are recording. This is what the structure `RecordItemHeader` is for:

```
typedef struct _RecordItemHeader {
    unsigned __int16 type;
    unsigned __int16 flags;
    unsigned __int32 argA;
    unsigned __int32 argB;
    unsigned __int32 argC;
    unsigned __int32 threadId;
    unsigned __int32 functionResult;
    unsigned __int32 inputBufferLength;
    unsigned __int32 outputBufferLength;
    unsigned __int32 additionalBuffers;
} RecordItemHeader;

#define TYP_NTDEVICEIOCONTROLFILE    0x1
#define TYP_ZWWAITFORSINGLEOBJECT    0x2
#define TYP_ZWSETIOCOMPLETION        0x4
#define TYP_ZWREMOVEIOCOMPLETION     0x8

#define FLAG_RETURNED_DIRECTLY       0x1
#define FLAG_BLOCKED                 0x2
#define FLAG_RETURNED_FROM_BLOCKED   0x4
```

Listing 7.5: Structure `RecordItemHeader`

This structure represents a function call or a function return. The `type` defines the type of the function, e.g., `TYP_NTDEVICEIOCONTROLFILE 0x1` for `NtDeviceIoControlFile()`.

The `flags` define how the function behaves during the recording. If the function returned directly (i.e., no other function was called or returned while the function executed), `FLAG_RETURNED_DIRECTLY 1` is used. `FLAG_BLOCKED` is used, if the function was called and before it returned, an other function was called or returned (i.e., an other thread was executed during the execution of this function and therefore the thread of this function blocked). `FLAG_RETURNED_FROM_BLOCKED` is used for a function that returned from a blocking state (i.e., a function that was recorded with `FLAG_BLOCKED` has to return with flag `FLAG_RETURNED_FROM_BLOCKED`). All definitions for the types and flags are designed in a way so that they can be used as bit masks. This is needed for the Replay Mode in `CNetworkRecord` class for the search algorithm (if e.g., multiple types are searched).

The variables `argA` to `argC` receive different values, depending on the type of the function.

Function	*argA*	*argB*	*argC*
`TYP_NTDEVICEIOCONTROLFILE`	IoControlCode	Event handle	File Handle
`TYP_ZWWAITFORSINGLEOBJECT`	Event handle	IoControlCode	
`TYP_ZWSETIOCOMPLETION`	Port Handle	Completion Key	Completion Value
`TYP_ZWREMOVEIOCOMPLETION`	Port Handle	Completion Key	Completion Value

Table 7.1: Meaning of `argA` to `argC` values for all function types

The IoControlCode for `TYP_ZWWAITFORSINGLEOBJECT` is the IoControlCode for the corresponding `NtDeviceIoControlFile()` - it is needed for a more precise mapping at Replay Mode.

The variables `threadId` and `functionResult` are self-explanatory. Obviously, functions with a flag `FLAG_BLOCKED` cannot have a function result.

The variables `inputBufferLength`, `outputBufferLength`, and `additionalBuffers` are only needed for `TYP_NTDEVICEIOCONTROLFILE`. If no (additional) buffers are used, these values are 0. The additional Buffers are needed for function calls, where the input and output buffers contain pointers to other regions of memory which have to be recorded too (e.g., for `TCP_SEND`).

This structure is part of the `CNetworkItem` class. This class provides the functions to set and read all parameters and implements the writing and reading from a file. As minimum, the structure is written, the buffers (if present) are written immediately after the structure. For this, the `operator<<` and the `operator>>` are overloaded.

CNetworkRecord

The class `CNetworkRecord` is responsible for managing the recording - for creating and for reading the records. The functionality for reading the record are described in section 7.2.3. Beyond the creation and filling of the record file, the core role of `CNetworkRecord` is to act as buffer for `CNetworkItems`. `CNetworkRecord` is notified about called and returning functions and has to decide, when to write a function to a file and with which flags. Therefore, for every thread there may be a `CNetworkItem` buffered.

7.2.3 Replay Mode

For the Replay Mode, the core class is, as for Record Mode, `CNetworkRecord`. This class keeps track of all called functions that have to be replayed. The main challenge is to create a mapping from the "old system" (the recorded system) to the "new system" (the replayed system). This mapping includes threads and handles for files, events, and I/O Completion Ports. As long as only one thread is used (as in our simple function `foo()` from Listing 4.1), the mapping and therefore the replay is determinable and works without problems. As more threads and different types of synchronisation are used, the more complex is the mapping between old and new system and therefore the decision which `CNetworkItem` should be used for a concrete replay.

First, the record file is read and all contained `CNetworkItem`s are saved in a vector. If e.g., the class `CAfdDecoder` receives a call of `NtDeviceIoControlFile()` for an AFD device, this function call has to be replayed. For this, `CNetworkRecord::getRecordItem()` is queried to search for the corresponding `CNetworkItem`. This function receives some informations about the function to be replayed: `type`, `threadID`, `argA` (i.e., the IoControlCode for `TYP_NTDEVICEIOCONTROLFILE`) and the addresses of the input- and output buffer, and the IoStatusBlock (these are needed for blocking calls of `NtDeviceIoControlFile` to know where these buffers are located).

Before the `type` and `argA` are used for a search, it is checked if the `threadId` is already mapped. If not, a non-mapped thread ID is searched, whereas for this search the `type` and `argA` are needed too. The search starts with the first `CNetworkItem` and goes toward the last one. The `threadId` of the first element, where the thread ID is not already mapped and `type` and `argA` are matching, is used for the new mapping.

For every thread there is a "current position" saved, which describes the current position in the execution. Therefore if a thread is mapped, the next item is always search towards the end. The success of a replay depends mainly on this algorithm, if the thread mappings are correct.

Insertion of function calls

As described in section 6.4.3, we sometimes need to insert function calls. This does not work with the method used for TTAnalyze. During the work on TTAnalyse we found that the current implementation of function insertion does only work as long as no thread switch happens during the execution of the inserted function. The function we need to insert sometimes will cause an thread switch (e.g., `WaitForSingleObject()` to simulate a blocking `NtDeviceIoControlFile()` are inserted to cause an thread switch), therefore this approach would not work for our purposes. The approach of TTAnalyze is build to insert any function call, even functions with many parameters or even pointers to larger memory areas. All these data have to be placed on the stack, therefore it is not possible to write these data on the stack without ever reaching non-mapped memory. Therefore, these data are written on the stack with code inserted into the virtual system.

For functions like `WaitForSingleObject()` we only need a small amount of memory (12 bytes for the parameters and 4 byte for the return address), therefore we can insert a function by writing the 16 bytes directly into the virtual memory and switching the stack pointer 16 bytes up[11].

I/O Completion simulation

For the simulation and replay of synchronisation I/O Completion Ports have to be considered. For this, there are two different approaches:

- Simulation of all function calls relating to I/O Completion ports, which includes `NtCreateIoCompletion()`, `NtSetIoCompletion()`, `NtRemoveIoCompletion()`, `Nt-SetInformationFile()` etc. One problem with this approach would be that we do not know from the beginning which I/O Completion ports are used for the network purposes and therefore have to be simulated (the ports are associated after the creation with an AFD device, therefore at creation time it is impossible to decide if a `NtCreateIoCompletion()` has to be simulated or not). Furthermore, one I/O Completion Port is sometimes used for several purposes (i.e., one port may be associated with several devices and additionally it may be filled manually with items via `NtSetIoCompletion()`), so it would be some effort, to simulate all these function calls.

- The creation of I/O Completion Ports is executed as usual, even the functions `NtSetInformationFile()`, `NtSetIoCompletion()`, and `NtRemoveIoCompletion()`. Only the execution of `NtDeviceIoControlFile()` using as handle a device `\Device`

[11]We did not encounter problems with these technique - it seems that Windows does not allow to get the free stack smaller than these 16 bytes

\Afd\AsyncXXXHlp needs a simulation. For these cases, a completion message is queued on the queue inside the kernel with an inserted `NtSetIoCompletion()` function call.

The second approach is the more efficient one, but it has still a problem to be solved. Especially the actions executed in an asynchronous way (e.g., connect, select) need a certain amount of time be finished. Therefore, the completion messages cannot be queued to the kernel immediately with the simulation of `NtDeviceIoControlFile()`. Thus, if the port is not only used for one purpose, the order of the queued messages it is not deterministic. The problem during the Replay Mode is to decide, when a completion message has to be queued to the port.

This problem is not easy to solve, because many different situations have to be considered. Most of these problems result from the fact, that the scheduling of thread is not deterministic which results especially at processes with several threads at Replay Mode a complete different order of execution of the I/O Operations. Consider, for example, the following scenario.

In the "new" system a function `NtRemoveIoCompletion()` is executed and in the record a function `NtRemoveIoCompletion()` returns immediately. In the "new" system the corresponding `NtDeviceIoControlFile()` has not been executed and, therefore, there is no completion message on the kernel intern queue. Thus, if the replay lets this function return with the recorded values (whereas the handles have to be replaced from "old" to "new" ones), this will cause an unwished behaviour and will pretty sure result in an error.

To avoid such situations, the theoretical status of the kernel-intern queue is simulated. Thus, all function calls of `NtDeviceIoControlFile()` for an "async AFD device" and `NtSetIoCompletion()`, which cause a message to a completion port (sometimes with a temporal delay), cause an virtual completion message to the simulated Completion Port. Therefore, at Replay and Simulation Mode it is known if on the kernel intern queue a message is available.

Consider the example described above: By looking at the simulated queue, `NtRemoveIoCompletion()` can return only if a completion message is available which is only the case, if the corresponding `NtDeviceIoControlFile()` has already been executed. Obviously, if a `NtRemoveIoCompletion()` is simulated, the corresponding message has to be removed from the simulated queue.

7.2.4 Simulation Mode

In Simulation Mode the complete interaction is simulated, therefore the responses must be inserted manually (e.g., if the test-subject is executing an `AFD_RECV` the replay has to provide some data for the test-subject). The class `CManReplay` provides a set of functions, which are queried for the corresponding informations. In the following, the functions for a TCP receive are discussed.

```
Bit32u tcpReceive(Bit32u Handle, Bit32u eventHandle,
                             Bit8u *buff, Bit32u size);
```

```
void letTcpReceiveReturn(Bit32u Handle, Bit32u eventHandle,
                         Bit8u *buff, Bit32u size);
```

Listing 7.6: Receive functions of class `CManReplay`

In Listing 7.6 the function declarations of the callback functions for a TCP receive are shown. If the `CAfdDecoder` receives in Simulation Mode a function call of `NtDeviceIoControlFile()` with IoControlCode `AFD_RECV`, the function `tcpReceive()` of the class `CManReplay` is called. The parameters `buff` and `size` describe the buffer the client is able to receive (the size is equivalent to the buffer, the client allocated for a `recv()` of Winsock). Therefore, this buffer has to be filled manually. The return value defines the `information` field of the IoStatusBlock and therefore the size of the response. Obviously, the function result has to be a value smaller or equal than the `size` parameter. In this case, the function is simulated as returning immediately.

To simulate a function that blocks, `tcpReceive()` has to return `0xFFFFFFFF`. In this case, the `CAfdDecoder` inserts a `ZwWaitForSingleObject()` function, which causes the thread to block. Obviously, the question arises how this thread can be released so that the function returns? This is what the function `letTcpReceiveReturn()` and the parameters `Handle` and `eventHandle` from `tcpReceive()` are for. `letTcpReceiveReturn()` (which calls the corresponding function of the `CAfdDecoder` class) receives the `Handle` and `eventHandle` of the virtual blocking function. The corresponding function of the `CAfdDecoder` class asserts that such a function is noted as blocking and then inserts a `NtPulseEvent()` function call (to release the inserted `NtWaitForSingleObject()`). The `buff` parameter is a newly allocated buffer that is filled with the response, the `size` parameter defines the size of the buffer (again, the `size` is used to set the `information` field of the IoStatusBlock and may not larger than the size of the buffer allocated by the test-subject).

As for the Repaly Mode, for Simulation Mode the synchronisation is the most challenging task. For a complete simulation of more complex test-subjects the testing algorithm has to be advanced, to understand the dependences of the different threads.

Chapter 8

Evaluation

The first part of this chapter describes how I worked to discover all the information outlined above. The second part explains how I verified my work.

8.1 Mode of Operation

Most of the information that was required for writing this thesis is not public available. Therefore, beside the implementation of the different modes, the main challenge was to reverse engineer the mode of operation of the AFD codes and the synchronisation techniques used. This reverse engineering was an iteration of different tasks.

1. **Analyzing the advanced log files**: during the work, I enriched TTAnalyze with the ability to log all functions of all loaded DLLs. This required a more advanced mechanism to keep track of all loaded DLLs: In Windows it seems to be possible for a process to load DLLs implicitly without explicitly calling a corresponding function (e.g., `LoadLibraryExW()`). These functions are used in TTAnalyze to keep track of all loaded DLLs, but I discovered that sometimes DLLs are loaded or mapped into the address space of a process without any of the functions for loading a DLL being called. Nonetheless, such DLLs appear in the PEB (Process Environment Block) so that it is possible to find the corresponding file on the virtual hard disk and to analyse this file in order to register all exported functions. It would be very expensive for every translation block to iterate over all registered loaded DLLs to find implicitly loaded DLLs. Therefore for every loaded DLLs the executable areas are registered and for every translation block it is verified whether the current instruction pointer points to an registered memory area. This is expensive too (thus, this feature should only be used for reverse engineering), but it is more efficient than iterating over the PEB, because this would result in a lot of RPC calls for reading the virtual memory, which is necessary for iterating over the PEB.

   These advanced log files gave a good overview of which system calls and sub-DLLs are used for e.g., connecting a socket. However, for a detailed view, these log files were not suitable because it is not possible to view the memory or function parameters of all function calls. However, this method delivers hints as to which function calls are the significant and relevant ones.

2. **Analyzing the "normal" log files**, which include for functions hooked for TT-Analyze the function parameters. With the method described above, the relevant function parameter could be tracked down to the systems calls. Therefore, with the logged function parameters the usage of e.g., a handle for a socket could be watched, for which system and function calls this socket handle is used, etc. But even with this method, it is not possible to reverse engineer e.g., structures used in input buffers, etc.

3. **Debugging session**: during debugging sessions it is possible to halt the executable and search for specific function parameters, e.g., the IP a socket has to be connected to. However, there are many function calls even for only one function (e.g., `NtDeviceIoControlFile`), therefore, the breakpoints have to be conditional to receive only or mostly relevant function calls. For this, a handle, a control code or the like has to be known. This information is gathered in the preceding steps, therefore this task can be used to approve of or discard speculations about specific function calls or function parameters.

4. **Implementation**: if the first three steps were successful, the hypotheses derived from Steps 1-3 were implemented and tested with different executables. If under different test scenarios all executables behaved as expected, I could be nearly sure, that my hypotheses were correct.

Obviously, it has not been possible to go straight from Step 1 to Step 4 . There were a lot of incorrect and misleading assumptions, which could not be verified in a subsequent step. E.g., if I did not recognise a function in Step 1, I could not analyze it in Step 2. It was often the case that the assumptions were correct until Step 4, and in Step 4 I had to realize that my implementation considered only one specific scenario and several other scenarios occur with different executables, different parameters or a different behaviour caused by race conditions.

Furthermore, it was sometimes difficult to realize which problem was currently the most significant one. There are many function calls in different threads. Even the execution of the example function `foo()` from Listing 4.1 results in a log file of about 40.000 lines. The log file for starting the Internet Explorer and loading http://seclab.tuwien.ac.at produces a 140MB logfile (about 1.8 million lines). Finding the correct function(s) for a concrete problem in all logged functions is a tedious task. The synchronisation tasks were especially difficult to understand, because in contrast to pure socket functionality, it is difficult to isolate a concrete set of function calls. For AFD, if the function `send()` from Winsock is called, the system calls for implementing this functions must be executed between the call and the return of this function. For synchronisation tasks, this is not as simple because several threads have to be monitored and the function call(s) searched cannot be isolated as comfortably as for AFD calls. [1]

8.2 Verification

For the verification of the work I first tried to replay function calls implementing network access. For this, the two modi “Record Mode” and “Replay Mode” were implemented. To verify the flexibility of my work, I simulated the network for applications whose behaviour could be verified easily - browsers and self-written test-executables:

- Microsoft’s Internet Explorer was the most difficult application, because it uses for a simple site a couple of concurrency threads. Parts of these threads are used to synchronize the “real working” threads, also using sockets. This raises the number of threads and handles to map from the old to the new system and makes it more difficult to replay the function calls in correct order. With the tool-set described above the scheduling of threads cannot be influenced directly, it is only possible to influence the state of a thread (waiting or blocking), but if two threads are returning it is not possible to influence the decision of the operating system which thread is released first.

  Thus, the more threads and handles are executed, the more difficult it is to guess how the handles and threads are mapped correctly from the old to the new system. For the Internet Explorer, I am able to replay about the first 30 function calls (depending on how different the non-deterministic execution of Record and Replay Mode are) and therefore about 5 thread switches.

- Opera uses a different approach for synchronising its threads. That is, it uses the already mentioned WSAAsyncSelect Model, which uses I/O Completion Ports for synchronisation. Out of this, there are less threads executed and the mapping of threads and handles is more simple. Therefore, for Opera it is possible to replay the loading of a simple site.

- The self-written test-executable executing the function `foo()` from Listing 4.1 is single-threaded and therefore, the mapping of the handles from old to new system is quiet simple. Thus, a replay for single-threaded application has not been any problem.

The next step is to replay the corresponding function-calls without any recording, therefore to simulate the interaction between user-mode and kernel-mode completely. This is what the simulation mode is for.

The complete simulation of network traffic requires a deeper understanding of the data exchanged between user-mode and kernel-mode, the used synchronisation techniques. This

[1]Nonetheless, the work was, though sometimes annoying, mostly great hacking fun.

knowledge is needed by the human analyst, who tries to simulate the network traffic. Furthermore, a deeper understanding of the test-executable is required (i.e., it is essential to know when a particular thread has to return, which data are filled into the response buffers, etc.). This is particularly difficult if e.g., sockets are used as synchronisation technique (such as in Microsoft's Internet Explorer).

Obviously, this is exactly what was described in the introduction and what the goal of this theses was. Therefore, as I am able to simulate the network access for the code from Listing 4.1, I have been able to reach my objectives.

Chapter 9

Future Work and Conclusion

9.1 Future Work

Networking under Windows is a wide-ranging topic, there are some APIs TTAnalyze is currently not able to handle. Beside sockets, Windows provides other possibilities to access network resources [15] that are not accessed via AFD:

- Named pipes and mailslots *provide reliable bidirectional and unreliable unidirectional data transmission* [15], e.g., for network drives.
- Remote procedure call (RPC) are an independent API, but are executed over named pipes, LPC and AFD.

Furthermore, for synchronisation tasks, methods will have to be found to simulate them in a more comfortable way, without understanding the current technique for every test subject. The currently used system, which tries to map handles and threads from the old to the new systems, does not work with more complex scenarios. One possible solution could be to write a more general and more intelligent analyzing part, which is able to understand different synchronisation techniques and help the human analyzer using the Simulation Mode to avoid inconsistent system states. Consider a local socket that is used for synchronisation: Such a more intelligent analyzing module has to understand that two sockets are connected to one another and receiving in one thread can only return bytes if another thread has been sent bytes over this socket.

Obviously, there are a lot of techniques, which have to be understood. Therefore, it will take a lot of research, reverse engineering and coding work until software will be is able to understand all these techniques.

9.2 Conclusion

At the beginning of my project, I expected that the most difficult part would be to understand the undocumented AFD protocol. During my work I realised that that was a relative easy task provided I had sufficient knowledge of function call conventions and assembler code, some reverse engineering practice, and a set of good tools. For AFD, it is possible to reduce the set of functions which have to be examined to a relatively small number. After I had understood the general behaviour and usage of AFD, the research of further AFD codes was similar to that of the preceding AFD codes.

For synchronisation this is not the case. The usage of events is relatively simple, easy to understand, documented and more like the schemes of every day programming. This may result from the fact that events occur in every programmer's life and are used day by day.

However, for "normal" programmers the usage of I/O Completion Ports is something completely new. Certainly, the technique behind I/O Completion Ports was not invented by Microsoft for the Windows Kernel, nonetheless, it is not a basic concept for every programmer. Furthermore, the creation, setup and usage of these ports do not occur in chronological order. In particular, these ports are sometimes used without this being apparent (e.g., when a completion message is queued inside the kernel to a completion port by a call of `NtDeviceIoControlFile()`) and, therefore, have been difficult to detect and understand.

Finally, I showed that it is possible to omit the kernel for network access with sockets and therefore that it is possible to build tools that carry out a more sophisticated dynamic analysis by running through different scenarios of the execution of a test subject.

Bibliography

[1] Sven B. Schreiber. *Undocumented Windows 2000 Secrets.* 2001.

[2] Ed Skoudis. *Counter Hack.* 2002.

[3] H. D. Moore. Month of browser bugs. `http://browserfun.blogspot.com/`.

[4] Eric Bangeman. Number of browser vulnerabilities rising. `http://arstechnica.com/news.ars/post/20060925-7818.html`, Sept. 2006.

[5] Merijn Bellekom. HijackThis. `http://www.merijn.org/programs.php`. A freeware spyware-removal tool for Microsoft Windows.

[6] Peter Szor. *The Art of Computer Virus Research and Defence.* symantec press, 2005.

[7] Fred Cohen. Computer viruses: Theory and experiments. *Computers & Security*, 6:22–35, 1987.

[8] Greg Hoglund and James Butler. *Rootkits - Subverting the Windows Kernel.* Addison-Wesley, 2006.

[9] Ulrich Bayer, Christopher Kruegel, and Engin Kirda. TTAnalyze: A tool for analyzing malware. *15th European Institute for Computer Antivirus Research (EICAR 2006) Annual Conference*, 2006.

[10] Microsoft PECOFF. `http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx`, 2006.

[11] IDA Pro. `http://www.datarescue.com/`. IDA Pro is a Windows or Linux hosted multi-processor disassembler and debugger.

[12] Gary Nebbett. *Windows NT/2000 Native API Reference.* 2000.

[13] Fabrice Bellard. Qemu, a Fast and Portable Dynamic Translator. *Usenix annual technical conference*, 2005.

[14] *Intel 64 and IA-32 Architectures Software Developers Manual Volume 3A: System Programming Guide, Part 1.* Intel, 2006.

[15] David Solomon Mark Russinovich. *Microsoft Windows Interals*. Microsoft, 2005.

[16] Prasad Dabak, Milind Borate, and Sandeep Phadke. *Undocumented Windows NT*. 1999.

[17] System Services for Windows. http://msdn2.microsoft.com/en-us/library/aa830632.aspx.

[18] Debugging Tools for Windows. http://www.microsoft.com/whdc/devtools/debugging/default.mspx.

[19] RFC 791, Internet Protocol. 1981.

[20] RFC 2460 Internet Protocol, Version 6 (IPv6). 1998.

[21] University of Southern California Information Sciences Institute. RFC 793, Transmission Control Protocol. 1981.

[22] J. Postel. RFC 768, User Datagram Protocol. 1980.

[23] J. Postel. RFC 792, Internet Control Message Protocol. 1981.

[24] P. Mockapetris. RFC 1034, Domain Names - Concepts and Facilities. 1987.

[25] ISO/TC97/SC 16 Reference model of open systems interconnection. 1979.

[26] Windows Sockets 2. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winsock/winsock/windows_sockets_start_page_2.asp, 2006.

[27] Winsock functions. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winsock/winsock/winsock_functions.asp, 2006.

[28] Transport Driver Interface. http://msdn.microsoft.com/library/en-us/NetXP_d/hh/NetXp_d/303tdi_af260005-f147-404f-8883-d4b6328b09fe.xml.asp?frame=true, 2006.

[29] Anthony Jones and Amol Deshpande. Windows Sockets 2.0: Write Scalable Winsock Apps Using Completion Ports. *MSDN Magazine*, 2000.

[30] Anthony Jones and Jim Ohlund. *Network Programming for Microsoft Windows*. Microsoft Press, 1999.

[31] Protocol-Independent Out-of-Band Data. http://msdn2.microsoft.com/en-gb/library/ms740102.aspx, 2006.

[32] Inside I/O Completion Ports. http://www.microsoft.com/technet/sysinternals/information/IoCompletionPorts.mspx, 2006.

[33] Terence Parr. Antlr. http://www.antlr.org/.

[34] Reactos. `http://www.reactos.org`. ReactOS is an advanced free open source operating system providing a ground-up implementation of a Microsoft Windows XP compatible operating system.

[35] Ethereal. `http://http://www.ethereal.com/`. Network sniffer and protocol analyzer.