TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

Master's Thesis

TTAnalyze: A Tool for Analyzing Malware

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

under the guidance of

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

by

Ulrich Bayer

Kirchberggasse 37/6
1070 Wien

Vienna, December 12, 2005

Kurzfassung

Diese Diplomarbeit beschreibt TTAnalyze - ein Tool, um das Verhalten von ausführbaren Windows Dateien (PE-Dateien) und im Speziellen von Malware zu analysieren. Das Ergebnis eines Programmaufrufs von TTAnalyze besteht in einer Report-Datei, die es einem menschlichen Benutzer ermöglicht, den Zweck einer ausführbaren Datei einzuschätzen. Der generierte Report wird wahlweise als HTML-Datei oder als einfache Text-Datei ausgegeben. Er enthält detaillierte Information über die vom Testsubjekt gemachten Veränderungen an der Windows Registry oder am Dateisystem, sowie über Interaktionen mit dem Windows Service Manager oder anderen Prozessen. Der gesamte Netzwerkverkehr wird ebenfalls mitgeloggt.

Das Herzstück der Analyse besteht aus einem PC-Emulator, in dem das Testsubjekt gefahrlos und vollständig überwacht ablaufen kann. Die Analyse der Programm-Ausführung beschränkt sich dabei ausschließlich auf sicherheits-relevante Aspekte, wodurch der Analyse-Prozess vereinfacht und die Ergebnisse genauer werden. TTAnalyze ist das ideale Hilfsmittel für Personen, die Viren und Malware untersuchen, um möglichst schnell möglicht viel über eine unbekannte ausführbare Datei herauszufinden.

Abstract

This thesis describes TTAnalyze: a tool for analyzing the behavior of Windows PE-executables with special focus on the analysis of malware. Execution of TTAnalyze results in the generation of a report file that contains enough information to give a human user a very good impression about the purpose and the actions of the analyzed binary. The generated report includes detailed data about modifications made to the Windows registry or the file system, about interactions with the Windows Service Manager or other processes and of course it logs all generated network traffic.

The analysis is based on running the binary in an emulated environment and watching i.e. analyzing its execution. The analysis focuses on the security-relevant aspects of a program's actions, which makes the analysis process easier and because the domain is more fine-grained it allows for more precise results. It is the ideal tool for the malware and virus interested person to get a quick understanding of the purpose of an unknown binary.

Acknowledgments

This master's thesis was carried out in cooperation with Ikarus-Software GmbH. I want to thank them for the possibility to work on a very interesting subject that is also of practical relevance.

Moreover, I want to thank my advisors Engin Kirda, and Christopher Krügel for their patience and their support during the whole work.

Special thanks go to my colleagues at Ikarus, Martin Stöfler and Jürgen Wohlmuth. Jürgen has helped implement parts of the analysis-framework. I am grateful to Martin for keeping me focused on the important things in life.

I would also like to thank Helmut Petritsch (who created the build-system for TTAnalyze) for the world's best and coolest build-system.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

If you read a recent newspaper, chances are high that you will find an article about a worm outbreak or new virus being in the wild. Even people without any special interest in computers are aware of computer virus names such as Nimda or Sasser. Computer viruses and more generally speaking malicious code has become a problem for everyday-computer-users.

The root of all evil is that users are "double-clicking" (i.e. executing) unknown binaries. They do not know what the binary does and have no reliable way of finding out what it does. The security-aware user is going to use a virus scanner before executing the file and that is practically all the user can do. Although modern virus scanners all possess extensive heuristic capabilities, the backbone of malware detection is still a simple pattern matching mechanism i.e. comparing the file to a database of known malware signatures. The disadvantages and advantages of this approach are well-known and have extensively been discussed in literature. For example, this approach will not work for hand-crafted malware targeted at a single company that is not circulated widely and never reaches the labs of an anti-virus company.

One way to combat malware is based on the principle of trust. Trust can be realized by the means of digital signatures. If the user trusts an organization or company and the user can determine that a specific executable has been created by this company, then he may decide to trust the executable and run it. In case his trust was unjustified he still has the advantage of knowing who is responsible for this executable and thus he could theoretically take further legal steps. It is obvious, though that this is not a perfect solution. It is a work-around.

What we ultimately desire are better ways, and tools to find out what an executable does. If such tools could be made easy-to-use, they might even be an option for the casual user.

Note that not only home users, but also professional users are in need of tools for analyzing executables. Just consider the following example: A new virus sample shows up and finds its way to the virus lab. The people working there have not much time for analyzing it because of time constraints. Hence, they need tools such as TTAnalyze in order to quickly gain an understanding of the unknown executable. If the results are not

satisfying they can still start up their debuggers.

1.1 Goals

Quite enthusiastically, we have set ourselves the following goal: Let us create a (command line) tool that expects an executable as input and generates a report that describes what this executable does. So we defined the following more concrete list of requirements:

- The analysis shall work for a broad range of executables, including exe-packed executables, polymorphic and metamorphic viruses.
- The analysis should focus on user-mode applications because most existing malware programs runs as normal user-mode applications.
- We target Windows executables [1] since Windows is the prevalent operating system and most malware is written for Windows.
- We do not have a hard time limit for the duration of an analysis run. For our purposes it is sufficient if the run takes a couple of minutes.
- We wish to create a tool that allows a human user to quickly get a basic understanding of the purpose and behavior of an unknown executable.

1.2 Challenges of the Field

In this section we briefly describe the problems and the challenges that one faces when one designs and creates a tool such as TTAnalyze.

1.2.1 Creating High Level Information

Finding out what an executable does is **not** the real problem when analyzing executables. The actions that a processor executes on behalf of a program are completely defined in the executable. [2] The problem rather is that machine code is not easy to understand. Even if one translates the machine code into assembler memnomics, which is a lot more human readable, one is still faced with a great number of instructions that each says very little about the program behavior. Thus, it is necessary to decrease the number of information units that a human user has to look at and increase their information content. High-level

[1] known as PE (Portable Executable) files

[2] This is not completely true. Apart from the processor specification and the executable itself a description of the operating system is necessary to completely define the behavior of a program. Also an executable does not only consist of instructions that are to be executed by the processor but it also contains resources like graphics and buttons, data like strings constants and meta-information that describes how this executable should be treated by the operating system. Moreover, much code is normally contained in libraries and not in the executable itself.

information has to be created and reengineered from the low level assembler instructions. Thus, the goal of this master's thesis is to reduce the size of the information, and to increase the level of information. We aim to generate a report that is precise and short, but still contains all security-relevant information.

1.2.2 Static Analysis vs. Dynamic Analysis

Static analysis of binaries is the process of analyzing an executable by opening the file, reading it, maybe disassembling it, maybe generating control or data flow graphs etc. and drawing conclusions by these means. In contrast, we call dynamic analysis the process of analyzing an executable by running the program and watching its execution in order to draw conclusions. Both approaches have their strengths and weaknesses and both are used in real world programs.

A general purpose machine that is based on the Turing machine can even execute programs that create some of their instructions at runtime. In other words, the processor executes instructions that were only created by the execution of a couple of prior instructions and hence are not part of the executable file that is stored in the file system. These programs are known as *self modifying programs* in literature. This is one reason why the static analysis of an executable is limited in its usefulness.

Static analysis has the advantage that it sees the complete program code and is usually faster than its dynamic counterpart. Its main weakness is that self modifying programs and exe-packed executables are out of its scope. Dynamic analysis only sees the execution trace of a program but it is immune to obfuscation attempts and has no problems with self modifying programs.

1.3 Our Solution

Because of the already discussed problems of static analysis, we decided to implement a tool based on dynamic analysis. This way, we can guarantee that the program also works for exe-packed executables. In the face of malware though, some additional problems appear. Clearly, executing malware is a dangerous and bad idea. That is why we only emulate the execution of the program. We do not execute the unknown binary - we execute the program on a virtual processor.

This is a border line case between static and dynamic analysis. Technically speaking, we do not run the program so it has to be classified as static analysis. On the other hand, we emulate the execution of the program in software and so our analysis shares all the advantages and disadvantages of dynamic analysis. For this reason we speak of dynamic analysis when we describe how TTAnalyze works.

Emulating malware instead of directly executing it is not a new idea. In fact every modern virus scanner can emulate Windows executables. This allows for good heuristics and moreover is necessary for the detection of some otherwise hard to detect viruses.

Of course, using an emulator slows down the analysis process. However, it has one more advantage: We know at every moment what is going on in the (virtual) processor, we know the values of all CPU registers, we can read the memory - in short we have total control over everything. This is an important advantage and it also gives us the possibility to change the execution flow of the analyzed program. We will discuss later why we would wish to do that. We have to mention one disadvantage of emulation tough: The emulation may contain errors and may differ from a real execution of the program and thus introduce errors into the analysis process. Fortunately, open-source emulators exist and we did not have to write our own one. TTAnalyze uses QEMU, an emulator written by Fabrice Bellard, as its emulator component. [5]

The question arises what should be emulated. Shall we emulate the hardware only so that an off-the-shelf operating system may be installed or shall we emulate a CPU and an operating system. Virus scanners typically emulate the CPU and (parts of the) operating system.

A distinguishing feature of TTAnalyze is the fact that it is emulating an entire computer system and running an off-the-shelf Windows XP on it. As a consequence the accuracy of the emulation is excellent. Emulating an entire computer system as well as running a normal operating system and running the binary that should be analyzed is admittedly a lot slower than the sort of emulation that virus scanners do, but this is only a fair trade-off. We trade execution time against emulation accuracy because speed is not as important for an analysis tool as it is for a virus scanner.

The analysis itself is based on watching the execution of the program in this emulated system. Primarily we keep track of the Windows API functions that are called by the test subject. For a couple of reasons that we will explain in detail later, simply watching the execution of the program is not enough for drawing significant conclusions. That is why we also alter the execution of the test subject in important, but otherwise minor and unobtrusive ways.

1.4 Structure of This Thesis

By now, the reader has an overview of TTAnalyze and has a basic understanding how it works. In the next section, we define some of the terms that are going to be used consistently throughout the thesis.

Chapter 2 covers emulation. It begins with a classification of different emulation techniques, gives several examples for them, and explains QEMU in detail.

In Chapter 3, we will describe similar software tools and related work. Of course we focus on how they differ from TTAnalyze, what their advantages are and what their disadvantages are.

Chapter 4 details the implementation of TTAnalyze. It describes all components of TTAnalyze and documents the design decisions that we have made.

Chapter 5 contains several case studies that show the results of running TTAnalyze on known malware samples. We then compare the results to virus descriptions of well-known

anti-virus companies.

We finish this master's thesis with Chapter 6 where apart from recapitulating we outline possible future research possibilities.

1.5 Terminology

As in all scientific disciplines the terminology used to describe phenomena might not always be clear and its meaning might not always be standardized. In this section, important technical terms that are used throughout this thesis are described.

Virus There are many definitions of the term virus. In this paper, the classic definition as given by Frederick Cohen is used.

> "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." [8]

In some publications, you can read "virii" for the plural form of virus, but we use "viruses" in this paper, which is the correct form according to [11].

Polymorphic Virus A *polymorphic virus* uses multiple techniques to prevent signature matching. First, the virus code is encrypted, and only a small in-clear routine is designed to decrypt the code before running the virus. When the polymorphic virus replicates itself by infecting another program, it encrypts the virus body with a newly-generated key and it changes the decryption engine by generating new code for it. To obfuscate the decryption routine, several transformations are applied to it. These include NOP-insertion and instruction reordering.

Metamorphic Virus A *metamorphic virus* uses even more complex obfuscation techniques than a normal polymorphic virus. One classic and often used technique to avoid detection is the so-called "entry point obfuscation".

Binary The term *binary* is sometimes used as an alias for executable.

Test-subject We use the term *test-subject* for binaries that are to be analyzed by TTAnalyze.

Malware "A common name for all kinds of unwanted software such as viruses, worms, trojans and jokes." [13]

Trojan Horse "A *Trojan Horse* portrays itself as something other than what it is at the point of execution. While it may advertise its activity after launching, this information is not apparent to the user beforehand. A Trojan Horse neither replicates nor copies itself, but causes damage or compromises the security of the computer. A Trojan Horse must be sent by someone or carried by another program and may arrive

in the form of a joke program or software of some sort. The malicious functionality of a Trojan Horse may be anything undesirable for a computer user, including data destruction or compromising a system by providing a means for another computer to gain access, thus bypassing normal access controls." [40]

Exe-Packer A utility which compresses an executable file in a way that is transparent to the user. When the packed executable is started, it automatically expands the original file in memory and transfers control to it. Malware authors use packers for obfuscating their work and making it harder to detect by virus scanners. One of the most popular packers is UPX [42]. *Executable packers* differ from traditional compressing utilities, which work for all kinds of files and usually require a special program to unpack the compressed file.

Chapter 2

Emulation

Before we look at the implementation of TTAnalyze, we define the term *emulation* and distinguish it from related, but different concepts. Emulation is a very ambiguous term and over the time[1], it has been used differently by different people.

Emulation is typically defined as "a process whereby one computer is set up to permit the execution of programs written for another computer. This is done with...hardware features...and software..." [17].

This master's thesis concentrates on software emulators only (i.e., emulation with software features) and even if we just say emulator, we mean a software emulator. What is crucial about the definition is that the environment that a foreign program expects is emulated (in the sense of imitated) in order to run the foreign program.

A PC emulator is a piece of software that emulates a personal computer(PC), including its processor, graphic card, hard disk, etc. with the purpose of running an unmodified operating system. Note that emulation does not normally not aim to be perfect. It is not necessary to imitate all the internal details of a real CPU (e.g., instruction cache). It is enough to imitate everything of a CPU that is visible to the outside world (e.g., visible to the operating system). In other words, the goal of a PC emulator usually is to run an unmodified operating system and not to provide an exact simulation of a CPU.

It is important to differentiate emulators from virtualizers. Like PC emulators, virtualizers can run an unmodified operating system, but they execute a statistically dominant subset of the instructions directly on the real CPU. This is in contrast to emulators, which simulate all instructions in software. Virtualizers such as VMWare [44], PearPC [29] and VirtualPC [23] are popular as they are fast. Virtualizers are said to "virtualize" the hardware they are running on (allowing the hardware to be shared between several running operating systems). Thus, they cannot run foreign programs, which were compiled for a different instruction set. An emulator, on the other hand, can also imitate hardware (CPU) that is not present on the host computer and thus, can run programs that were written for a different instruction set.

[1]The first appearance of the word emulator in a scientific article that we were able to find dates back to 1969 [17].

In this chapter, we often speak of host platform and target platform. The *host platform* is the platform where the emulator itself is executed on, while the *target platform* is the platform that is emulated by the emulator. In the case of TTAnalyze, the host platform and target platform are identical. Both are Windows x86[2].

2.1 Classification of Emulators

Several classifications of emulators are possible. We classify emulators according to the object that they emulate. Figure 2.1 gives an overview of different emulators.

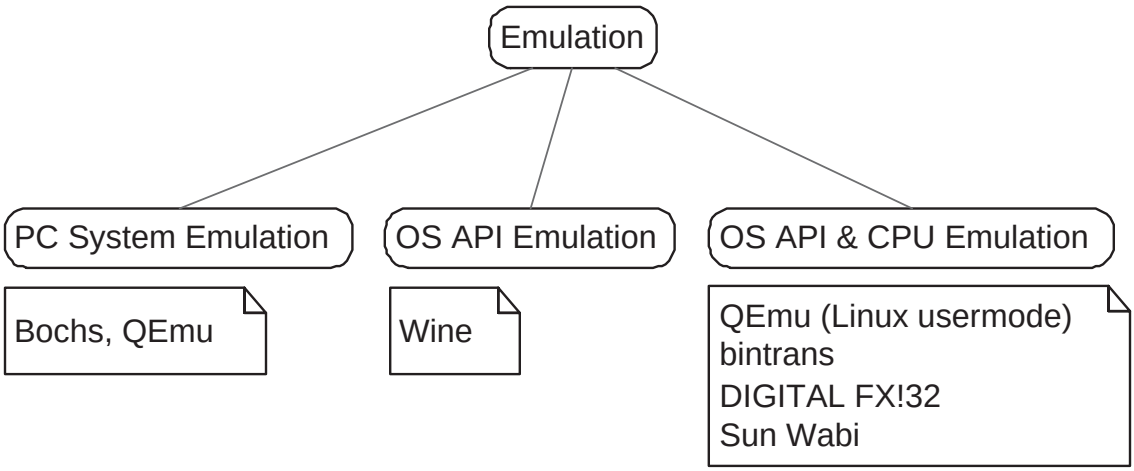


Figure 2.1: Overview of different emulators

Bochs [12] and QEMU [5] are both examples of PC emulators. They can run modern operating systems such as Windows XP and Linux. A PC emulator has to provide an implementation of the complete software-hardware interface as it is expected by an operating system. The software-hardware interface, the boundary between hardware and software, consists of everything that software sees from the hardware. In particular, it consists of the CPU instruction set. The resources exposed by hardware devices are accessed by a subset of the CPU instruction set. Hardware devices thus are visible to software via special CPU instructions. The BIOS[3] plays an important role: It is a memory area which contains code that has been programmed into a hardware component(the BIOS chip). Among other things, the BIOS controls the PC boot process.

Another interesting approach is used by the Wine [46] program. Wine is a Linux program that enables the execution of Windows user-mode programs under Linux. It works by emulating a Windows environment for the program to be executed. In particular, it provides implementations for most of the Windows API functions. The Windows API represents the services that all Windows operating systems provide for (user-mode) programs.

[2]We use x86 as a generic term for IA-32 CPUs, AMD's x86-64 and Intel's EM64T CPUs.
[3]Basic Input Output System.

It does not include a CPU emulation, that is why the emulation works only for Windows x86 programs on Linux x86 platforms. The advantage of this approach is performance. Since the instructions of the program are directly executed on the real CPU, there is no performance loss. Although the name[4] suggests that Wine is not an emulator, it is an emulator. The reason is that a Windows environment is fully simulated in software. This shows that our definition of emulation goes beyond emulation of CPU instructions only. The implementors of Wine probably aimed to clarify that Wine does not contain a CPU emulator and does not emulate the CPU instructions which - as pointed above - makes Wine a lot faster.

A third approach is to emulate both OS API and CPU. This is the logical solution to the need of running user-mode programs on the local computer that were written for a different OS and for a different processor, respectively. Those systems implement the OS API like Wine does and additionally emulate all CPU instructions. The disadvantage is that the CPU emulation takes time and makes the overall emulation much slower.

2.2 Virtualization

Virtualization is an established research topic and a technology that has already been in use for over 40 years in the area of mainframe computers. Although it was very popular in the 1960's and 1970's, the scientific community and the industry lost interest in it during the following twenty years. Only recently, virtualization has made a comeback, and it has also become an interesting topic for PC users. Both AMD and Intel have announced hardware virtualization support for their future CPUs. Intel calls their virtualization support Vanderpool, AMD has named it Pacifica. In the following paragraphs, we will explain virtualization, why its becoming interesting again, why hardware virtualization support is reasonable, and how it impacts current virtualization software such as VMWare.

Virtualization is commonly understood as the process of creating one or several virtual machines where "a virtual machine is taken to be an efficient, isolated duplicate of the real machine" [31]. Originally, a virtual machine was an operating system level concept - nowadays, the term virtual machine is not as clear anymore. For example, there is the Java virtual machine, which is the duplicate of a non-existing machine. Taken literally, a virtual machine is simply a machine that does not really exist and so people even use it for PC emulators. That is why we are going to avoid the term virtual machine in the future and instead speak of virtualization. For this section the term virtualization is used as defined before.

The piece of software that creates virtual machines is called virtual machine monitor (VMM) or virtualizer. See Figure 2.2.

According to [31] a VMM has three essential characteristics:

1. The VMM provides an environment for programs that is essentially identical with the original machine.

[4]Wine is a recursive acronym for "Wine is not an emulator".

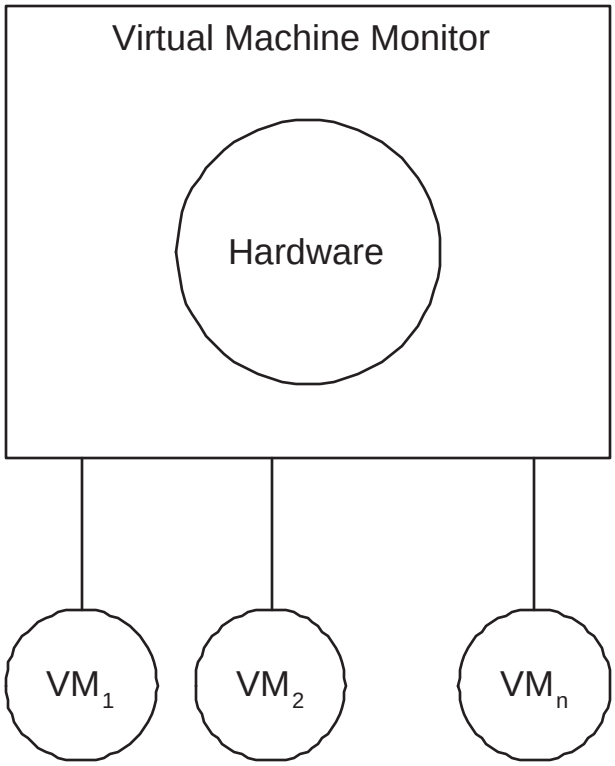


Figure 2.2: Virtual machine monitor

2. Programs run in this environment show at worst only minor decreases in speed.

3. The VMM is in complete control of system resources.

Any program that runs on the original machine runs on the virtual machine in exactly the same way because the created environment is essentially identical. The qualification "essentially identical" is necessary to rule out differences caused by the availability of system resources and timing dependent code.

Characteristic 2, efficiency, demands that a statistically dominant subset of the virtual processor's instructions be executed directly by the real processor. This is the decisive point in differencing VMMs from emulators.

2.2.1 Classification of Virtualizers

Virtualizers are often classified according to the following two characteristics.

Virtualization vs. Paravirtualization Virtualization is the normal case. It creates virtual machines that are "efficient, isolated duplicates of the real machine". Paravirtualization is a different technique that changes the hardware-software interface a bit in order to achieve greater speed (especially on the widespread x86 architecture full virtualization is

complicated and requires speed losses). The downside is that unmodified operating systems do not run anymore. They have to be modified in order to run under a paravirtualizer.

An example of a paravirtualizer is Xen [4]. Future versions of Xen will take advantage of Intel's and AMD's hardware virtualiziation support though, which will allow Xen to run umodified operating systems and at the same time make Xen a full virtualizer. VMWare, VirtualPC and PearPC are all full virtualizers.

Type I VMM vs. Type II VMM A Type I VMM runs directly on the machine hardware, while a Type II VMM runs as an application on a host operating system. VMWare, VirtualPC, and PearPC are Type II VMM's. Xen is a Type I VMM.

2.2.2 VMM Advantages

Mendel Rosenblum, VMWare chief scientist, names the following advantages of a VMM [34].

Isolation A VMM can create several virtual machines, which all run on the same computer. Although they run on the same computer, they are isolated from each other, and an operating system running in one virtual machine does not notice the presence of other virtual machines. For an OS, the virtual machine is indistinguishable from the real machine.

This form of isolation is far superior to the level of isolation that is granted to a regular operating system process.

Low overhead/high performance The overhead caused by a VMM is measured in a small percentage because the virtual hardware is in most cases directly mapped to the real hardware. The virtual processor for most instructions maps to the real processor, thus allowing it to be directly executed. In the same way virtual I/O is mapped to the real I/0 system.

2.2.3 Hardware Virtualization Support

The principle of VMMs demands that most instructions are executed directly. Only a couple of sensitive instructions that change the overall system state or I/O instructions have to be processed by the VMM in order to virtualize the hardware and give every virtual machine the idea that it is the real machine. Usually, VMM's implementations require that the CPU traps on sensitive instructions, i.e. returns control to the VMM when sensitive instructions are encountered. In [31] Robert P. Goldberg gives a more formal model of a virtual machine and describes the formal requirements for machines to support VMMs.

As pointed out in [33], the Intel Pentium processor family has many features that support the implementation of a VMM. However, there are a couple of non-trapping sensitive instructions. Slight modifications to the processor would significantly facilitate the

development of VMMs. The introduction of hardware virtualization support by Intel and AMD is going to address exactly this point, making VMM's easier to develop and faster.

2.2.4 Relationship between VMMs and Emulators

Although we have treated VMMs as something completely different than emulators, they have a great deal in common. Efrem G. Mallach points out that even the question of instruction set is becoming less valid as a point of distinction between the two concepts [18].

Consider the virtual I/O system as one similarity. A virtual machine cannot send input/output instructions directly to the hardware, as the hardware has to be potentially shared between several virtual machines. Emulators can neither send input/output instructions directly to the hardware because they have to simulate input/output instructions in software like every other instruction. "They are both faced with programs that behave as if they had full control over the hardware... They must both, in many cases, map operations onto devices having meaningful physical differences from the virtual device: for example, a disk file might be used instead of a tape unit, a remote terminal instead of a line printer, and so on." [18]

These similarities are furthermore proved by the fact that Xen uses IO device simulations that are part of the QEMU source code.

QEMU itself is an interesting program. It can be run in different modes. One mode, called kqemu, is intended for running x86 target programs on x86 hosts only. It directly executes all user-mode instructions, thereby significantly increasing the speed of the emulation. Note however that the QEMU module that is used together with TTAnalyze does not use this mode. Although it would be faster, it is against our principle of having complete control over the processor and the emulated system.

2.3 CPU Emulation

CPU emulation is the process of emulating the behavior of a CPU in software. A CPU emulator is the core of every PC emulator. The state of the virtual CPU is normally represented by a struct that, among other things, contains the registers of the virtual CPU. As an example consider the following parts of the CPU state structure definition that is used by QEMU for emulating x86 CPUs.

```
(1) #ifdef TARGET_X86_64
(2) #define CPU_NB_REGS 16
(3) #else
(4) #define CPU_NB_REGS 8
(5) #endif
(6)
(7) typedef struct CPUX86State {
(8)
(9)     /* standard registers */
```

```
(10)     target_ulong regs[CPU_NB_REGS];
(11)     target_ulong eip;
(12)     target_ulong eflags;
```

Line 10 shows the general-purpose registers of the emulated CPU. QEMU can emulate 32 bit Intel CPUs, as well as 64 bit CPUs. In case of a 64 bit CPU, the macro TARGET_X86_64 is defined, and the number of general-purpose registers is 16 instead of 8. The data type "target_ulong" has a size of 4 bytes for 32 bit CPUs, and a size of 8 bytes for 64 bit CPUs. Line 11 shows the EIP - register, which holds the instruction pointer. In Line 12 the "eflags" register is declared.

```
(13)    /* segments */
(14)    SegmentCache segs[6];
(15)    SegmentCache ldt;
(16)    SegmentCache tr;
(17)    SegmentCache gdt;
(18)    SegmentCache idt;
(19)
(20)    target_ulong cr[5];
(21)
(22)
(23)    /* FPU state */
(24)    unsigned int fpstt; /* top of stack index */
(25)    unsigned int fpus;
(26)    unsigned int fpuc;
(27)    uint8_t fptags[8];   /* 0 = valid, 1 = empty */
(28)    union {
(29)#ifdef USE_X86LDOUBLE
(30)        CPU86_LDouble d __attribute__((aligned(16)));
(31)#else
(32)        CPU86_LDouble d;
(33)#endif
(34)        MMXReg mmx;
(35)    } fpregs[8];
```

In the above code excerpt, additional Intel CPU - registers are declared. Namely, in line 14, the six segment registers (CS, DS, SS, ES, FS, and GS) are declared as an array of six SegmentCache's. The data type "SegmentCache" is not detailed here. Line 16 shows the task register, which "holds the 16-bit segment selector, 32-bit base address, 16-bit segment limit, and descriptor attributes for the TSS (Task State Structure) of the current task." [15] Line 15, 17 and line 18 show the Global Descriptor Table Register, the Local Descriptor Table Register, and the Interrupt Descriptor Table Register, respectively. Line 20 defines the five CPU control registers (CR0, CR1, CR2, CR3, and CR4). The Intel

CPU control registers determine the operating mode of the processor. Lines 24 - 35 define the FPU registers.

```
(36)    XMMReg xmm_regs[CPU_NB_REGS];
(37)
(38)    target_ulong breakpoints[MAX_BREAKPOINTS];
(39)    int nb_breakpoints;
(40)    int singlestep_enabled;
```

In line 36, additional registers required for emulation of XMM are defined. Line 38 contains support for setting hardware breakpoints on the virtual CPU. Line 40 determines whether hardware singlestepping is enabled for the virtual CPU. If singlestepping is enabled, the CPU traps after each instruction.

```
(41)    /* processor features (e.g. for CPUID insn) */
(42)    uint32_t cpuid_level;
(43)    uint32_t cpuid_vendor1;
(44)    uint32_t cpuid_vendor2;
(45)    uint32_t cpuid_vendor3;
(46)    uint32_t cpuid_version;
(47)    uint32_t cpuid_features;
(48)    uint32_t cpuid_ext_features;
(49)    uint32_t cpuid_xlevel;
(50)    uint32_t cpuid_model[12];
(51)    uint32_t cpuid_ext2_features;
(52)
(53)} CPUX86State;
```

The above code snippet shows the attributes that hold the processor identification information. This information can be retrieved by means of the CPUID instruction.

A CPU emulator first reads an instruction, decodes it, and finally executes it. In most cases, execution means changing the values of some elements of the CPU state struct. Consider the hexadecimal byte sequence 83 C0 05 as an instruction input for the virtual processor for example. The Intel-assembler representation for this byte sequence is ADD EAX, 5. The implementation of this operation in the virtual CPU might look like this.

```
cpu_state->regs[REG_EAX]+= 5;
```

2.3.1 Interpretation

An interpretative CPU emulator consists of a main loop where it reads, decodes and then executes the next instruction. As a simple example, look at the following pseudo-code:

```
unsigned char *code= {0x83, 0xC0, 0x0};
CPUX86State cpu_state;
```

```
void main_loop()
{
    unsigned char *ins= code;

    for (;;)
    {
      switch(*ins)
      {

        case 0x83:
            unsigned modRm= *++ins;
            cpu_state.regs[REG_EAX]+= get_operand(modRm, ++ins);
            break;

        ...
      }
    }
}
```

This naive technique is called interpretation. Its primary advantage is that it can easily be implemented and that it is portable. One of its disadvantages is speed: For every instruction executed by the virtual CPU, the dispatch mechanism alone requires an overhead of at least two jump instructions (which are among the most expensive instructions on modern architectures). One jump instruction is necessary in the beginning of the for-loop in order to jump to the body of the correct case-branch and another one is necessary to jump back to the start of the for-loop. The PC-emulator Bochs [12] uses this technique for its CPU emulator.

2.3.2 Dynamic Translation

Another popular technique used in the implementation of CPU emulators is dynamic translation. Dynamic binary translation is the process of translating code written for one instruction set architecture to code for another on the fly [32].

It is a more efficient form of emulation and allows for faster execution. The idea is to translate one piece of code, cache it, and if this code piece is needed again, use the already translated piece instead of translating it again. In case the emulated CPU has the same instruction set as the host PC, no instruction set translation is necessary. However, instructions affecting real system resources (memory, registers) have to be translated to instructions using virtual system resources. Consider an instruction like ADD EAX,5 for example. This instruction has to be translated to one that that modifies the virtual EAX register, which is stored somewhere in the main memory, instead of the real register.

Dynamic translation works in terms of basic blocks. A basic block is a sequence of one

or more instructions that ends with a "jump instruction or an instruction modifying the static CPU state in a way that cannot be deduced at translation time" [6].

Consider the following pseudo-code for a CPU-emulator using dynamic translation:

```
unsigned char *code= {0x83, 0xC0, 0x0};
CPUX86State cpu_state;

void main_loop()
{
    for (;;)
    {
     BasicBlock *b;
      b= find_translation_for(CURRENT_PROGRAM_COUNTER);

      if (b == NULL)
      {
        b= generate_host_code(CURRENT_PROGRAM_COUNTER);
      }

      execute_basic_block(b);
    }
}
```

First a translation for the current program counter (value of the EIP-register on x86 CPUs) is searched. If none is found, it is generated. Afterwards the code is executed.

Dynamic translation is faster because of several reasons:

- If a piece of code is executed more than once, the following executions of this code piece are almost as fast as its execution on a real CPU (if it can be found in the translation cache).

- The basic unit of code that is translated or executed at a time (called basic block) normally consists of more than one instruction. This greatly reduces the overall overhead for the dispatch mechanism.

- By focusing on basic blocks instead of single instructions, a number of optimizations become possible. One example are flag optimizations. In this case, we make use of the fact that it is not necessary to correctly calculate the EFLAGS register for each instruction, if the value of the EFLAGS register is not used by a later instruction. Jump optimizations. Under certain circumstances it is possible to directly patch basic blocks and make one basic block jump directly to another one, thus bypassing the dispatch mechanism.

The downside of dynamic translation is that it increases complexity. The implementation becomes harder to write, harder to read, and as a consequence probably contains more bugs. Moreover, it is normally not portable because a dynamic translator is tailored to one specific platform much like a traditional compiler backend targets one platform.

2.4 QEMU

QEMU uses dynamic translation in order to increase its emulation speed. Although no final version of QEMU exists yet - at the time of writing this document, 0.7.2 was the most recent version of QEMU - it is considerably faster than Bochs and based on the author's experience also stable.

As described in [6], QEMU uses a new approach for its dynamic translation engine, which makes it portable and still fast at the same time. Describing QEMU's implementation in detail is out of the scope of this master's thesis. However, in the following we explain a few concepts that are important for TTAnalyze.

QEMU's Portable Translation Engine

QEMU is able to emulate several computer systems and several processors. Moreover, QEMU is able to run on many different different computer systems and processors (*host processors*). For every processor that QEMU can emulate (the so-called *target processor*) QEMU has to support a translation engine that translates instructions from the target processor to instructions of the host processor. This translation is a two-step process in QEMU as shown in Figure 2.3.

Figure 2.3: QEMU's translation engine

First of all, the target instructions are converted into an intermediate representation. In the next step, this intermediate representation, called *micro operations* in QEMU's documentation, is converted to host instructions. The transformation from intermediate representation to host instruction has been implemented in a way that is independent of the host instruction set. Therefore, QEMU does not have to implement a translation engine for each combination of target and host CPU. This is why QEMU's dynamic translation engine is called portable.

In the case of TTAnalyze, QEMU runs on a PC system with an IA-32 processor and emulates a PC system with an IA-32 processor. The following example demonstrates the

translation process in such a configuration. Figure 2.4 shows the instructions that will be emulated (i.e., executed on the target CPU). In this example, the AT&T assembler syntax is used.

In Figure 2.5, one can see the intermediate representation for these input-instructions.

```
add    $0x8,%esp
jmp    *0x806ed384
```

Figure 2.4: Input to QEMU's translation engine

```
movl_T1_im 0x00000008
movl_T0_ESP
addl_T0_T1
movl_ESP_T0
update2_cc
movl_A0_im 0x806ed384
ldl_kernel_T0_A0
jmp_T0
set_cc_op 0x00000008
movl_T0_0
exit_tb
```

Figure 2.5: QEMU intermediate code

T0, T1, A0 are temporary registers. A0 is a temporary register for holding addresses. As one can see in this example, the intermediate representation contains only a limited number of instructions. The intermediate code does not support every possible combination of source and target operand locations. Usually, it only supports operands stored in temporary registers. That is why in the intermediate code, the operands of add (namely the immediate value 0x08 and the value in register ESP) are copied to temporary registers first.

`ldl_kernel_T0_A0` loads 4 bytes from the address given in A0 to the register T0. The two intermediate instruction `update2_cc` and `set_cc_op` deal with the correct computation of the EFLAGS register. They can be ignored for the moment. Translation blocks typically end with a sequence of `movl_T0_*` and `exit_tb`. The `movl_T0_*` instruction controls the chaining of translation blocks (QEMU can patch a translation block so that control directly flows from one translation block to the next one without returning to QEMU's main execution loop). `exit_tb` marks the end of a translation block and is translated to a "RET" instruction on x86 hosts.

Chapter 3

Related Work

Analyzing unknown executables is not a new problem. Consequently, several solutions already exist. We present these solutions in this chapter. Moreover, we compare them to TTAnalyze, point out the differences, advantages, and disadvantages of each solution.

3.1 Sysinternals Tools

The website *www.sysinternals.com* hosts many advanced Windows utilities, which together make a lot of internal Windows activity visible. The most known tools are the *Process Explorer* [35], which lists all running Windows processes similar to the Windows Task Manager, *Regmon* [37], which shows all registry activity, and *Filemon* [36], which shows all file activity.

Filemon and Regmon together cover two areas of the information that is provided by TTAnalyze. Of course, running malware on a normal computer is going to be disastrous. Thus, one has two alternatives to running the executable on a normal computer:

1. *Virtualized PC* - Running the executable in a virtualized PC such as one provided by VMWare is an often chosen approach. In this case, running the malware can only effect the virtual PC and not the real one. After running the malware, the infected HD-image is discarded and replaced by a clean one. VMWare is sufficiently fast, so that there is almost no difference to running the executable on the real computer. The only drawback is that the executable to be analyzed may determine that it is running in a VMWare machine (instead of a real PC) and may behave differently then. In fact, several different ways of how a program can detect if it is run inside a virtual machine have been published in the past and hence, are available for use by malware authors. For example, the online article [45] shows how to detect the presence of VMWare and Virtual PC. It is more difficult for a program to detect that it is running an emulator such as the one used by TTAnalyze(QEMU). This is one of TTAnalyze's advantages.

2. *Dedicated PC* - Having a dedicated real PC can help avoid the problem of a different

execution behavior as mentioned above. The dedicated PC that I am referring to is installed by a clean disk-image that not only contains the operating system but also possible useful tools like Filemon, Regmon, Windows Debug symbols, a debugger, etc. Of course, this process involves more work than using VMWare, but it is still useful for the cases when VMWare does not work.

Independent of how the executable is run - when it is run, one can find out its file and registry activity by having Filemon and Regmon execute in the background. Both of these tools work similar. Accessing the file system or the registry is only possible by performing a system call. These two tools include a kernel-level device driver, which overwrites the system-call table that Windows uses to look up the implementation for a requested system call. As a consequence, Regmon's or Filemon's code gets called by file and registry system calls where it is logged and afterwards forwarded to the real Windows implementation.

Regmon and Filemon show the registry and file activity of all running processes and they do so in realtime. Normally, the amount of data returned by regmon or filemon becomes large in a very short time. Filtering mechanisms are available though. Still, the filtering possibilities are rather simple and not sufficiently documented. In Filemon, for example, it is possible to narrow down the data to data tuples that reference a specific executable. In this case, all data tuples that originate from the process running the specified executable and all data tuples that operate on a path containing the specified string are shown. Hence, the filtering mechanism is a string search on *all* columns of a data tuple.

Usually, the person who analyzes an unknown binary is only interested in the files that have been created, changed, read and written to during its execution. TTAnalyze shows exactly this information. Filemon shows a lot more information and its filtering capabilities are not advanced enough to confine the output to the desired criteria. The same assessment also applies to Regmon.

Still, Regmon and Filemon are valuable tools. And it would be possible to save their output to a text-file and have this file parsed by a little script that filters unnecessary information. Moreover, Regmon and Filemon show all file and registry activity on a computer, and this can be an advantage. It often happens that the unknown executable starts another process or service. The two Sysinternals tools still work then, while TTAnalyze has to be started again for the created process. Also, TTAnalyze provides no direct support for analyzing services at all. In this sense, the Sysinternals tools offer more flexibility than TTAnalyze does.

Last, but not least, these tools have been a great help in the development of TTAnalyze. Their output made it possible to compare TTAnalyze's results to a known and verified source.

3.2 Debugger

The debugger is the traditional tool for analyzing unknown executables and malware. Typically, no source code and no debugging information is available for unknown executables,

which only leaves debugging at the assembly level. The main disadvantages of debugging are listed in the following:

- It is possible for a program to detect if it is run under a debugger. Under Windows the presence of a debugger is recognizable by looking at the PEB-structure[1] of a process. This is even possible without calling a Windows API function.
- Obviously, debugging is just a special form of running the program. That is why care needs to be taken. Basically, the same options that have been pointed out in the last section for the use of the Sysinternals tools apply here.
- It is tedious, it takes time, and it requires an experienced user. Consider the following example: Most malware nowadays is exe-packed. The first action of an exe-packed binary is to decompress itself and then call the original entry-function. Following the decompression-algorithm (which effectively is a long loop that applies a decryption function to each compressed byte) in a debugger by single-stepping takes too long. Thus, the user must be able to understand the assembly code responsible for decompression and set a breakpoint which triggers after the decompression has taken place.

Despite all its disadvantages, debugging is still a very powerful and flexible approach. It greatly helps to have the Windows debugging symbols installed. This package, released by Microsoft, consists of PDB-files for all Windows code (System DLLs such as kernel32.dll as well as system files such as ntoskrnl.exe). If this symbol information is available, calls to Windows API functions can be recognized as such (one sees the function name instead of the address only). Needless to say, the debugger must have support for reading in symbol information from PDB-files. Popular assembly-level debuggers for Windows include *WinDbg* by Microsoft (which is part of the Debugging Tools for Windows package [20]), *OllyDbg* [48], a freeware application by Oleh Yuschuk, or *IDA Pro* [14].

3.3 Disassembler

While debugging an executable involves executing it, disassembling does not. Disassembling is the process of transforming the code into assembler mnemonics. One of the problems is that it is not always clear where the code starts and where data ends. In particular, disassembling exe-packed or encrypted binaries is only possible for the small portion of code that contains the decryptor-routine. As described in [43], disassembling obfuscated binaries is difficult.

[1] Process environment Block

Chapter 4

Design and Implementation of TTAnalyze

TTAnalyze is a tool for analyzing the behavior of Windows PE-executables with special focus on the analysis of malware. To this end, the executable is run in an emulated environment and its actions are monitored.

TTAnalyze is a command line tool that runs on Windows NT/2000/XP/2003. The name of the executable to be analyzed is passed as an argument to TTAnalyze. The tool was written in C++.

In this chapter, we explain the design and implementation of TTAnalyze. We focus on illustrating the "big picture". Nevertheless, TTAnalyze's implementation is explained accurately, and details (even source-code) is shown, when it serves understanding. We state the reasons for decisions we made during the design and implementation phase and point out alternatives to our decisions together with their advantages and disadvantages.

4.1 System Architecture

Before delving into the details of TTAnalyze's implementation, we present TTAnalyze's architecture and explain how its different components interact. We look at TTAnalyze from three different points of view.

First, we divide TTAnalyze's source code into its major components and provide an overview for them. Second, we concentrate on TTAnalyze's actions at runtime and picture its dynamic behavior. Third, we show a sample invocation of TTAnalyze.

4.1.1 Components

TTAnalyze consists of four major components. See Figure 4.1 for a graphical overview.

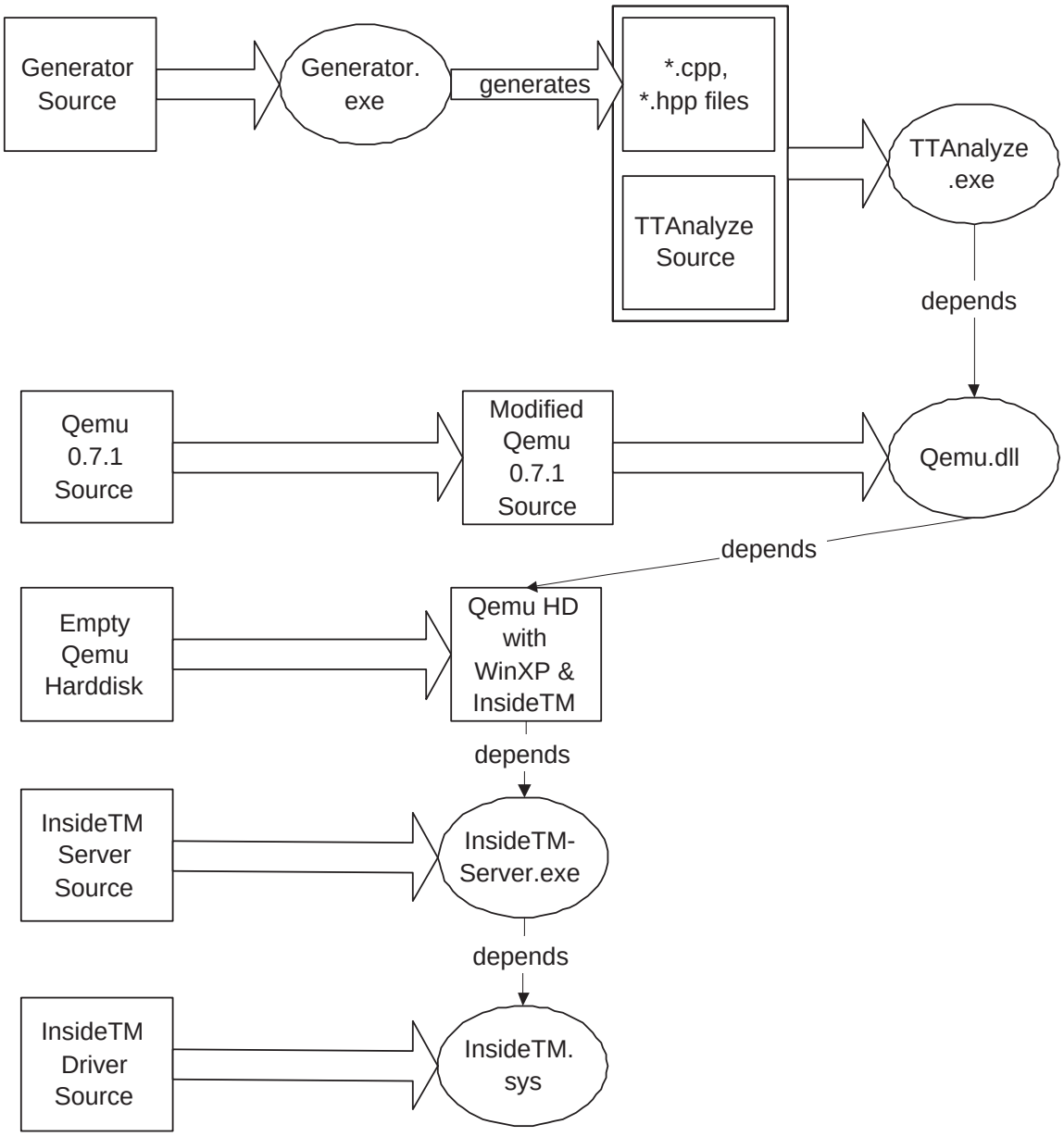


Figure 4.1: Overview of TTAnalyze

Emulation environment

Of course, we need an emulator to run the test-subject. For this, we chose QEMU [5]. QEMU is an open source PC emulator. It is one of TTAnalyze's main components. However, QEMU cannot be used without modification, several changes were of course necessary. In particular, QEMU was transformed from a standalone executable into a Windows shared library (DLL). TTAnalyze.exe makes use of QEMU by calling the DLL's exported functions. The modified PC emulator boots from a harddisk, which has Windows XP installed. The PC emulator's harddisk is emulated by a file on the host computer. The lengthy boot-process is avoided by starting QEMU from a snapshot file, which represents the state of the PC system at a certain point in time. In the case of TTAnalyze, the snapshot was created at a time, where both Windows XP and InsideTM were running.

InsideTM

Another main component of TTAnalyze is InsideTM. This component has been written by ourselves. InsideTM, which stands for "Inside The Matrix", is a program that runs inside the the virtual system. This means that it is installed on the PC system that is run by QEMU. Its main task is to act as a bridge component between the emulation environment and the outside. To this end, it implements an RPC [1] server that allows a file to be uploaded from an RPC client , and be executed. TTAnalyze makes use of this functionality when it uploads an executable for analysis and executes it in the virtual system.

InsideTM also supplies important information that allows tracking the execution of the test-subject and distinguishing between CPU instructions of the test-subject's process and other processes. This last point is essential because the emulated CPU does not only execute instructions of the test-subject, but also instructions of the Windows operating system and of several Windows' user-mode processes. Therefore, a mechanism is required that enables TTAnalyze to determine for each CPU instruction whether or not this instruction belongs to the test-subject. InsideTM provides exactly this mechanism.

As can be seen in Figure 4.1, InsideTM itself consists of two components: A normal executable and a kernel mode driver. The kernel mode driver allows InsideTM to read memory that cannot be accessed by a Windows user-mode program.

The Generator

To analyze a binary, TTAnalyze executes it in the virtual system and monitors its actions. In particular, the analysis focuses on which operating system services are used by the binary. A Windows application typically accesses OS services by dynamically linking to system DLLs and calling their exported functions.

We have built an analysis-framework that automatically calls a user-specified callback

[1] Remote Procedure Call

function whenever the application calls a specific operating system function[2]. In this callback function, the arguments of the function call are evaluated, and can be logged subsequently. For example, if the test-subject calls the operating system function `NtCreateFile`, a callback function is invoked where we naturally want to access the argument which specifies the name of the file to be created. Normally, accessing an argument means reading it from the virtual system by specifying its memory address and size. This implicates that the writer of a callback function has to know the size of its arguments. Clearly, accessing function call arguments this way would be tedious and it would certainly reduce the number of callback functions.

The Generator is a standalone executable, which generates C++ source code for reading the arguments of a function call from the virtual system. Now, the writer of a callback function can access the arguments of an operating system function call by simply reading the arguments of the callback function. Hence, there is not a single callback function, but one individual callback function for each monitored operating system function. The parameter list of a callback function mirrors the parameter list of its corresponding operating system function.

The Generator requires a file containing the declarations of all monitored operating system functions as input. By parsing the function declarations, the Generator is able to determine the argument sizes of a function and can thus generate the appropriate C++ code for reading it. As shown in Figure 4.1 the generated C++ code is compiled and linked to TTAnalyze.exe .

Analysis-Framework

The Analysis-Framework is the fourth major piece of TTAnalyze. It is the component that is responsible for performing the actual analysis work. It monitors operating system function calls, evaluates them and generates the analysis-report afterwards.

4.1.2 Threads

TTAnalyze is a multithreaded application. Here, we describe TTAnalyze's different threads and their purposes.

After modifying QEMU, it runs (as a standalone executable) in four threads:

- *QEMU primary thread* - QEMU's main execution loop, which reads, decodes, and afterwards executes an instruction is the only thread that exists right after QEMU has started.

- *Windows TAP driver thread* - The TAP driver is a piece of software that is needed for handling the network traffic between the emulated environment and the outside. The Windows TAP driver patch (as explained in 4.5.1) adds a thread to QEMU. This thread writes and reads data from the Windows TAP device in regular intervals.

[2]We use the term "operating system function" as a generic term for Windows API and native API functions.

- *QEMU timer thread* - QEMU sets up a dedicated timer thread. It is used to process all work that has to be dealt with in regular intervals. Hence, it would be possible to merge the Windows TAP driver thread with this one. The thread is created implicitly by calling the Windows API function *timeSetEvent*.
- *SDL thread* - QEMU currently uses the SDL library [39] for providing a platform-independent GUI. The library requires its own thread.

TTAnalyze, which uses QEMU in the form of a shared library, has up to five threads. Its own main thread as well as QEMU's four threads. TTAnalyze starts with a single thread, but creates four additional ones when it starts the virtual system (i.e. QEMU).

4.1.3 Sample Invocation

A typical invocation of TTAnalyze looks like this:

```
TTAnalyze <filename>
```

<filename>is a placeholder for the filename of the executable that should be analyzed, called *test-subject* in the following. For the complete list of accepted arguments see appendix B.

Here is an example of TTAnalyze analyzing a Hello-World program:

```
C:\TTAnalyze>TTAnalyze.exe C:\HelloWorld.exe
[VIRTUAL_SYSTEM]: Starting the virtual system...
[VIRTUAL_SYSTEM]: Successful start: The virtual system is up and running.
[VIRTUAL_SYSTEM]: Uploaded file C:\HelloWorld.exe.
[VIRTUAL_SYSTEM]: Executing 'HelloWorld.exe' in the virtual system.
[MAIN]: Stopping the virtual system...
[MAIN]: Time needed: 15s
[MAIN]: Writing the report to
'C:\TTAnalyze\Report_HelloWorld.exe\ttanalyze_report.txt' and
'C:\TTAnalyze\Report_HelloWorld.exe\ttanalyze_report.html'.
```

We can see that TTAnalyze's first step is to start the virtual system. Second, the test-subject, in this case the file HelloWorld.exe, is uploaded with the help of InsideTM. After this step, the test-subject can finally be run in the virtual system. Of course, its execution is monitored by the analysis-framework. The gathered information is reported in the files ttanalyze_report.txt and ttanalyze_report.html. The HTML-file might look better but otherwise presents the same information.

4.2 InsideTM

InsideTM, an abbreviation for "Inside The Matrix", consists of an executable named InsideTMServer.exe and a Windows device driver called InsideTM.sys. InsideTMServer.exe

depends on the driver for reading kernel-mode memory and will not start without it. They both run "inside the matrix", meaning that they run inside the emulated environment.

InsideTM fulfills two purposes:

- It is a means for loading the test-subject into the virtual system and have it run there.
- It serves as an information source that provides valuable data about the execution of the binary.

4.2.1 InsideTM Server

InsideTM is a typical network server that runs in an infinite loop waiting for clients to connect and send requests. Although writing a network server was our first thought, there are other options. The crucial idea is that there is an application running inside the virtual environment, which fulfills essential tasks, and we have to find a way to communicate with it. Creating a server that listens on the network and setting up QEMU so that network traffic is supported has the following advantages:

- Standard software engineering task - So developer experience and well established software components are available.
- Easy testing - It can easily tested outside the virtual system. Client and server can both run on the same computer.
- QEMU networking - It is possible that we are going to require QEMU networking support anyways. For example, in the future we might decide that the test-subject should be able to contact (certain) other computers.

The only drawback of configuring QEMU to allow networking is that we must guarantee that malware, which is run inside the virtual system, cannot escape and cannot infect other computers. This problem can be solved by implementing a packet filter, which only allows packets going to or coming from the InsideTMServer.

Instead of using the network as a communication medium, a number of other approaches are possible. A Windows user-mode program such as InsideTM has many ways to receive input and to send output. For example, there is the stdin/stdout stream, serial and parallel devices, files, network cards, keyboard input, mouse input. The main requirement is that one has to get InsideTM's output outside of the virtual system and to get input from outside into the virtual system. We control the PC emulation (and in particular, the emulation of the physical devices) and can thus access I/O that InsideTM sends and receives from a physical device. We have no access to output that is sent to e.g. a stdout stream that is saved somewhere inside the emulation's physical memory, as we have no knowledge where the Windows operating system, which is run by the emulated PC, saves this output. Summing up, we can get InsideTM's output outside of the virtual system and

can get input from outside into the virtual system if InsideTM uses I/O that is backed by a real physical device (as opposed to e.g., a stdin/stdout stream which is attached to a console window). This leaves primarily the serial port or a parallel port as alternatives. QEMU running under Windows cannot forward the emulated serial or parallel port to anything useful such as the real serial port on the host PC though. Moreover, it would more difficult to write a client and server that communicate via serial or parallel port.

So we have implemented a server that listens on the network card for client requests and sends its replies back via the network card. We do not use the socket API, but use Microsoft RPC for the network communication. RPC shields us from the complexities of low level socket-programming. Using RPC results in less implementation work.

RPC Server interface

In this section, we are explaining the RPC interface that InsideTMServer.exe provides. The interface is completely specified in the file insidetm.idl, but we are only going to list the important RPC-functions here. The IDL file is written in a language similar to C. During the build-process, it is processed by MIDL[3], which generates C-Code for the server and client stubs.

```
void rpcUploadFile([in, string] const char *filename,
                [in, size_is(data_len)] const unsigned char file_data[*],
                [in] unsigned int data_len);

unsigned rpcDownloadFile([in, string] const char *filename,
                         [in] unsigned int file_pos,
                         [out] unsigned char file_data[FILE_DATA_SIZE]);

Rproc_Handle rpcExecuteSP([in, string] const char *filename,
                          [in, string] const char *argv);

PhysicalAddress rpcGetPageDirectoryAddress([in] Rproc_Handle handle);

VirtualAddress rpcGetPEBAddress([in] Rproc_Handle handle);

unsigned int rpcResumeExecution([in] Rproc_Handle handle,
                                [in] unsigned int milliSeconds);
```

`rpcUploadFile` uploads a file to the server. The server is going to save the file in its current working-directory. If the file already exists, `file_data` will be appended to the existing file. This way also larger files may be transferred. This function is normally used for uploading the test-subject and files it depends on.

[3]Microsoft IDL Compiler

`rpcDownloadFile` downloads a file from the server. The function returns the number of bytes read. `file_pos` determines the starting position of the read process. Since only `FILE_DATA_SIZE` bytes can be transmitted at once, several rpc-calls will be necessary for downloading larger files. Typically, this function is used for downloading the DLLs that a test-subject depends on. It is necessary to have local copies of these DLLs, so that TTAnalyze can determine their exported functions together with their addresses. This downloading process can be skipped when a local copy already exists. Moreover, `rpcDownloadFile` is used for downloading the files that get created or modified by running the test-subject in the virtual system.

`rpcExecuteSP` tells the server to run the specified executable in a new process. The process is created in suspended state. This means that Windows creates the process, but does not run it until `ResumeThread`[4] is called. It is important to not immediately run the process after creation, because we have to determine the physical address of the page directory for this process before the process is run as explained in the following paragraph.

`rpcGetPageDirectoryAddress` retrieves the physical address of the page directory for the process specified by the argument "handle". This information is essential for tracking the execution of the specified process and distinguishing between CPU instructions of this process and other processes. Windows assigns each process its own, unique page directory address. Giving each process its own page directory protects the processes (their virtual memory address space) from each other and ensures that each process has its own virtual memory space. The page directory address of the currently running process is always stored in the CR3 CPU-register. Consequently, Windows loads the CR3 register on every context switch.

The CR3 register, also known as the PDBR (page-directory base register) contains the physical address of the base of the page directory. To be precise, only the 20 most significant bits of the address are stored, the other 12 bytes are used for storing two 1-byte flags and 10 bits are undefined. The 12 least significant bits of the PDB address are assumed to be 0. Thus, the page- directory base is always aligned to a 4KByte boundary. The processor reads the CR3-register in order to obtain the address of the PDB. The CPU needs the PDB for translating virtual addresses to physical addresses.

Of course, other process identifiers such as the process id, which Windows assigns to every process, exist, but we require an identifier that can be efficiently used in comparisons. For obtaining the process-id it is necessary to read memory-contents from the virtual system, whereas the value in the emulated CR3 register can be obtained by reading an attribute of the CPU struct. Reading memory from the virtual system requires a function call and hence is dramatically slower.

When an instruction gets executed, we just have to look at the value of the CR3 register and compare it to the value that is returned by `rpcGetPageDirectoryAddress` in order to determine whether or not this instruction is from the test-subject. It is clear now that `rpcGetPageDirectoryAddress` must be called before the first instruction of the process is executed. Otherwise instructions of the process get already executed, but TTAnalyze does

[4]ResumeThread is a Windows API function.

not know it.

`rpcGetPEBAddress` retrieves the address of the PEB block. The Process Environment Block or PEB, is a structure that Windows maintains for each process. What makes the PEB special is that it lies in the user-mode address space of a process (in the region below 0x80000000). In particular, the PEB lists all loaded DLLs of a process. This RPC function is not absolutely necessary, as there are also other ways to get the PEB address.

`rpcResumeExecution` makes the server call ResumeThread for the suspended process. It is going to block until the specified program has finished its execution or the timeout as specified by the "milliseconds" argument has elapsed. The return valued can be interpreted the same way as WaitForSingleObject's one.

Order of execution `rpcUploadFile` is normally the first RPC-function that is called. It uploads the test-subject. Afterwards `rpcExecuteSP` is invoked, followed by calls to `rpcGetPageDirectoryAddress` and `rpcGetPEBAddress`. Finally, TTAnalyze performs a call to `rpcResumeExecution`.

4.2.2 InsideTM Driver

InsideTM Driver is a simple Windows XP kernel-driver that has only one purpose: It shall determine the physical address of the page directory base (PDB) when supplied with a process ID. The RPC function `rpcGetPageDirectoryAddress` forwards requests to this driver. It is necessary to write a kernel-driver, because the PDB-address is stored in a memory region that is only accessible to the Windows NT kernel (and device drivers).

The PDB-address is an attribute of the EPROCESS (executive process) struct. The EPROCESS struct represents a process at the kernel-level. The list of processes - consisting of EPROCESS members - is maintained as a linked list. The kernel variable `PsActiveProcessHead` points to the first member of the process list. It is InsideTM driver's responsibility to iterate through this list until the EPROCESS struct with the specified process ID is found. Then, it reads the PDB-address from the struct and returns it. For determining the address of the variable `PsActiveProcessHead`, TTAnalyze requires the debug information for the file ntoskrnl.exe (which contains most of the kernel-code). There is no (documented) kernel function for getting this list, as this information is normally not needed by device driver authors. However, Microsoft provides debug information in the form of PDB-files (Program Database) for all of its system files and for all versions of Windows. This is mainly a service to device driver authors and facilitates debugging of device drivers. The debug information contains only the names of global variables and functions - everything else has been stripped. For our purposes this is sufficient though, because `PsActiveProcessHead` is a global variable. This is why the file ntoskrnl.pdb is part of the TTAnalyze DVD-image. Following the principle of running as much code as possible in user-mode, the process of finding `PsActiveProcessHead's` address is not done in kernel-mode but in user-mode. The InsideTM server program determines the address and reports it to the driver later (IO control code INSIDETM_SET_NTOS_ADDR).

The Windows API function `DeviceIoControl` is used for sending IO control codes and data to the driver, as well as for receiving data from the driver. InsideTM Driver understands the following three device IO control codes:

- *INSIDETM_GET_VER* - This control code causes the driver to return its version numbers.
- *INSIDETM_SET_NTOS_ADDR* - It allows the client to report the address of the symbol PsActiveProcessHead to the server.
- *INSIDETM_GET_PCB* - This is the workhorse of the driver. It advises the server to return the physical address of the PDB.

4.3 Analysis-Framework

The analysis-framework is the heart of TTAnalyze. All other features and components only exist to support the analysis-framework. The analysis-framework performs the task of analyzing the execution of the test-subject. Here, we are going to reveal how it works and how the source code is organized. We are also discussing its strengths and weaknesses.

4.3.1 Mode of Operation

We aim to determine the purpose and behavior of a binary. To this end, the executable is run in a virtual environment and its actions are monitored.

Every action (such as creating a file) that involves communicating with a device (e.g., ethernet card, harddisk) requires a Windows user-mode process to make use of an appropriate operating system service. There is no way for a Windows user mode process to directly interact with a physical device. The reason for this stems from the design of modern operating systems. Modern operating systems prohibit direct hardware access, so that multiple processes can run concurrently and can share the same hardware.

A Windows NT based operating system[5], as an example of a modern operating system, runs in a more privileged processor mode than normal applications and so only the operating system is able to execute I/O instructions and other privileged instructions on the processor. An application thus relies on operating system services when it wants to create a file or perform another action that requires interaction with a device. An operating system that runs in a privileged CPU mode must offer a way for applications to let the operating system take control and to let the operating system execute privileged CPU instructions on behalf the application. Normally, this is accomplished via system calls.

System calls are special function calls to services exported by the OS for use by applications. Performing a system call involves switching to privileged mode and transferring control to the operating system. On Intel platforms this is normally realized by an interrupt. Windows user-mode applications execute an "int 2E" CPU instruction, which

[5]This includes Windows XP, Windows 2000 and Windows 2003

triggers a software interrupt, and causes the CPU to transfer control to the exception handler that is responsible for the exception code 2E. The exception handlers are registered by the operating system during its startup. User-mode applications pass a numeric argument in the EAX register to indicate the system service being requested. Recent Intel processors (Intel Pentium II and higher) have a special "sysenter" instruction for faster and more efficient system calls. On these platforms (new AMD processors have a similar feature) Windows system calls make use of this "sysenter" instruction instead of the "int 2e" instruction.

Thus, it is reasonable to monitor the system calls that a process makes in order to analyze its behavior. On Windows things are a bit more complicated though. The actual system call interface, called Native API interface, is mostly undocumented and not meant to be used by applications. Applications are supposed to use the documented Windows API, which in turn calls native API functions when necessary. The Windows API adds a layer of indirection and shields applications from changes[6] and subtle complexities in the native API. The native API is implemented in the file ntdll.dll. It consists of all exported functions that start with the prefix "Nt".

Consider the case when an application wants to create a file. For these purposes, it calls the Windows API function CreateFile (directly or indirectly), which in turn calls NtCreateFile (exported by ntdll.dll). NtCreateFile is just a small stub that sets the EAX register to the right value (i.e., the value corresponding to CreateFile) and then raises an interrupt (or a sysenter instruction on newer processors).

The Windows API is documented by Microsoft in the Platform SDK [22]. Parts of the Native API are documented by Microsoft in the Windows DDK [19] and the Windows IFS kit [21]. Moreover, Gery Nebbett has written an unofficial documentation of the native API [26], which covers about 90% of the functions in the Native API. Malware authors sometimes use the Native API directly in order to avoid DLL dependencies and in order to confuse virus scanner's emulators. For this reason, TTAnalyze monitors not only the Windows API function calls of an application, but also its Native API function calls. Native API functions are more uncomfortable to use (they tend to have many parameters) and they are only badly documented or not documented at all. Consequently, hooking the native API is more work than hooking the Windows API. Nevertheless, hooking the Native API is a worthwhile task. It increases the reliability of the analysis. If we only monitored the Windows API, we could never be sure that the application does not circumvent the Windows API by directly calling a Windows Native API function.

4.3.2 Architecture

This section gives an overview about the architecture and design of the analysis framework. Figure 4.2, an UML class diagram, shows the important classes.

[6]The native API may change between Windows and service pack versions.

Figure 4.2: Analysis framework

CVirtualSystem

As can be seen in Figure 4.2, the class `CVirtualSystem` [7] shields the rest of TTAnalyze's source code from QEMU. `CVirtualSystem` is the only class that talks to QEMU. The rest of the system talks to `CVirtualSystem`. If we replace QEMU with another emulator, we will only have to change the implementation of `CVirtualSystem` (and probably the emulator itself).

`CVirtualSystem` provides methods for reading the registers of the emulated CPU, for reading the virtual system's main memory, for starting the system and for stopping the system. Most importantly, it provides the method *execute*. It is used for running the test-subject in the emulated environment.

`CVirtualSystem` is a singleton-class (i.e., only one instance of this class can be created). This makes sense because qemu.dll is not suited for running multiple emulator instances. Moreover, it provides every class in the analysis-framework with an easy way to obtain a reference to the virtual system.

`CVirtualSystem` depends on the class `CRpcClient` for communicating with the InsideTM RPC server, which runs inside the virtual system. `CRpcClient` hides all RPC interaction behind its interface.

CAnalysisCoordinator

The class `CAnalysisCoordinator` coordinates the analysis work. Each `CVirtualSystem` instance has exactly one `CAnalysisCoordinator` instance assigned. Since `CVirtualSystem` is a singleton, only one instance of `CAnalysisCoordinator` exists. In the following, the *analysis coordinator* refers to this single instance of `CAnalysisCoordinator`.

The analysis work is distributed among several *analyzer objects*. An analyzer object is an instance of a class that implements the `CAnalyzerObject` interface. For example, TTAnalyze has two analyzer objects. There is `CDisassembler`, which disassembles a translation block, and there is `CFunctionCoordinator`, which we explain in the following paragraph. Each analyzer object has to be registered with an appropriate instance of CAnalysisCoordinator by calling its `registerAnalyzerObject` method.

`CAnalysisCoordinator's analyzeCallback` method is indirectly called by QEMU before a translation block is called. Note that QEMU executes a sequence of instructions, called a translation block, at once as explained in Chapter 2.3.2. The analysis coordinator performs some basic analysis work, but its main task is to call back all registered analyzer objects, whenever QEMU is about to execute another translation block. To be precise, the analysis coordinator calls the `analyze` method of all analyzer objects, whose callback mode is satisfied, for a translation block. The callback mode for an analyzer object is specified at the time of registration. Currently, two callback modes are supported.

- *OWN_PROC_USERM* - The analysis object is called back for all user-mode translation blocks.

[7] We prefix all class names with a big "C".

- *OWN_PROC_USERMEP* - The analysis object is called back for all user-mode translation blocks after the entry point of the program has been reached. This mode makes sense because a process' first instruction is not the one at the executable's entry-point[8]. Before the entry point is reached, a lot of boiler plate windows process-startup code has already run in the context of this process.

Besides analyzer objects, event objects can be registered at the analysis coordinator. Event objects are instances of classes that implement the `CEventObject` interface. Hence, they provide an implementation of the `event` method. When certain events of interest take place (e.g. when the test-subject's entry point has been reached, when a DLL was dynamically loaded,...), the `event` method of event objects is called by the analysis coordinator. The type of event is passed as an argument to `event`.

In addition, `CAnalysisCoordinator` enables other classes to obtain a reference to a `CAnalysisInformation` instance by providing the method `getAnalysisInfo`.

CFunctionCoordinator

The class `CFunctionCoordinator` determines whether an operating system function call is happening and if so calls back all registered objects. Only one instance of this class exists. `CFunctionCoordinator` implements the `CAnalyzerObject` interface and so it is called for every user-mode translation-block.

The function coordinator gives other objects the possibility to be notified when an operating system function is called. While the analysis coordinator calls back instances of `CAnalyzerObject` for translation blocks, the function coordinator calls back objects for function calls. For each operating system function that is monitored, a callback-interface exists. For example, the callback-interface for the function

```
int NtClose(unsigned __int32 Handle)
```

looks like this:

```
class CNtCloseInterface: public CCallbackObject
{
public:

    virtual void NtCloseWasCalled(unsigned __int32 Handle) = 0;

    virtual void NtCloseHasReturned(unsigned __int32 Handle,
                                    __int32  _functionResult) = 0;

};
```

[8]The entry-point of a Windows executable is specified in its PE-Header.

An object has to implement the callback-interface corresponding to the operating system function that is monitored and be registered at the function coordinator before the function coordinator is able to call it back. Hence, if an object wishes to be called back when the test-subject performs a call to `NtClose`, then it has to implement the `CNtCloseInterface` and register itself at the function coordinator.

As can be seen in the declaration of `CNtCloseInterface`, an object can be called back at two points in time. Either it is notified of a function call, or of its return, or both. It is often the case that the return value of the function or its out-parameters are of interest. Then, it makes sense to have an object notified of the return of a function (and not the call). The method with the postfix "WasCalled" is invoked by the function-coordinator when the function is called. The postfix "HasReturned" denotes the method that is called for a return of this function. The callback-interface for all monitored operating system functions is automatically generated by the Generator. See Section 4.4 for more details.

Determining whether a function call is happening is a relatively cheap operation. The function coordinator only needs to compare the virtual address of (the first instruction of) the translation-block that is going to be executed next with the start addresses of all the monitored functions. In case of a match, all objects that monitor this specific function are called back. Thus, the callback functions are invoked before the first instruction of a function executes, but after the "call" instruction that jumps to the function.

Determining whether a function return takes place requires that the return-address, which is pushed on the stack as part of the "call" CPU-instruction, be saved at the function call. Every time the function coordinator's `analyze` method gets called, it checks if the virtual address of (the first instruction of) the translation-block that is going to be executed next matches with the saved return-address. The check for a return-address will only be performed if the check for the beginning of a function has failed.

Of course, function calls can be nested, and can even be recursive. To cope with this situation, return-addresses are saved on a stack and only the top element is used in determining whether a function is returning. This implementation improves efficiency as a side-effect, but it assumes that each call to a function A also ends with a return from function A to its saved return-address. Normally, this is true, but there is one exception to this rule: In case an error happens during the execution of a function, a SEH exception[9] may be raised by the Windows operating system.

SEH roughly works as follows: If an application commits an error such as dereferencing a NULL-pointer, the Windows operating system raises an exception and calls back an error-callback-function provided by every program for such circumstances. As a result, an exception stops the normal execution flow and execution continues at the error-callback-function. This way, a function may be called and never return to its saved return-address. For this reason, the analysis checks for the occurrence of SEH exceptions. Detailed information SEH can be found in the Microsoft Platform SDK [22] and an article by Matt Pietrik [30].

The test-subject may have multiple threads running. Therefore, there is a stack of

[9]Structured Exception Handling exception

return-addresses for each thread in the test-subject.

Much code for the callback mechanism is automatically generated by the Generator. These parts of the callback-mechanisms are explained there in Section 4.4.

CAnalysisInformation

The class `CAnalysisInformation` functions as the central information repository of the analysis process. It stores all information that is gained during the analysis phase, and it allows other classes to retrieve already existing analysis results. For example, the filename of the test-subject, the physical address of the page directory (as soon as it has been determined), the exit code of the test-subject are all stored in an instance of this class. Although this class is not a singleton, only one instance exists, because each analysis coordinator (and there is only a single one) has exactly one instance of `CAnalysisInformation`.

The method `addInfo` is also important. Analyzer objects such as `CFileAnalyzer` report their analysis results to the analysis information object by calling `addInfo`. The method `printReports` generates the report files, when the analysis is finished. Currently, TTAnalyze always generates an HTML and a text-report. They contain the same information and differ only in the file format and presentation of the information.

4.3.3 Analyzers

The analyzers are objects that contain the *operating system function callbacks*. An operating system function callback is a method that is called back when the function coordinator realizes that the test-subject is calling an operating system function or is returning from an operating system function.

TTAnalyze employs the following analyzers:

- *File Analyzer* - The File analyzer monitors all file activity. It hooks functions such as NtCreateFile, NtWriteFile, NtReadFile, etc..
- *Registry Analyzer* - The Registry analyzer monitors all registry activity. It hooks functions such as NtCreateKey.
- *Process Analyzer* - The Process analyzer hooks functions for process and thread creation, as well as process killing.
- *Service Analyzer* - The Service analyzer monitors all interactions with the Service Control manager in order to find out which services are created, started, stopped, etc.

4.4 The Generator

The Generator is a standalone program that is used during the build-process of TTAnalyze. The Generator parses a file containing the declarations of all the operating system functions

that the programmer would like to hook and generates eight C++ source files (as shown in Figure 4.3), which afterwards are compiled and linked to the TTAnalyze executable.

Hooking in this context means giving the programmer the chance to register callback functions that are invoked when the test-subject performs a call to an operating system function. As explained in 4.3.2, it is the task of the class CFunctionCoordinator, which is a part of the analysis-framework, to recognize when a function is called and to invoke all registered callback functions.

However, simply notifying an object that an operating system function was called by the test-subject is not sufficient for our purposes. It is our goal to give the notified object easy access to all arguments of the function call. If the test-subject, for example, calls the Windows API function *CreateFile*, then we will have to look at the function arguments in order to find out which file was created.

Normally, accessing the arguments of a function call involves the following steps: First of all, the number and types of the function's parameters, in this case *CreateFile*, have to be known. Then, each argument has to be read from the virtual system's physical memory with the help of CVirtualSystem's

```
Bit8u* readMemory(VirtualAddress addr, int size);
```

method. `readMemory` takes a virtual memory address (or logical address) as argument and converts it to a physical memory address. "Size" specifies the number of bytes that should be read. Windows API functions and Native API functions obey the *stdcall* calling convention, which means that arguments are passed on the stack and that the argument-passing order is right to left. Thus, the (virtual) address of the first argument (i.e., the left-most in the declaration) is the value of the ESP-register + 4, the value of the second argument is the value of the ESP-register + 8, etc. Note that we base this address calculation on the assumption that the function's first instruction is going to be the next instruction executed.

Consider the case of accessing the arguments of NtCreateFile. NtCreateFile's function signature looks like this:

```
NTSTATUS
NtCreateFile(HANDLE *FileHandle, //OUT
             ACCESS_MASK DesiredAccess,
             OBJECT_ATTRIBUTES *ObjectAttributes,
             IO_STATUS_BLOCK *IoStatusBlock, //OUT
             LARGE_INTEGER *AllocationSize,
             ULONG FileAttributes,
             ULONG SharedAccess,
             ULONG CreateDisposition,
             ULONG CreateOptions,
             void *EaBuffer,
             ULONG EaLength);
```

As a first step, we want to obtain the name of the file that is opened. To this end, we have to access the argument `ObjectAttributes`, which is a pointer to the structure

OBJECT_ATTRIBUTES. This structure contains, among other things, the filename. In case of a pointer, the pointer-value has to be read first. Afterwards, the pointer-target has to be loaded from the address that has been obtained in the first step. Reading the pointer-value (i.e., the address of the pointer-target) only is useless, because this address is only valid in the context of the virtual system. On a 32-bit Intel-system, pointers are 4 bytes wide, so the source code for reading ObjectAttributes looks like the following.

```
CVirtualSystem *vSys= CVirtualSystem::getInstance();

Bit32u addr= *dynamic_cast<Bit32u*>(
             vSys->readMemory(vSys->getESP() + 12), 4 );

OBJECT_ATTRIBUTES attr=
  *dynamic_cast<OBJECT_ATTRIBUTES*>(
    vSys->readMemory(addr, sizeof(OBJECT_ATTRIBUTES)));
```

The source code presented will only work correctly if the struct OBJECT_ATTRIBUTES does not contain any members of a pointer-type. If a structure has members that are pointers themselves, it is necessary to not only read out these members but also their pointer-targets. We have not shown the declaration of the struct OBJECT_ATTRIBUTES here, but it has one pointer-attribute and so the real source-code looks slightly different.

Clearly, accessing a function call's arguments this way is tedious and error-prone. This is why we desire to automatically generate the source-code for reading in the arguments. In Chapter 4.3.2, we have already shown how an object is informed of the operating system function's arguments. They are passed as arguments to the callback function. The callback function looks differently for each operating system function and the parameter list of the callback function mirrors the parameter list of the hooked function. The Generator not only generates the definitions of all callback-interfaces, but it also generates the code for reading the arguments of an operating system function.

Figure 4.3 shows exactly what files are created by the generator. CFCDecoders.hpp and CFCDecoders.cpp contains the code for reading (decoding) the arguments of an operating system function (See Chapter 4.4.3). The callback functions can be found in CFCCallback-interfaces.hpp and are described in Chapter 4.4.2. FCDeclarations.hpp contains structure- and enum-declarations. It is included by some of the other generated files. We explain its purpose in Chapter 4.4.2. CFCCallbackFactory.hpp and CFCCallbackFactory.cpp contain the implementation of the class CFCCallbackFactory, which is explained in detail in 4.4.4. Besides reading arguments of an operating system function from the virtual system, it makes also sense to write arguments to the virtual system. The so-called *function call insertion* does this and is explained in detail in 4.4.5. The input files windows_api.h, ntdll.h and functions_to_be_hooked.h are described in the following section.

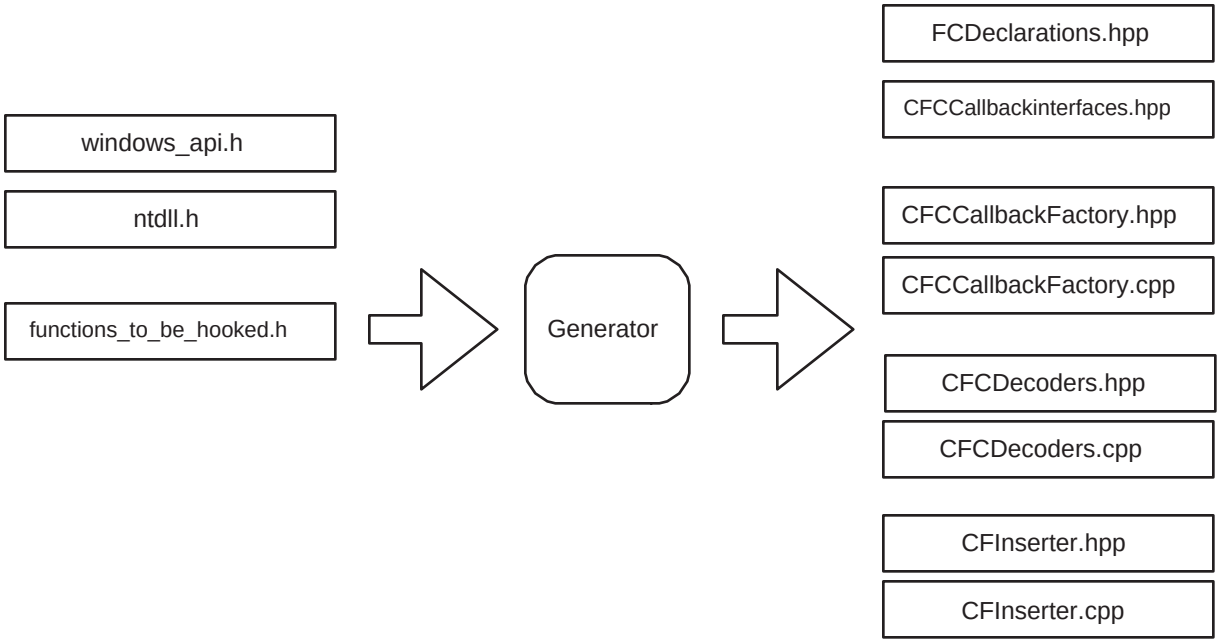


Figure 4.3: Files created by the Generator

4.4.1 Input

The Generator requires a single file that contains the declarations of all monitored functions as input. In TTAnalyze, this file is called *functions_to_be_hooked.h*. In addition, the Generator accepts several declaration-files as input. A declaration-file can include structure-declarations, typedef- declarations, enum-declarations and even function-declarations, although function-declarations are ignored. The purpose of the declaration-files is to deliver the type-declarations for types that are used as parameters or return-values in the function-declarations given in *functions_to_be_hooked.h*.

TTAnalyze's build process calls the Generator with two declaration files as shown in Figure 4.3. The first declaration file, named windows_api.h, contains all Windows API declarations. The file is the result of running the preprocessor of Visual Studio's compiler on the file Windows.h. Windows.h is the top-level include file of all headers in the Platform SDK, thus the declarations for all Windows API functions and types are found in windows_api.h. The second declaration file, called ntdll.h, contains the struct and enum-declarations used by NT native API calls. Ntdll.dll is not documented and no (complete) header-file exists that describes its functions. Thus, we had to write it ourselves. Our declarations are primarily based on Garry Nebbett's book [26] and the Windows DDK.

The Generator depends on ANTLR [28] for creating the parser and lexer for the input-files. The grammar for the input-files resembles the grammar of the programming language C. The difference is that our grammar only supports declarations and no statements. Moreover, we have slightly extended the C-syntax in two cases.

1. Extension 1 - Parameter-declarations of functions may include the keyword "[out]",

"[in]" or "[inout]". This keyword is used for specifying the direction of a parameter. It effects the point in time when an argument is read. In or Inout parameters are read when a function call takes place, while out parameters are read when the function returns. If a direction specification is missing, "in" is assumed by default.

2. Extension 2 - Array specifications of the form "[ARGx_B]" or "[ARGx_U]", where "x" stands for a number between 1 and 15, are possible. We assume that no monitored operating system function has more than 15 parameters. It means that the identifier in front of "[]" is a dynamic array with its size given by argument number x. The postfix "_B" specifies that the size is given in bytes. The postfix "_U" states that the size is given in units of the array base type. The special form "[NT]" stands for a null-terminated array. C-Strings are treated as null-terminated arrays. The reason that array arguments have to be augmented is that TTAnalyze has to know how to determine the number of elements of an array during runtime. To this end, it can either be informed about the argument that specifies the number of array elements or assume that an array is terminated by a null element. Both cases need to be indicated by proper annotation. For example, the function `int main(int argc, char *argv[])` should be declared as `int main(int argc, char argv[NT][ARG1_U])` in our header file.

We had to manually annotate the function-prototypes in the header files. In particular, we had to assign appropriate qualifiers to output and array parameters.

4.4.2 Callback-Interfaces

The file *CFCCallbackInterfaces.hpp* contains the callback-interfaces for all monitored functions. For each function that is listed in *functions_to_be_hooked.h* a corresponding callback-interface is generated. The name of the callback-interface is composed of the prefix "C", the name of the corresponding function and the postfix "Interface".

Objects have to implement the callback-interface, so that they can be called back by the function coordinator for this operating system function. Each callback-interface consists of two methods that both mirror the parameter list of the function. The method with the postfix "WasCalled" is invoked by the function-coordinator when the function is called. The postfix "HasReturned" denotes the method that is called for a return of this function. The "HasReturned" method also includes the return-value of the function in its parameter-list.

Following, we give an example of an callback-interface.

```
//__int32 NtTerminateProcess([in] unsigned __int32 ProcessHandle ,
//                           [in] __int32 ExitStatus );

class CNtTerminateProcessInterface: public CCallbackObject
{
public:
```

```
    virtual void NtTerminateProcessWasCalled(
        unsigned __int32 ProcessHandle,
        __int32 ExitStatus) = 0;

    virtual void NtTerminateProcessHasReturned(
        unsigned __int32 ProcessHandle,
        __int32 ExitStatus,
        __int32  _functionResult) = 0;

};
```

All callback-interfaces derive from the class CCallbackObject. The main purpose of this class to unify all objects deriving from one of the generated callback-interfaces under a common root. CCallbackObject is a simple, abstract class that has only one method:

```
enum ECallBackMode {CALL=1 , RETURN=2, BOTH=3};

class CCallbackObject
{
public:
    virtual ECallBackMode getCallbackMode(
        const string& interfaceName) = 0;
};
```

The method `getCallbackMode` is inherited by all callback-interfaces. Consequently, all objects implementing a callback-interface must override it. It allows an object to declare when it wants to be called back. Since objects often implement multiple callback-interfaces, the name of the interface is passed as an argument and can be used to differentiate between the callbacks for different system functions.

The file *FCDeclarations.hpp* contains all struct and enum-declarations that are required to describe the parameters of the functions listed in functions_to_be_hooked.h.

4.4.3 Decoder

The two files, `CFCDecoders.hpp` and *CFCDecoders.cpp*, together provide the implementation of the decoder-classes. Again, for each function contained in functions_to_be_hooked.h, an individual decoder-class is generated. A decoder class is responsible for reading in the arguments of its corresponding operating system function. The function-coordinator relies on the decoder-class for reading in the arguments and for invoking the callback-functions.

The decoder-class for the function `NtTerminateProcess` is shown in the following:

```
class CNtTerminateProcessDecoder: public CFCDecoder
{
```

```
private:
    unsigned __int32 ProcessHandle_in;
    __int32 ExitStatus_in;
    __int32 _functionResult;
public:
    ~CNtTerminateProcessDecoder();

    virtual void functionWasCalled(Bit32u threadId,
                                   const vector<CCallbackObject*>& cbs);
    virtual void functionHasReturned(Bit32u threadId,
                                     const vector<CCallbackObject*>& cbs);
};
```

As one sees in the example, a decoder class provides two public methods. One of them is invoked just after the hooked function was called. The other one gets called by the function-coordinator, when the function has returned. The objects that have to be called back are passed in form of the argument `cbs`. The parameter `threadId` is only used for logging purposes and can be ignored.

It is insightful to look at the implementation of the class.

```
void CNtTerminateProcessDecoder::functionWasCalled(Bit32u threadId,
     const vector<CCallbackObject*>& cbs)
{
    logger->log(LD_DECODER, LL_INFORMATION,
      "Function NtTerminateProcess was called(0x" +
       intToStr(threadId, true) + ").");

    //Save all parameters
    VirtualAddress va;
    va= vSys->getESP() + 4;
    ProcessHandle_in=
      *reinterpret_cast<unsigned __int32*>(vSys->readMemory(va, 4));

    va= vSys->getESP() + 4+4;
    ExitStatus_in= *reinterpret_cast<__int32*>(vSys->readMemory(va, 4));

    //now call back all CCallbackObjects
    for (unsigned i=0; i < cbs.size(); i++)
    {
        if (cbs[i]->getCallbackMode("CNtTerminateProcessInterface")
            & CALL)
        {
            CNtTerminateProcessInterface* cb=
              dynamic_cast<CNtTerminateProcessInterface*>(cbs[i]);
```

```
            cb->NtTerminateProcessWasCalled(
              (unsigned __int32) ProcessHandle_in,
              (__int32) ExitStatus_in);
        }
    }
};
```

The logging statement is primarily used for debugging purposes and can also be suppressed by one of the Generator's command line options. The main part of this method deals with reading in the arguments from the virtual system. The statements are similar to the ones shown in the introduction to this chapter and also similar to what one would manually write. `vSys` is a pointer to the virtual system (instance of `CVirtualSystem`) that allows one to access the address space of the monitored process. The method concludes with calling back all callback-objects. The implementation of the method `functionHasReturned` is similar and so we do not show it here.

4.4.4 Factory

Previously, we have briefly explained that the function coordinator (in its `analyze` method) obtains a reference to the decoder-object corresponding to the function-call or function-return currently happening and then invokes its method `functionWasCalled`. Until now, we have not addressed how the function coordinator obtains a reference to the correct decoder-object.

The decoder factory returns the correct decoder-object when supplied with the name of the (system) function. The factory is implemented in the two files, *CFCCallbackFactory.hpp* and *CFCCallbackFactory.cpp*. Its classname is "CFCDecoderFactory". Note that these two files contain only a single class (as opposed to a class for every function in functions_to_be_hooked.h).

The class declaration is shown below. It has no constructor and provides only a single public method. Note that this method is declared as static. The function coordinator maintains a list of function names and their start addresses. When it detects that a function call is occurring, it calls `CFCDecoderFactory::createDecoderObject(functionName)` with functionName being the name of the function in order to obtain a reference to the appropriate decoder object.

```
class CFCDecoderFactory
{
public:
    static CFCDecoder* createDecoderObject(const string& functionName);
;
```

The implementation of `CFCDecoderFactory` (as shown in the following) is simple and subject to further optimizations.

```
CFCDecoder*
CFCDecoderFactory::createDecoderObject(const string& functionName)
{
    if (functionName == "NtDuplicateObject")
    {
        return new CNtDuplicateObjectDecoder();
    }
    if (functionName == "NtClose")
    {
        return new CNtCloseDecoder();
    }
    if (functionName == "NtQueryObject")
    {
        return new CNtQueryObjectDecoder();
    }

    ....

    if (functionName == "KiUserExceptionDispatcher")
    {
        return new CKiUserExceptionDispatcherDecoder();
    }
    return NULL;
}
```

Currently, the implementation performs many expensive string comparisons. In case of an unknown function-name a NULL-pointer is returned.

4.4.5 Function Call Insertion

In this section, we explain how the Generator contributes to the "function call insertion" -mechanism. Inserting a function call into the program executed in the virtual system means that we change the test-subject's normal execution flow and cause it to call a certain operating system function before continuing. This feature is used by the analysis-framework in order to get additional information that it cannot obtain otherwise. The mechanism is explained in more detail Chapter 4.5.2.

When inserting a function call, it is necessary to copy all arguments from the host system to the virtual system. This process is the opposite of reading the arguments. The Generator generates the files *CFInserter.hpp* and *CFInserter.cpp*. They contain an inserter-class for each function in functions_to_be_hooked.h. The name of the inserter-class is composed of the prefix "C", the name of the function and the postfix "Call".

The generated source code is best explained by giving an example:

```
class CNtTerminateProcessCall: public CFunctionCall
```

```
{
private:
    unsigned __int32 ProcessHandle;
    __int32 ExitStatus;

public:

    CNtTerminateProcessCall(unsigned __int32 ProcessHandle,
                            __int32 ExitStatus);

    virtual int getStackContent(VirtualAddress stackPointer,
                                Bit32u **stack_entries,
                                unsigned *freeSize);
};
```

The constructor of an inserter-class accepts all parameters as arguments that have been labeled as "[in]"-parameters. When one wishes to insert a function call into the program executed in the virtual system, an instance of the inserter-class is constructed and all arguments to the function call are supplied in the constructor call. There is no sense in specifying values for out-only parameters.

Copying the arguments to the virtual system is not done by the inserter-class. Instead, the inserter-class just prepares the arguments correctly and returns them in a way that allows them to be easily pushed on the stack in the virtual system. In the process of inserting a function call, the method *getStackContent* gets called. Its argument "stack_entries" is an out-parameter and is used to return the individual values that have to be pushed on the stack. The first member in the stack_entries array is the first value that is pushed on the stack.

We do not show the source code for the implementation of *getStackContent* as it is not easily readable. The main challenges in the implementation of *getStackContent* have been to correctly deal with structures and pointers (structures may of course contain pointers too). Note that it is not sufficient to only push a pointer value on the stack. The pointer value is only valid on the host-system. That is why the pointer-targets are also pushed on the stack. They are pushed before the actual parameters follow.

The parameter "stackPointer" of *getStackContent* contains the value of the virtual ESP-register at the time the function call insertion takes place. The implementation of getStackContent has to know this value so that it can calculate the value of pointer variables. As we have stated above, the first pointer target is pushed first. In other words, the first pointer target is stored at the address contained in the virtual ESP-register. Thus, the pointer-variable, which is going to be pushed later, has to contain this ESP-value. Obviously, the calculation for other pointer values has take into account the fact that other pointer-targets have already been pushed.

The parameter "freeSize" is used to return the number of bytes that have to free'ed from the stack again. This number only includes the sizes of all pointer-targets, because

the function arguments themselves are already cleaned by the called function itself as long as the called function obeys the stdcall-calling convention (and we require the stdcall calling-convention for all inserted functions).

4.5 QEMU

This section covers the modifications made to QEMU, why they were made, how they were made and design decisions that were taken during the modification process. We use a QEMU version from CVS that lies between the release of QEMU 7.1 and QEMU 7.2 as a starting point for our modifications.
As explained in the following, there exist two modified versions of QEMU.

QEMU Standalone

QEMU Standalone is a version of QEMU similar to the original one. It consists of a standalone executable named qemu.exe. It is not directly used by TTAnalyze but it is a tool for manipulating and updating QEMU's harddisk file and snapshot file, which in turn are used by TTAnalyze. This version of QEMU differs from the original QEMU by supporting networking via a TAP device, by supporting a couple of additional commandline parameters and by having the same packetfilter and packetdump mechanism as *QEMU TTAnalyze*. (These features are explained in detail later in this section.) The important point to note is that *QEMU Standalone* does not differ from *QEMU TTAnalyze* in the emulation of the PC and thus harddisk files and snapshot files can be used and shared by both of these versions.

QEMU TTAnalyze

QEMU TTAnalyze is a shared library version of QEMU. It consists of a DLL file named qemu.dll. *QEMU TTAnalyze* is the emulator component of TTAnalyze. TTAnalyze depends on this DLL and cannot start otherwise. *QEMU TTAnalyze* incorporates all modifications that are already part of *QEMU standalone*, but adds its own modifications on top of *QEMU standalone*. In particular, *QEMU TTAnalyze* is not a standalone executable any longer but a shared library that exports function for use by TTAnalyze and that has special features to support the analysis process of TTAnalyze.

The fact that *QEMU TTAnalyze* makes the same modifications to QEMU's source code as *QEMU Standalone*, but moreover adds its own modifications on top of that, is also expressed by the way these two QEMU versions are stored. Both modifications are saved in the form of a patch file as output by the "diff" command. The *QEMU Standalone* patch contains the differences between the original QEMU and *QEMU Standalone*, whereas the *QEMU TTAnalyze* patch contains the differences between *QEMU Standalone* and *QEMU TTAnalyze*. Since all modifications are managed in the form of two patch files, it is possible to easily upgrade to newer QEMU versions as long as the QEMU source code has not changed dramatically.

In the following, we detail the modifications made to QEMU. If not stated otherwise we are always referring to the *QEMU TTAnalyze* version.

4.5.1 Networking

Communication between InsideTM and TTAnalyze requires QEMU to have networking support.

QEMU emulates a whole PC and this emulation also includes a NE2000-compatible network card. The question arises, what the network card emulation does, when data has to be sent and where data can be read from. Out of the box, QEMU allows the user to choose between two different network modes:

- TAP device
- User mode network stack

TAP Device

A TAP device is a device driver that emulates a network card. A TAP device can be compared to the virtual CDROM/DVD drive as provided by the Daemon Tools package for example. The TAP device has to installed on the host computer. Note that the TAP device is different from the QEMU network card emulation. While the QEMU network card emulation is a part of the emulated PC, the TAP device is a device driver running on the host PC that behaves like a network card for all user-mode programs. QEMU (and TTAnalyze using QEMU as a shared library) is a user mode program and it can use the network card created by the TAP device driver like every other network card.

If QEMU is operated in the "TAP device" networking mode, all packets sent by QEMU's network card are used as an input for the TAP device's network card on the host PC. Vice versa, all packets sent by the TAP device's network card are used as an input for QEMU's network card. While QEMU does support this networking mode on Linux, it normally does not on Windows. A patch exists however on the QEMU mailing list, which makes this mode work on Windows too. This Windows-TAP patch is part of *QEMU standalone* (and consequently part of *QEMU TTAnalyze*).

The TAP device driver comes from the Windows distribution of OpenVPN [47].

User Mode Network Stack

User mode network stack is QEMU's other networking mode. The name refers to the fact that this mode does not depend on a kernel mode component (i.e., device driver) running on the host system and thus requires no root privileges. Instead packets sent by QEMU's network card are stripped of their ethernet, IP and TCP headers and only the packet's data is then sent by QEMU using normal socket functions. That is, QEMU as a user-mode process contacts the receiver as specified in the original packet's IP header on behalf of the sender in the virtual system. Vice versa, QEMU repackages packets that it has received

as a normal process on the host system and uses them as input for its emulated network card.

Choice of Network Mode

For our implementation of TTAnalyze, we had to choose between these two networking modes. We decided to use a TAP device. Our decision was guided by four main requirements.

1. *Allow communication with InsideTM* - The main reason for having to deal with networking at all is that TTAnalyze has to be able to communicate with InsideTM, which is running inside the virtual environment.

2. *Minimal overhead* - For each run of TTAnalyze, a data volume of several megabytes is transferred (this includes the test-subject itself, additionally needed libraries and result files). Although the time needed for a run of TTAnalyze is not as important as other factors, it has to be considered.

3. *Security* - Networking has to be secure. It must not be possible for malware to escape from the emulated environment and spread to other computers.

4. *Emulation correctness* - Programs running inside the virtual environment, which are sending and receiving packets, should behave as if run on a normal computer, with the possible exception of differences needed to satisfy requirement 3.

QEMU's user mode stack networking mode's big advantage is that it can be used by users who have no root privileges. However, for an analysis-program like TTAnalyze, requiring the user to install a kernel mode driver is acceptable, as the number of users is few anyways. The downside of QEMU's user mode networking stack is that packets are heavily modified and that not all protocols are supported. For example, it does not reliably work for ICMP messages, which breaks programs such as *ping*. When using a TAP device, packets are not modified (and no processing time is spent for modifying packets) and all protocols are supported. Thus, in terms of emulation correctness and overhead, using a TAP device is superior.

Requirement one, allowing communication with InsideTM, is fulfilled by both networking modes. Although it is possible to secure both networking modes, having a TAP device is more flexible. In the following paragraphs, this is discussed in detail.

Secure Networking

As stated above, it must not be possible for malware to escape from the emulated environment and spread to other computers. To reach this goal, it would be best not to allow network communication at all. This is not possible though, because we depend on networking to be able to communicate with InsideTM. Also, malicious code sometimes requires the availability of networking to operate properly.

We solve this problem by implementing a network packet filter. Each packet is checked before QEMU's network card is allowed to send it (which would result in an input to the TAP device). The idea is to allow only packets that are part of the communication with InsideTM and to disallow all other packets. Unfortunately, the communication with InsideTM uses normal protocols and packets just like every other program, so there is no easy 100% secure way to identify InsideTM packets. (i.e., packets sent to InsideTM or received by InsideTM)
We use the following algorithm for deciding whether to allow or disallow QEMU sending a packet:

1. Ethernet packet - Check if the packet is an ethernet packet. If yes go to step 2. If not disallow the packet.

2. ARP packet - Check if the packet is an ARP packet. If yes allow the packet. If no go to step 3.

3. IP packet - Check if the packet is an IP packet. If yes go to step 4. If no disallow the packet.

4. Source IP address - Check if the source IP address is in the list of allowed source IP's. If yes go to step 5. If no disallow the packet.

5. Destination IP address - Check if the destination IP is in the list of allowed destination IP's. If yes go to step 6. If no disallow the packet.

6. TCP packet - Check if the packet is a TCP packet. If yes continue with the next step. If no disallow the packet.

7. TCP source port - Check if the TCP source port is in the list of allowed source ports. If yes allow the packet. If no disallow the packet.

The list of allowed source IP addresses consists of the IP address that is statically assigned to QEMU's emulated network card.
The list of allowed source ports consists of the port number that the InsideTM server is using.
The list of allowed destination IP addresses consists of all valid IP addresses of the host system. Normally this list is going to contain at least two entries. One entry for the IP address of the TAP device, and another entry for the IP address of the real network card (assuming the host PC has a real network card built in).

Additionally, all packets that are sent by QEMU's network card are logged. Allowed packets and disallowed packets are both written to their own file. The format of the packet log file conforms to the packet dump format used by tcpdump [41]. This gives the user the possibility to use tcpdump and all tools built on tcpdump's library for analyzing the packet dump log file.

4.5.2 Integration into TTAnalyze

The main reason for modifying QEMU was to allow it to be used together with TTAnalyze. It was clear from the beginning that QEMU should be transformed into a shared library - as opposed to mixing QEMU's source code with TTAnalyze's source. A shared library isolates QEMU from the rest of the system. It can be built separately, and it exports a well-defined interface. In the next section, we are going to discuss why it is important to be able to build QEMU separately from TTAnalyze. Afterwards, we are going to address the open question of how the DLL interface looks like.

Compiling QEMU under Windows

Compiling QEMU under Windows is a tedious task. The reason is that QEMU's main platform is Linux, and it is developed and best-tested there. Also, QEMU cannot be compiled with Microsoft's Visual Studio, which is the most widely used C/C++ compiler for Windows.

Instead, QEMU needs to be compiled with MinGW [25], a GCC port for Windows. MinGW produces native Windows binaries, which are not limited by any licensing issues. This is in contrast to Cygwin's GCC that produces binaries depending on cygwin1.dll [9]. cygwin1.dll acts as a Linux API emulation layer. The DLL is covered by the GPL license, thus requiring programs using this DLL to be under the GPL as well.

Unfortunately using MinGW complicates the build process. It requires the developer to install another compiler, and another build environment. Note that MinGW itself does not suffice for compiling QEMU. A bourne shell that is able to run QEMU's (home-grown) "configure" shell-script, the SDL library, and the zlib library are needed. MinGW's most serious drawback is that its debug format differs from Visual Studio's. As a consequence Visual Studio's debugger cannot read QEMU's debug information. TTAnalyze has to be compiled with Visual Studio's C++ compiler. Visual Studio's debugger cannot be used to debug code residing in qemu.dll on a source-code level, which makes debugging QEMU difficult.

In the beginning, we have considered porting QEMU from GCC to Visual Studio. We have decided against it for the following reasons:

- *Inline Assembler* - The QEMU source code contains parts of inline assembler. The syntax of inline assembler is not standardized and thus would have to be adapted for Visual Studio.

- *Dyngen* - Dyngen is a standalone executable that is part of the QEMU distribution. It is used during the build-process of QEMU to create C source code. It parses the file op.o (i.e., the object file corresponding to op.c) and generates the C-function "dyngen_code" that translates QEMU's intermediate code to host CPU instructions. Dyngen makes a lot of assumptions about the object file, in particular dyngen only works for specific GCC versions. Dyngen would have to be changed in order to support object files produced by Visual Studio's compiler.

- *GCC extensions* - QEMU relies on many of GCC's extensions to the C language. For example, GCC supports the definition of global register variables, which guarantees that a certain global variable is saved in a specific CPU register. If QEMU is ported to Visual Studio, one has to deal with GCC extensions without an equivalent in Visual Studio's compiler. This can lead to serious problems when trying to port QEMU to Visual Studio's compiler

- *Unknown Risks* - QEMU is very sensitive to changes in the compiler configuration. The author, for example, has experienced problems when certain files were not compiled with full optimization.

- *Upgrading* - Porting would make upgrading to a new QEMU version more difficult.

DLL interface

As mentioned previously, TTAnalyze uses QEMU as a shared library, a Windows DLL. Basically, there are two ways to design the DLL interface:

1. *QEMU calls back* - Change QEMU in such a way that it calls a given function for every instruction executed on the virtual processor. The DLL interface consists essentially of a function `void set_callback_function(Analysis_Callback *f)` that registers the function to be called back by QEMU.

2. *The Analysis drives the emulation* - Understand the QEMU library as a whole PC-system that exports functions for initializing the PC system, configuring it, and executing an instruction on the virtual processor. The analysis-component has to initialize the PC system, execute one instruction, analyze it, execute the next instruction, analyze it, and so on.

Approach one is less obtrusive and easier to implement. For this reason we have taken approach one. It may be less flexible then approach two, but approach two requires substantial changes to QEMU.

In the following we are going to present the DLL interface in detail. The DLL interface is simple and short, so that we can easily present the complete interface. (i.e., all functions and variables that are exported by the DLL are listed here)

```
#ifdef BUILD_DLL
#define EXPORT __declspec(dllexport)
#else
#define EXPORT __declspec(dllimport)
#endif

EXPORT HANDLE mq_qemu_has_loaded; //handle to a semaphore
EXPORT HANDLE mq_analysis_is_ready; //handle to a semaphore
```

QEMU TTAnalyze exports two semaphore handles for synchronization purposes. This is necessary because QEMU runs in its own thread.

```
EXPORT int mq_init_qemu(const char *report_dir);

EXPORT void mq_clean_up_qemu();

EXPORT int mq_system_startup(int argc, char **argv);

EXPORT void mq_system_shutdown();

EXPORT void mq_start_analysis(uint32_t phys_pagedir_addr,
                              MQAnalyzeCBFunc *cb_func);

EXPORT void mq_stop_analysis();
```

`mq_init_qemu` initializes the semaphores and some other resources. It has to be called before `mq_system_startup` may be invoked.

`mq_clean_up_qemu` frees all resources that were allocated in `mq_init_qemu`. This function should be called after `mq_system_shutdown` is called.

`mq_system_startup` starts the virtual system. It is just a simple wrapper around the original main function of QEMU.

`mq_system_shutdown` stops the virtual system.

`mq_start_analysis` is called to begin with the analysis. This means that TTAnalyze is called back from now on. `cb_func` is the address of the function that QEMU calls back. `phys_pagedir_addr` is the physical address of the test-subject process' page directory (This address is different for each Windows process). QEMU calls `cb_func` only for user-mode instructions[10] of the test-subject process.

`mq_stop_analysis` is called to stop the analysis. It causes TTAnalyze to stop calling back TTAnalyze.

```
EXPORT int mq_cpu_memory_rw_debug(CPUX86State *env, unsigned int addr,
                                  unsigned char *buf, int len,
                                  int is_write);

EXPORT CPUX86State* mq_get_cpu_state();

EXPORT void mq_exec_pf_block(target_ulong addr);

EXPORT void mq_insert_function_call(TranslationBlock *tb,
                                    uint32_t functionBeg,
```

[10]Windows NT-based operating systems consider instructions that are in the memory region lower than 0x7FFFFFFF as user-mode instructions.

```
                                    uint32_t *stack_entries,
                                    int entries_num,
                                    unsigned free_size);

EXPORT void mq_set_log(int log_flags);
```

`mq_cpu_memory_rw_debug` allows reading the main memory (RAM) from the virtual system. 'addr' is a virtual address that is interpreted according to the current value in the emulated CR3-register. Only virtual addresses that map to valid (i.e., loaded) physical addresses can be read.

`mq_get_cpu_state` is just a small getter function that returns the address of the CPUX86State structure.

`mq_exec_pf_block` is executed when `mq_cpu_memory_rw_debug` cannot read the memory at a given address. It forces the target environment to make a reference to the memory address given, thus causing a page fault in the virtual system that is handled by the virtual system. The next time we call `mq_cpu_memory_rw_debug`, it should succeed.

`mq_insert_function_call` executes the function at address 'functionBeg'. The array `stack_entries` contains all parameters and its members are pushed on the stack before calling the function. The first array member is pushed first.

`mq_set_log` causes qemu to output debug information to a file named 'qemu.log'. Any combination of MQLogOption members is allowed.

Analysis Callback

Previously, we have explained that QEMU calls back an analysis-function for every instruction executed on the virtual processor. This is not entirely true. QEMU does not translate and execute a single instruction, but it translates and executes a *basic block*. A basic block is a sequence of one or more instructions that ends with a "jump instruction or an instruction modifying the static CPU state in a way that cannot be deduced at translation time" [6]. QEMU calls these basic blocks "translation blocks". It is more efficient to translate several instructions at once than only a single one.

Thus, QEMU always calls the analysis-function before it executes a basic block. This is realized by changing QEMU's translation mechanism such that the translation of each basic block starts with a call to the analysis-function. We have introduced an instruction in the intermediate representation named "callback_ttanalyze". The address of the callback-function is passed as an argument to it. Let's revisit the example that we have already presented in Chapter 2 (Figure 2.4 and Figure 2.5). The intermediate code is changed now to start with a "callback_ttanalyze" instruction.

Without discussing QEMU's internals, it is still worth looking at the implementation of "callback_ttanalyze":

```
typedef void MQAnalyzeCBFunc (struct TranslationBlock *);
void OPPROTO op_callback_ttanalyze(void)
{
```

```
add    $0x8,%esp
jmp    *0x806ed384
```

Figure 4.4: Input to QEMU's modified translation engine

```
callback_ttanalyze 0x7125cf80
movl_T1_im 0x00000008
movl_T0_ESP
addl_T0_T1
movl_ESP_T0
update2_cc
movl_A0_im 0x806ed384
ldl_kernel_T0_A0
jmp_T0
set_cc_op 0x00000008
movl_T0_0
exit_tb
```

Figure 4.5: Intermediate code generated by QEMU's modified translation engine

```
    extern MQAnalyzeCBFunc *mq_analyze_cb_func;
    extern volatile uint32_t mq_ts_pagedir_addr;

    if ((env->cr[3]& 0xFFFFF000) == mq_ts_pagedir_addr
        && ((struct TranslationBlock*) PARAM1)->pc < 0x80000000)
        mq_analyze_cb_func( (struct TranslationBlock*) PARAM1 );
}
```

The callback function is only invoked if this block is executed on behalf of the test-subject process (as determined by comparing CR3 to the known physical address of the test-subject's page directory) and if it is user-code. Still, for every translation-block several "compare" instructions are additionally executed. This decreases the speed of the emulation a bit, but it is necessary.

The alternative would be to only start translation-blocks of the test-subject with a "callback_ttanalyze" instruction. However, this is not possible. The reason is that lots of translation-blocks that are executed on behalf of the test-subject process are shared with other processes. Sharing of translation-blocks is frequently possible, because every Windows applications uses the same system DLLs (kernel32.dll, user32.dll, ntdll.dll,... to just name a few.).

Handling Page Faults

The function `mq_cpu_memory_rw_debug` is used to read memory from the test-subject's process. For example, this function is used to read the arguments of a system function call. Unfortunately, this does not always work.

The physical main memory in QEMU's emulated PC system simply is a large malloc'ed area on the host system. Together with other memory management-information, it is always possible to read from the emulated main memory when supplying a physical address to read from. When supplying a virtual address however, the virtual address has to be converted to a physical address first. Unfortunately, the possibility exists that the content is not in the emulated physical main memory, but only on the emulated harddisk (i.e., the contents are paged out). In this case, `mq_cpu_memory_rw_debug` would fail.

There are also other cases where `mq_cpu_memory_rw_debug` is not able to retrieve the contents for a virtual address. The Windows MMU (memory management unit) uses lazy evaluation as often as possible to save resources [38]. *Lazy evaluation* means waiting to perform a task until it is required. In particular, in the beginning of a process' lifetime, its page tables often do not include shared libraries used by that process. Instead, the page tables are updated, when the processor first references memory in the shared library.

`mq_cpu_memory_rw_debug` has the postfix "debug" because it is not used by normal QEMU code. It is meant to be used for debugging purposes only. In particular, it does not perform access checks and it is not going to raise a page fault in the virtual system when a non-mapped address is requested. Instead of raising a page fault[11], a return code indicating an error situation is returned.

Failing to read an argument of an operating system function call would be a serious drawback for our analysis. TTAnalyze must be able to read the memory contents at a specified virtual address. If necessary, it should invoke the page fault handler of the emulated operating system and read the memory after the page fault handler has brought the memory contents in.

We have implemented the following solution: If `mq_cpu_memory_rw_debug` fails because the specified virtual address does not refer to a valid location in physical memory, we change the execution flow of the emulated program to read from this memory location. Then, we call `mq_cpu_memory_rw_debug` again. The second call is going to succeed if the virtual memory address is valid[12]. By having the emulated program read from the memory location in question, we are provoking a page fault. After the page fault is triggered, the page fault handler is going to execute in the virtual system. When the handler has done its work, `mq_cpu_memory_rw_debug` is called again.

To give an example, Figure 4.6 shows the intermediate code that is executed if `mq_cpu_memory_rw_debug` fails to read from the address 0x77ddabb8.

As explained earlier, TTAnalyze's code hides QEMU behind the class `CVirtualSystem`. Among other methods, `CVirtualSystem's` public interface contains the method

[11] In QEMU page faults as well as all exceptions and interrupts are implemented by using the C-functions `setjmp` and `longjmp`.

[12] An example of an invalid virtual address would be 0x00000000.

```
INTERMEDIATE CODE:
movl_A0_im 0x77ddabb8
ldsb_user_T0_A0
exit_tb
```

Figure 4.6: Intermediate code for provoking a page fault

```
Bit8u* readMemory(VirtualAddress addr, int size),
```

which in turn calls QEMU's DLL function `mq_cpu_memory_rw_debug`. If `mq_cpu_memory_rw_debug` returns an error, `readMemory` throws the C++ exception `CMemReadException`. This exception is caught in `CAnalysisCoordinator::analyzeCallback`. The exception is dealt with by calling `mq_exec_pf_block`, which makes QEMU execute the intermediate code shown before in Figure 4.6.

Figure 4.7 shows a typical callstack of a situation where the exception `CMemReadException` is raised because `mq_cpu_memory_rw_debug` has returned an error. Since the function call to analyzeCallback() is the first action in a translation block, the actual translation block is not executed when a virtual memory address cannot be accessed. Remember that a CReadMemException is dealt with by calling `mq_exec_pf_block` which replaces the execution of the current translation block with the manufactured memory read operation.

It is important to note that the virtual EIP register is not changed by `mq_exec_pf_block`. That is, after executing the injected memory read operation and possibly after executing the page fault handler, the original translation block is executed again. Its first action consists of calling `analyzeCallback` again. In the end, `CVirtualSystem::readMemory` is invoked again with the same memory address as before, but this time the call is going to succeed.

Is it possible that a second call to `CVirtualSystem::readMemory` causes another `CMemReadException` and thus triggers an infinite recursion? Fortunately, this cannot happen. The worst case scenario is that we perform a call to `CVirtualSystem::readMemory` passing an invalid virtual address as argument. This means that we would make a reference to an invalid address on behalf of the test-subject process, thus causing a page fault that Windows cannot satisfy. Windows reacts by sending an "Access Violation"-SEH-exception to the test-subject-process, which normally results in its termination.

However, the readMemory method is mainly used for reading in the arguments of calls to system functions. If an invalid parameter were passed in such a call, we would merely provoke the access violation a couple of instructions before it would normally happen (i.e., without us modifying its execution flow). This could be avoided by embedding the memread-instruction into an SEH protection block. If done, access violations are caught. It can lead to infinite recursion situations though, so we moreover have to implement a solution for this too.

Insertion of Function Calls

In the last section, we have explained that TTAnalyze changes the execution of the test-subject. That is, if necessary, it makes the test-subject read from certain memory addresses. In other words, TTAnalyze *inserts* memory-read instructions. It makes sense to not only insert memory-read instructions, but also calls to Windows API functions (or more generally calls to a function that is exported by a DLL).

Inserting function calls is necessary for the following reasons:

- *File created or opened* - The Windows API function `CreateFile` and its native API equivalent `NtCreateFile` can both be used for creating as well as opening a file. There is no way to reliably differentiate between the opening and the creation of a file alone from the arguments used in the function call. To differentiate between these two situations, we have to insert a function that checks whether the file already exists or not. The same situation arises, when the Windows API function `RegCreateKeyEx` is called because `RegCreateKeyEx` can be used for creating as well as opening a registry key.

- *File or directory* - In several situations, it is not possible to know if a filename refers to a file or a directory.

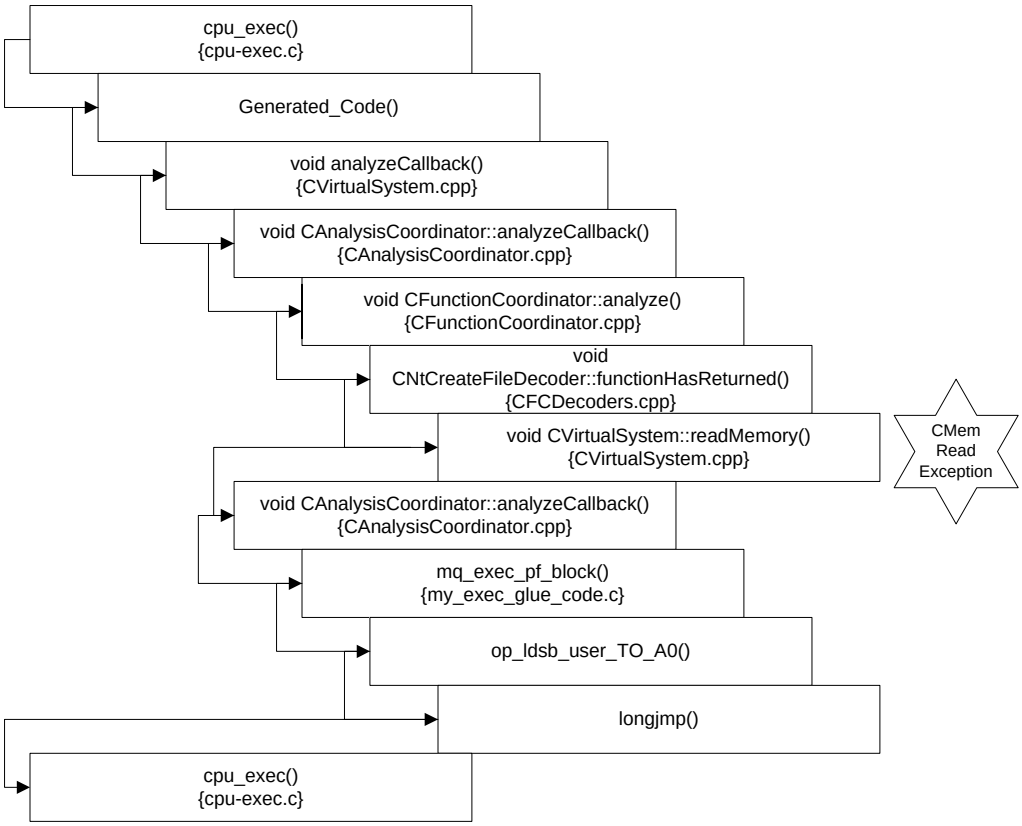
- *Unknown handles* - Normally, all Windows API, as well as native API function calls that return handles are monitored and so TTAnalyze knows to what resources these handles refer. Handles can be inherited from another process though or obtained by an unmonitored system function call. In these cases, function call insertion is required to find out information about an otherwise unknown handle.

At the instruction level, a function call is nothing more than a jump to an address (the function start) that is preceded by a push of all function arguments and the return address onto the stack. In essence, that is all what the function *mq_insert_function_call* does. First of all, *mq_insert_function_call* generates intermediate code for pushing the return address and all function arguments onto the stack. Second, intermediate code that moves the function's start address into the EIP register is created. Third, the intermediate code is transformed to host code and executed.

A function has to satisfy the following requirements before TTAnalyze can insert a function call into the test-subject's execution flow.

- *Module must be loaded* - The function's code must be in a module (DLL or exe file) that has already been loaded into the test-subject's address space. Obviously, a function cannot be called if its code does not exist. This requirement is relaxed by the fact that it's possible to have the test-subject process reload modules by inserting a function call to the Windows API function *LoadLibrary*.

- *Function must be a DLL export or its name be part of a PDB-file* - Only in these cases, the starting address of the function can be determined.


cpu_exec()
{cpu-exec.c}
Generated_Code()
void analyzeCallback()
{CVirtualSystem.cpp}
void CAnalysisCoordinator::analyzeCallback()
{CAnalysisCoordinator.cpp}
void CFunctionCoordinator::analyze()
{CFunctionCoordinator.cpp}
void
CNtCreateFileDecoder::functionHasReturned()
{CFCDecoders.cpp}
void CVirtualSystem::readMemory()
{CVirtualSystem.cpp}
CMem
Read
Exception
void CAnalysisCoordinator::analyzeCallback()
{CAnalysisCoordinator.cpp}
mq_exec_pf_block()
{my_exec_glue_code.c}
op_ldsb_user_TO_A0()
longjmp()
cpu_exec()
{cpu-exec.c}

Chapter 5

TTAnalyze Case Studies

This chapter focuses on the results of running TTAnalyze. In Section 5.1, we explain the structure of the report that is generated by TTAnalyze. In Section 5.2, we show the results of testing real malware samples with TTAnalyze and compare TTAnalyze's output to the malware descriptions of a well-known anti-virus company.

5.1 TTAnalyze's Report

TTAnalyze is a tool for analyzing malware. During the execution of the test-subject in the virtual system, its actions are monitored and analyzed. Information that is deemed essential is saved and later output to the report file. To be exact, there are two report files: a text-file and an HTML file. Both files contain the same information and only differ in the presentation of its content. Here, we describe what information is part of the report and how this information is structured.
The report is structured into five different sections:

1. General Information - This section contains information about TTAnalyze's invocation, command line arguments given, and some general information about the test-subject.

2. File Activity - This section covers the test-subject's file-activity (i.e., files created, modified, etc.).

3. Registry Activity - In this section, all modifications made to the Windows registry, and all registry values that have been read by the test-subject are described.

4. Service Activity - This section documents all interaction between the test-subject and the Windows Service Manager. If the test-subject starts or stops a Windows service, for example, this information is listed here.

5. Process Activity - In this section, the creation or termination of processes as well as other process-related actions can be found.

Although not formally part of the report, the network traffic of an application is also logged. It is written to an extra file that contains all packets sent by the emulated PC.

In the following, we describe each section in detail and show how a particular section looks in the report file.

General Information

The *General Information* section of a text-report-file is shown below. It is as real world example that was created when we analyzed the *Sober Y* worm with TTAnalyze. Note that only the *General Information* section presented directly below is taken from the *Sober Y* analysis. The excerpts shown in the following sections stem from different TTAnalyze runs.

```
Information about TTAnalyze's Invocation:
-----------------------------------------
Time needed: 53 s
Created: 04/12/2005, 15:50
Analysis End: normal


Configuration of TTAnalyze:
---------------------------------------
debug 0
disassemble 0
download C:\WINDOWS.0\WinSecurity\services.exe
execute C:\malware\cb720e191c21cc47dff1e471.bin
full-report 0
host-ip 192.168.2.1,62.178.159.202
kill C:\WINDOWS.0\WinSecurity\services.exe
log-allowed-packets 1
log-blocked-packets 1
qemu-ip 192.168.2.2
report-only-hlf 0
show-qemu 1
tap-adapter-name mytap
timeout 0
use-image winxp


Load-time DLLs:
---------------------------------------
ntdll.dll Base-address: 0x7C910000
kernel32.dll Base-address: 0x7C800000
```

```
MSVBVM60.DLL Base-address: 0x73390000
USER32.dll Base-address: 0x77D10000
GDI32.dll Base-address: 0x77EF0000
ADVAPI32.dll Base-address: 0x77DA0000
RPCRT4.dll Base-address: 0x77E50000
ole32.dll Base-address: 0x774B0000
msvcrt.dll Base-address: 0x77BE0000
OLEAUT32.dll Base-address: 0x770F0000


Dynamically loaded DLLs:
-----------------------------------------
uxtheme.dll Base-address: 0x5B0F0000
MSCTF.dll Base-address: 0x746A0000
Apphelp.dll Base-address: 0x77B10000
VERSION.dll Base-address: 0x77BD0000


Test-subject's Output:
-----------------------------------------
Exit-Code:  0
Stdout:
Stderr:
```

As one can see in the example, the test-subject shown above did not print anything to its stdout or stderr stream. Its exit code is 0, which indicates that the test-subject finished its execution without errors.

The list of loaded DLLs (whether loaded at program startup or later) gives a first hint of the functionality required by the test-subject. In this case, the test-subject depends on the DLL MSVBVM60.DLL amongst others. This file contains the Microsoft Visual Basic virtual machine and is needed by applications that are written in Visual Basic and compiled as native applications. Thus, we can conclude that Sober Y has been written in Visual Basic.

"Configuration of TTAnalyze" lists the values of all of TTAnalyze's program options. A program option can either be given on the command line or in a configuration file. Program options influence TTAnalyze's runtime behavior. The meaning of specific program option is explained in Appendix B.

"Analysis End" describes why the analysis was ended. "Normal", as in our case, means that the analysis ended because the test-subject terminated. "Timeout" is the alternative. It means that the analysis ended because a specified timeout (via the program option timeout) was reached.

File Activity

The following report excerpt shows all information that is provided in a report's *File Activity* section.

```
Files deleted:
----------------------------------------
C:\test.txt


Files created:
----------------------------------------
C:\win_test_file.txt


Files changed:
----------------------------------------
(stdout)
(stderr)


Files read:
----------------------------------------
C:\test.txt


Directories created:
----------------------------------------
C:\testDirectory


Directories deleted:
----------------------------------------
C:\testDirectory


DeviceIO Control:
----------------------------------------
\Device\KsecDD ControlCode: 3735560


File mapping objects created:
----------------------------------------
Filename: C:\WINDOWS.0\system32\Apphelp.dll(Handle: 0x7B8
```

```
Type: Filebacked)
Filename: C:\WINDOWS.0\system32\Apphelp.dll(Handle: 0x7BC
Type: Filebacked)
Filename: C:\Windows.0\AppPatch\sysmain.sdb(Handle: 0x7B8
Type: Filebacked)
Filename: C:\InsideTM\HelloWorld.exe(Handle: 0x7B4 Type: Filebacked)
Filename: C:\InsideTM\HelloWorld.exe(Handle: 0x7CC Type: Filebacked)
```

The streams stdout and stderr are treated as files. Thus, applications writing to one of these two streams cause an appropriate entry in the "Files changed" - section.

In general, all the provided analysis results are reliable. Since we are monitoring the native API, there are only two ways how an application could circumvent TTAnalyze's analysis methods and create files without TTAnalyze noticing it. First, it could invoke the appropriate interrupt for a native API call itself without calling the native API function. Second, it can use the Windows API function `DeviceIoControl` (or the corresponding native API function). This function allows an application to directly talk to a device driver. For example, an application could use this function to talk to the file system driver and send an appropriate control code for creating a file. These control-codes for drivers are mostly undocumented, but still an application can theoretically create a file by directly talking to the file system device driver. That is why all calls to this function are reported under "DeviceIO Control".

File mapping objects are used for mapping files directly into the address space of an application. After the mapping is established, normal memory-read and memory-write operations to the appropriate address space cause read and write operations to the corresponding file regions used by the mapping. Obviously, the creation of file mapping objects has to be part of TTAnalyze's report.

Registry Activity

The registry activity of an application is broken down into five different information pieces as shown in the following report excerpt. This excerpt is part of a TTAnalyze report that was created when we analyzed a test program with TTAnalyze.

```
Registry keys created:
-----------------------------------------
HKLM\Software\UlliSoftware


Registry keys deleted:
-----------------------------------------
HKLM\Software\UlliSoftware


Registry values deleted:
```

```
-----------------------------------------
HKLM\Software\UlliSoftware\UlliVersion


Registry values changed:
-----------------------------------------
HKLM\Software\UlliSoftware\UlliVersion (Data: '0xAABBCCDD'
(REG_DWORD))


Registry values read:
-----------------------------------------
HKLM\SYSTEM\WPA\MediaCenter\Installed (Data: '0x0' (REG_DWORD))
HKLM\Software\UlliSoftware\UlliVersion2 ((no data))
HKLM\Software\Microsoft\Rpc\MaxRpcSize ((no data))
```

Service activity

Here, we show the details of a report's service activity section.

```
Services created:
-----------------------------------------
UllisTestService (Type: SERVICE_AUTO_START, BinaryPath:
                  'C:\insidetm\TestService.exe')


Services deleted:
-----------------------------------------
UllisTestService


Services started:
-----------------------------------------
UllisTestService


Services changed:
-----------------------------------------
UllisTestService


Control-Codes, that were sent to services:
-----------------------------------------
```

```
UllisTestService (ControlCode: SERVICE_CONTROL_STOP)
```

"Services changed" refers to service entries in the Service Control Manager's database that have been changed. For example, its start-type, which determines if an application is automatically started when Windows starts, could be changed.

It is important to note that the monitored functions of the Windows Service API are not native API calls [1]. Instead, they are normal user-mode code and the service control manager is a normal user-mode application too. It is thus possible (though not recommended) for an application to create services by using other means. Most Windows versions allow the creation of services by writing to the Windows registry (as the database of all services is part of the Windows registry.). However, services created like that can still be catched by carefully looking at the "registry activity"-section in a report.

Process activity

The *process activity* section of a report deals with the processes and threads that were created or terminated by the test-subject.

```
Processes created:
------------------------------------------
C:\HelloWorld.exe


Processes terminated:
------------------------------------------

Test-subject's Process


Threads created:
------------------------------------------
0x7C810856 (runs in Test-subject's Process)
0x7C810867 (runs in C:\HelloWorld.exe)
```

In the example given above, the test-subject creates one thread and one process (together with its main thread) for running "HelloWorld.exe". When using the native API, creating a Windows process consists of two actions: First, the process object has to be created. Second, the main thread running in this process has to be created. This procedure is reflected in TTAnalyze's report by explicitly showing these two actions. The addresses used in "Threads created" are the thread entry-point addresses. The thread starting at 0x7C810856, which is running in the test-subject's process, is not the test-subject's main thread but an additional (second) thread. The subsection "Processes terminated" only lists

[1] All other sections in the report are backed by monitoring native API functions.

the test-subject process. The test-subject process is always listed here if the analysis ended normally (How a normal analysis end looks like, is explained in the "General Information" section).

Currently, TTAnalyze's network analysis functionality is quite limited. More precisely, all that exists is a packet dump file showing all the packets that QEMU's network card has sent. This is one of the areas where TTAnalyze has to be improved in the future.

5.2 TTAnalyze Test Results

In this chapter, we show the results of testing TTAnalyze with actual malware samples. As a starting point for our tests, we have taken the malware prevalence statistics given in Table 5.2. These are the 10 most prevalent malware samples worldwide according to the statistics of Ikarus Software. These statistics have been created on December 6th, 2005.

Malware name	*Number of Samples*
Email-Worm.Win32.Sober Y	129066
Email-Worm.Win32.Netsky.Q	10781
Net-Worm.Win32.Mytob.DE	9471
Email-Worm.Win32.Doombot.B	5618
Net-Worm.Win32.Mytob.BD	4570
Net-Worm.Win32.Mytob.BI	4437
Net-Worm.Win32.Mytob.J	3164
Win32.Sober.AD@mm	3130
Net-Worm.Win32.Mytob.C	2601
Email-Worm.Win32.Netsky.D	2023

Table 5.1: Malware prevalence table, Ikarus Software

We decided to carry out a test with all the available samples of this Top Ten-statistics. Two of them, "Email-Worm.Win32.Netsky.Q" and "Win32.Sober.AD@mm", were not available for us, and thus could not be included in the test. In Section 5.2.1, we are giving an high level overview of TTAnalyze's test results and how they compare to the virus descriptions of a well-known anti-virus company. Then, we describe two of the results in detail, discuss the difficulties that we have faced, and explain the steps that were necessary for obtaining the results.

Virus names are not unique and not standardized between different vendors of anti-virus solutions. Ikarus Software uses the same virus naming convention as Kaspersky's virus scanner. Hence, we were able to easily find virus descriptions on Kasperky's website for the virus names given in the malware prevalence table.

At Ikarus, a number of different samples are collected for a given virus name. Sometimes, they are packed differently, sometimes they are not even Windows PE-executables.

For our tests, we chose one working sample for each virus type. Furthermore, we have identified all samples by Kasperky's online virus scanner.

5.2.1 Test Results - Overview

We have tested TTAnalyze with the eight available malware samples listed in Table 5.2.1 and validated TTAnalyze's correctness by comparing the results to malware descriptions provided by Kaspersky or, if not available, to malware descriptions of Symantec.

Malware name	*File*	*Registry*	*Process*	*Service*
Email-Worm.Win32.Sober Y	✓	↯	✓	✓
Email-Worm.Win32.Netsky.Q	✓	✓	✓	✓
Email-Worm.Win32.Doombot.B	✗	✗	✗	✗
Net-Worm.Win32.Mytob.BD	✗	✗	✗	✗
Net-Worm.Win32.Mytob.BI	✗	✗	✗	✗
Net-Worm.Win32.Mytob.J	✗	✗	✗	✗
Net-Worm.Win32.Mytob.C	✓	↯	✓	✓
Email-Worm.Win32.Netsky.D	✓	✓	✓	✓

Table 5.2: TTAnalyze Test Results

For the entries containing ✗ the virus description and the results of TTAnalyze differ. For example, the names of created files differ. Nevertheless, TTAnalyze's reports have been proved correct with the help of other tools such as Filemon and Regmon. The reason for different file names could be that a virus generates a different file name (from a list of file names or even randomly) during each execution run. In this case, file names do not match precisely. However, the fact that a file was created was visible in TTAnalyze's output and the virus description. Also, the overall behavior as indicated by TTAnalyze's report and the given virus description matches. We assume that the classification into different malware variants is sometimes not as precise as it would be necessary for distinguishing their behavior. After all, most virus scanners have generic signatures that match several variants of a malware branch.

In the case of Sober Y and Mytob.C, the reason for TTAnalyze's failure to recognize all created registry values seems to be that the Windows client-server subsystem process csrss.exe makes the modifications to the registry on behalf of the test-subject process. Since only the test-subject process is monitored, these interactions with the registry are not noticed. However, there is no inherent restriction in TTAnalyze's design that prohibits monitoring more than one process.

5.2.2 Email-Worm.Win32.Sober Y

The beginning of December 2005 saw a Sober Y outbreak. Sober Y replicates by emailing itself to other computers. The number of emails infected by Sober Y was extremely high

(higher than any other worm in the last few months) and so, most anti-virus vendors described Sober Y as being a critical threat at the time of writing.

TTAnalyze Run 1

According to TTAnalzye, the following events occur.

- Created Directories
  - C:\WINDOWS.0\WinSecurity
- Created Files:
  - C:\WINDOWS.0\WinSecurity\csrss.exe
  - C:\WINDOWS.0\WinSecurity\services.exe
  - C:\WINDOWS.0\WinSecurity\smss.exe
  - C:\WINDOWS.0\WinSecurity\socket1.ifo
  - C:\WINDOWS.0\WinSecurity\socket2.ifo
  - C:\WINDOWS.0\WinSecurity\socket3.ifo
  - C:\WINDOWS.0\system32\dllcache\tcpip.sys
- Changed Files
  - C:\WINDOWS.0\ServicePackFiles\i386\tcpip.sys
  - C:\WINDOWS.0\system32\drivers\tcpip.sys
- Started Processes
  - C:\WINDOWS.0\WinSecurity\services.exe

We downloaded all created files from the virtual system and quickly determined that the files csrss.exe, services.exe and smss.exe were almost identical copies of the original file. They only differ in one byte at position 0xA0 (which is an otherwise unused byte in the PE-file header.). Moreover, we observed that the services.exe process has to be killed before the InsideTMServer is able to open the file services.exe and send it to the host system. We concluded that services.exe opens a handle to itself in an exclusive way as one of its first actions after being started. This way, other programs (including on-demand virus scanners running later) cannot read the infected file. This reasoning is confirmed by Michael St. Neitzel's very detailed virus description[27] of Sober Y.

TTAnalyze Run 2

As stated in the last section, the Sober Y executable copies itself to the C:\WINDOWS.0\WinSecurity directory under the name services.exe. Of course, we also had TTAnalyze analyze this process. This test run differs from the last one in two points:

1. We analyze services.exe that differs in one byte from the original file.

2. We explicitly upload services.exe to the directory C:\WINDOWS.0\WinSecurity instead of saving it in the default location [2].

The results are shown in the following.

- Created Files:
  - C:\WINDOWS.0\WinSecurity\mssock1.dli
  - C:\WINDOWS.0\WinSecurity\socket1.ifo
  - C:\WINDOWS.0\system32\bbvmwxxf.hml
  - C:\WINDOWS.0\system32\filesms.fms
  - C:\WINDOWS.0\system32\langeinf.lin
  - C:\WINDOWS.0\system32\nonrunso.ber
  - C:\WINDOWS.0\system32\rubezahl.rub
  - C:\WINDOWS.0\system32\runstop.rst
- Read Files (Excerpt):
  - C:\WINDOWS.0\WinSecurity\services.exe
  - C:\WINDOWS.0\system32\MSVBVM60.DLL
  - C:\WINDOWS.0\DtcInstall.log
  - C:\WINDOWS.0\FaxSetup.log
  - C:\WINDOWS.0\Fonts\desktop.ini
  - C:\WINDOWS.0\Help\access.hlp

We do not show the complete list of read files because it is very long. We can see, however, that the process reads all files that have a certain file extension such as ".ini" and ".txt".

[2]The default location is C:\InsideTM at the time of writing this chapter.

Kasperky's Virus Description

Kaspersky's virus description, which can be found at
http://www.viruslist.com/en/viruses/encyclopedia?virusid=99827, states the following.

"When installing, the worm creates a folder named 'WinSecurity' in the Windows root directory. It copies itself to this folder 3 times under the following names:"

```
%Windir%\WinSecurity\csrss.exe
%Windir%\WinSecurity\services.exe
%Windir%\WinSecurity\smss.exe
```

"The worm also creates the following files in the same folder:"

```
%Windir%\WinSecurity\mssock1.dli
%Windir%\WinSecurity\mssock2.dli
%Windir%\WinSecurity\mssock3.dli
%Windir%\WinSecurity\winmem1.ory
%Windir%\WinSecurity\winmem2.ory
%Windir%\WinSecurity\winmem3.ory
```

"Email addresses harvested from the victim machine will be saved in these files."

"The worm then registers itself in the system registry, ensuring that it will be launched each time Windows is rebooted on the victim machine:"

```
[HKLM\Software\Microsoft\Windows\CurrentVersion\Run]
"Windows" = "%Windir%\WinSecurity\services.exe"
[HKCU\Software\Microsoft\Windows\CurrentVersion\Run]
"_Windows" = "%Windir%\WinSecurity\services.exe"
```

"The worm also creates copies of itself in base64. The copies have the following names:"

```
%Windir%\WinSecurity\socket1.ifo
%Windir%\WinSecurity\socket2.ifo
%Windir%\WinSecurity\socket3.ifo
```

"The worm also creates empty files in the Windows system directory. The empty files have the following names:"

```
%System%\bbvmwxxf.hml
%System%\filesms.fms
%System%\langeinf.lin
%System%\nonrunso.ber
%System%\rubezahl.rub
%System%\runstop.rst
```

If one compares Kaspersky's virus description to TTAnalyze's report, one sees that their list of created files matches. Kaspersky's description is brief and does not mention that some of the files are only created after the worm has copied itself to the Windows-\WinSecurity directory and is started from there. The registry modifications as specified in Kaspersky's virus description are not found by TTAnalyze for the already stated reasons. Of course, we are only showing parts of Kaspersky's virus description here. In particular, the complete virus description also covers the text and subject of emails sent by the worm.

Chapter 6

Conclusion and Future Work

TTAnalyze is a prototype implementation. We believe that it clearly shows that the idea of running an executable in an emulated environment to monitor and analyze its execution works and is worth of further research. Of course, many more features would be possible, but were not realized because of time constraints. For example, the biggest weakness at the moment is the lack of a detailed network analysis. Many malware samples make use of email for replication, many malware samples connect to IRC servers or use NTP servers in order to get the current time. Hence, a network analysis would certainly be useful and it is one of the first features that will be implemented in the future.

Implementing a network analysis raises the problem that the emulation of one computer (PC) is not sufficient any longer. Our analysis is primarily based on monitoring the test-subject's local execution. When extending this analysis to take into account network connections, the test-subject has to be provided with a network (maybe even the Internet) so that we can monitor its network activities. We can provide the test-subject with this network by either emulating a network or by using a real one. Giving it access to a real network is too dangerous, but we can easily emulate a network by having fake HTTP, FTP, NTP, DNS, or IRC servers answer the test-subject's requests. In case the test-subject downloads other malicious components from the Internet, the URL can be extracted this way, but a human operator will have to manually download the component.

During the course of testing TTAnalyze with real malware samples, it became apparent that dynamic analysis alone might not be the perfect way to analyze unknown executables. Admittedly, TTAnalyze does more than only analyzing the test-subject's execution. It also changes the test-subject's execution (function call insertion and page fault handling) in certain situation. However, we aim to extend this feature and change the execution of the program in even more situations. Our goal is to watch several different execution paths of the program, for example, by changing the truth value for conditional jumps and monitoring both resulting execution paths. This would allow us to also obtain a more complete picture of the behavior of the executable in different environments, with different inputs, or under special circumstances (e.g., the executable is run at a certain day of the year). This brings us a step closer to the concept of static analysis. Static analysis does not only analyze a single execution trace, but the complete program. Building on

control flow graphs and data flow graphs, it is possible to analyze all execution paths that a program can ever follow. If it were not for disadvantages such as its inability to analyze exe-packed binaries and self-modifying programs, one could believe that static analysis is the perfect way for analyzing unknown binaries. The truth is that both approaches have their weakness. Together, however, they can complement each other and deliver the most precise analysis results.

Bibliography

[1] Apache Ant-Contrib Tasks. http://ant-contrib.sourceforge.net, 2005.

[2] Latex Task for Apache Ant. http://www.dokutransdata.de/ant_latex/index.html, 2005.

[3] Apache Ant. http://ant.apache.org/, 2005.

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[5] Fabrice Bellard. Qemu. http://fabrice.bellard.free.fr/qemu/, 2005.

[6] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference*, 2005.

[7] Boost library. http://www.boost.org/, 2005.

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

[9] Cygwin. http://www.cygwin.com, 2005.

[10] Daemon Tools. http://www.daemon-tools.cc/, 2005.

[11] David Harley et al. *Das Anti-Viren Buch*. mitp-Verlag, 2002.

[12] Kevin Lawton et al. Bochs. http://bochs.sourceforge.net/, 2005.

[13] F-Secure Virus Glossary. http://www.f-secure.com/virus-info/glossary.shtml, 2005.

[14] IDA Pro. http://www.datarescue.com/, 2005.

[15] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual Volume 3:System Programming Guide*, 2005.

[16] Sun Java. http://java.sun.com, 2005.

[17] H. A. Lichstein. When should you emulate? *Datamation*, 15:205–210, 1969.

[18] Efrem G. Mallach. On the relationship between virtual machines and emulators. In *Proceedings of the workshop on virtual computer systems*, pages 117–126, 1973.

[19] Windows Device Driver Kit 2003. http://www.microsoft.com/whdc/devtools/ddk/, 2005.

[20] MS Debugging Tools. http://www.microsoft.com/whdc/devtools/debugging/, 2005.

[21] Microsoft IFS KIT. http://www.microsoft.com/whdc/devtools/ifskit, 2005.

[22] Microsoft Platform SDK. http://www.microsoft.com/msdownload/platformsdk/, 2005.

[23] Microsoft Virtual PC. http://www.microsoft.com/windows/virtualpc/, 2005.

[24] Miktex. http://www.miktex.org/, 2005.

[25] Mingw. http://www.mingw.org, 2005.

[26] Gary Nebbett. *Windows NT/2000 Native API Reference*. New Riders Publishing, Thousand Oaks, CA, USA, 2000.

[27] Michael St. Neitzel. Win32/sober.y. http://www.eset.com/msgs/sobery.htm, 2005.

[28] Terence Parr. Antlr. http://www.antlr.org/, 2005.

[29] Pearpc. http://pearpc.sourceforge.net/, 2005.

[30] Matt Pietrek. A crash course on the depths of win32 structured exception handling. *Microsoft Systems Journal*, January 1997.

[31] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974.

[32] Mark Probst. Dynamic Binary Translation, UKUUG Linux Developers' Conference, 2002.

[33] J. Robin and C. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor, 2000.

[34] Mendel Rosenblum. The reincarnation of virtual machines. *Queue*, 2(5):34–40, 2004.

[35] Mark Russinovich. Sysinternal's Process-Explorer Tool for Windows Version 9.2. http://www.sysinternals.com/Utilities/ProcessExplorer.html, 2005.

[36] Mark Russinovich and Bryce Cogswell. Sysinternal's Filemon Tool for Windows. http://www.sysinternals.com/Utilities/Filemon.html, 2005.

[37] Mark Russinovich and Bryce Cogswell. Sysinternal's Regmon Tool for Windows. http://www.sysinternals.com/Utilities/Regmon.html, 2005.

[38] Mark E. Russinovich and David A. Solomon. *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server(TM) 2003, Windows XP, and Windows 2000 (Pro-Developer).* Microsoft Press, Redmond, WA, USA, 2004.

[39] SDL - Simple DirectMedia Layer. http://www.libsdl.org, 2005.

[40] Symantec Glossary. http://www.symantec.com/avcenter/glossary, 2005.

[41] Tcpdump, A Network sniffer. http://www.tcpdump.org, 2005.

[42] Upx. http://www.upx.org/, 2005.

[43] Virus bulletin june, 2005.

[44] Vmware. http://www.vmware.com/, 2005.

[45] VMWare and VPC detection: Detect if your program is running inside a Virtual Machine. http://www.codeproject.com/system/VmDetect.asp, 2005.

[46] Wine. http://www.winehq.org/, 2005.

[47] James Yonan. Openvpn. http://openvpn.net/, 2005.

[48] Oleh Yuschuk. OllyDbg. http://www.ollydbg.de/, 2005.

[49] ZLib - Compression Library, 2005.

Appendix A

External Dependencies

As every real-word program TTAnalyze makes use of several external tools and libraries. In this chapter, we try to systematically list and describe all these dependencies.

A.1 Runtime Dependencies

TTAnalyze is distributed in the form of a DVD image file that contains the TTAnalyze executable itself, dependent DLLs, files required by QEMU, the hard-disk image, the system snapshot, a template configuration file and a couple of other resources.

The only external dependency that is needed besides the zip-file is the TAP device driver that is part of the Windows distribution of OpenVPN [47]. OpenVPN is a full-featured open-source SSL VPN solution that is developed by James Yonan. A TAP device is a virtual ethernet card. It is a piece of software that imitates an ethernet card to Windows in the same way as the more popular daemon tools package [10] emulates a CD/DVD driver to Windows.

TTAnalyze uses the TAP device to communicate with QEMU's network card. After installing the driver it is necessary to configure it correctly. Therefore assign a static IP, which lies in the same subnet as QEMU's network card, to the TAP device. As a last step the name of the TAP device has to be specified in TTAnalyze's configuration file.

TTAnalyze is known to work with the TAP device that is part of OpenVPN version 2.0rc19.

A.2 Compiletime Dependencies

Before you can compile TTAnalyze, you have to satisfy the following list of requirements. In particular, setting up the MySYS environment, which is necessary for compiling QEMU, is a tedious task.

- Boost library [7] version 1.33

Boost is a free, portable C++ library, which contains a lot of functionality and works well with the C++ Standard Library.

- MinGW/MySYS environment for QEMU.
  - MinGW[25] - A GCC port for Win32.
  - MySYS[25] - A unix command line environment for Windows. Includes a port of the bash shell.
  - MySysDTK[25] - Adds more command line tools to the MySYS package.
  - SDL library[39] version 1.2.8 or 1.2.9
  - zlib library[49] version 1.2.2
- Apache Ant [3] version 1.6.5

  Apache Ant is an alternative to make that is especially popular in the Java community. TTAnalyze uses Ant for its build-system. Besides Ant itself, you will also need the following Ant extensions:

  - ant-contrib [1]

    The Ant-Contrib project is a collection of tasks (and at one point maybe types and other tools) for Apache Ant.
  - ant-latex [2]

    A Latex task for Apache Ant.
- ANTLR [28] version 2.75

  ANTLR, ANother Tool for Language Recognition, (formerly PCCTS) is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing Java, C#, C++, or Python actions. The Generator uses ANTLR for parsing a file that contains the declarations of all the functions that are hooked in TTAnalyze. The language understood by the Generator is a subset of C with a couple of extensions.
- Windows 2003 DDK [19]

  The Windows 2003 Device Driver Kit contains all the tools and libraries necessary to compile a kernel-level driver for Windows. Although the DDK has 2003 in its name it can be used to write drivers for Windows XP too. The InsideTM driver requires this dependency.
- Microsoft Visual Studio 2005

TTAnalyze uses Visual Studio 2005 project files and maybe depends on other features only present in this version of Visual Studio.

- Java [16] version 1.5.0

  Java is needed for running Ant and ANTLR.

- MiKTeX [24] version 2.4

  MiKTeX is a LaTeXdistribution for Windows. The documentation is written in LaTeX. If you want to generate the documentation, you will need MiKTex. (Other LaTeXdistributions probably work too.)

Appendix B

Command Line Parameters of TTAnalyze

This appendix lists all allowed command line parameters of TTAnalyze.

-h [−−**help**] produce a help message

-v [−−**version**] output the version number

−−**config-file-options** prints all options that are allowed in a config-file

−−**config-file arg** use config-file instead of the default TTAnalyze.cfg

−−**debug** calls a debugger shortly after program start.

−−**use-image arg** determines which image and snapshot files are going to be used.

−−**host-ip arg** [MultipleValues] set all valid IP addresses for this computer.

−−**qemu-ip arg** set the IP address of the NIC that QEMU emulates.

−−**args arg** set the command-line arguments for test-subject

−−**execute arg** set name of the executable that should be analyzed

−−**upload arg** [MultipleValues] upload files

−−**download arg** [MultipleValues] download files

−−**dll-download arg** [MultipleValues] download DLL files

−−**kill arg** [MultipleValues] kill processes which match the given exe-filenames

−−**show-qemu arg (=0)** show the QEMU-window while the virtual system is running.

−−**tap-adapter-name arg** set the name of the TAP-adapter

−−**report-only-hlf arg (=1)** report only the function calls at the highest level.

−−**timeout arg (=0)** stop the analysis after executing the test-subject x seconds in the virtual system. (0=no timeout)

−−**output-dir arg** change the name of the output-directory

−−**log-allowed-packets arg (=1)** activate logging of all packets that QEMU has sent and that were allowed to pass.

−−**log-blocked-packets arg (=1)** activate logging of all packets that QEMU has sent and that were blocked.

−−**qemu-log arg** [MultipleValues] activate QEMU logging options.
Possible values: [in_asm, out_asm, immediate]

−−**disassemble arg (=0)** activate disassembling of all program instructions.