J Comput Virol (2006) 2:67–77
DOI 10.1007/s11416-006-0012-2

ORIGINAL PAPER

Dynamic analysis of malicious code

Ulrich Bayer · Andreas Moser · Christopher Kruegel · Engin Kirda

Received: 13 January 2006 / Accepted: 27 March 2006 / Published online: 16 May 2006

Abstract Malware analysis is the process of determining the purpose and functionality of a given malware sample (such as a virus, worm, or Trojan horse). This process is a necessary step to be able to develop effective detection techniques for malicious code. In addition, it is an important prerequisite for the development of removal tools that can thoroughly delete malware from an infected machine. Traditionally, malware analysis has been a manual process that is tedious and time-intensive. Unfortunately, the number of samples that need to be analyzed by security vendors on a daily basis is constantly increasing. This clearly reveals the need for tools that automate and simplify parts of the analysis process. In this paper, we present TTAnalyze, a tool for dynamically analyzing the behavior of Windows executables. To this end, the binary is run in an emulated operating system environment and its (security-relevant) actions are monitored. In particular, we record the Windows native system calls and Windows API functions that the program invokes. One important feature of our system is that it does not modify the program that it executes (e.g., through API call hooking or breakpoints), making it more difficult to detect by malicious code. Also, our tool runs binaries in an unmodified Windows environment, which leads to excellent emulation accuracy. These factors make TTAnalyze an ideal tool for quickly understanding the behavior of an unknown malware.

Keywords Malware · Analysis · API · Virus worm · Static analysis · Dynamic analysis

U. Bayer (✉)
Ikarus Software,
Fillgradergasse 7, 1060, Vienna, Austria
e-mail: ulli@seclab.tuwien.ac.at

A. Moser · C. Kruegel · E. Kirda
Secure Systems Lab,
Technical University Vienna,
Vienna, Austria
e-mail: andy@seclab.tuwien.ac.at

C. Kruegel
e-mail: chris@seclab.tuwien.ac.at

E. Kirda
e-mail: ek@seclab.tuwien.ac.at

1 Introduction

Malware is a generic term to denote all kinds of unwanted software (e.g., viruses, worms, or Trojan horses). Such software poses a major security threat to computer users. According to estimates, the financial loss caused by malware has been as high as 14.2 billion US dollars in the year 2005 [5]. Unfortunately, the problem of malicious code is likely to grow in the future as malware writing is quickly turning into a profitable business [21]. Malware authors can sell their creations to miscreants, who use the malicious code to compromise large numbers of machines that can then be abused as platforms to launch denial-of-service attacks or as spam relays. Another indication of the significance of the problem is that even people without any special interest in computers are aware of worms such as Nimda or Sasser. This is because security incidents affect millions of users and regularly make the headlines of mainstream news.

The most important line of defence against malicious code are virus scanners. These scanners typically rely on a database of descriptions, or signatures, that characterize known malware instances. Whenever an unknown malware sample is found in the wild, it is usually necessary to update the signature database accordingly so that the novel malware piece can be detected by the scan engine. To this end, it is of paramount importance to be able to quickly analyze an unknown malware sample and understand its behavior and effect on the system. In addition, the knowledge about the functionality of malware is important for removal. That is, to be able to cleanly remove a piece of malware from an infected machine, it is usually not enough to delete the binary itself. It is also necessary to remove the residues left behind by the malicious code (such as unwanted registry entries, services, or processes) and undo changes made to legitimate files. All these actions require a detailed understanding of the malicious code and its behavior.

The traditional approach to analyze the behavior of an unknown program is to execute the binary in a restricted environment and observe its actions. The restricted environment is often a debugger, used by a human analyst to step through the code in order to understand its functionality. Unfortunately, anti-virus companies receive up to *several hundred* new malware samples each day. Clearly, the analysis of these malware samples cannot be performed completely manually. Hence, the necessity of automated solutions.

One way to automate the analysis process is to execute the binary in a virtual machine or a simulated operating system environment. While the program is running, its interaction with the operating system[1] (e.g., the native system calls or Windows API calls it invokes) can be recorded and later presented to an analyst. This approach relieves a human analyst from the tedious task of having to manually go through each single malware sample that is received. Of course, it might still be the case that human analysis is desirable after the automatic process. However, the initial results at least provide details about the program's actions that then help to guide the analyst's search.

Current approaches for automatic analysis suffer from a number of shortcomings. One problem is that malicious code is often equipped with detection routines that check for the presence of a virtual machine or a simulated OS environment. When such an environment is detected, the malware modifies its behavior and the analysis delivers incorrect results. Malware also checks for software (and even hardware) breakpoints to detect if the program is run in a debugger. This requires that the analysis environment is *invisible* to the malicious code. Another problem is when the analysis environment does not monitor the complete interaction with the system. When this happens, the malicious code could evade analysis. This might be possible because there exists thousands of Windows API calls, often with arguments that are composed of complex data structures. Furthermore, the malicious code could also interact directly with the operating system via native system calls. Thus, the analysis environment has to be *comprehensive* and cover all aspects of the interaction of a program with its environment.

[1] Because the vast majority of malware is written for Microsoft Windows, the following discussion considers only this operating system.

In this paper, we describe TTAnalyze, a tool that automates the process of analyzing malware to allow a human analyst to quickly obtain a basic understanding of the actions of an unknown executable. Running a binary under TTAnalyze results in the generation of a report that contains information to give the human analyst a very good impression about the purpose and the functionality of the analyzed sample. This report includes detailed data about modifications made to the Windows registry and to the file system, information about interactions with the Windows Service Manager and other processes, as well as a complete log of all generated network traffic.
The following list summarizes the key features of TTAnalyze:

- TTAnalyze uses emulation to run the unknown binary together with a complete operating system in software. Thus, the malware is never executed directly on the processor. Unlike solutions that use virtual machines, debuggers, or API function hooking, the presence of TTAnalyze is practically invisible to malicious code.
- The analysis is comprehensive because our system monitors calls to native kernel functions as well as calls to Windows API functions. It also provides support for the analysis of complex function call arguments that contain pointers to other objects.
- TTAnalyze can perform function call injection. Function call injection allows us to alter the execution of the program under analysis and run our code in its context. This ability is required in certain cases to make the analysis more precise.

The remainder of this paper is structured as follows. In section 2, we present static analysis techniques for malicious code and point out their inherent weaknesses in order to motivate the use of dynamic approaches. Then, section 3 introduces related work in the field of dynamic malware analysis. In section 4, we discuss the design and implementation details of our proposed system. Section 5 provides an experimental evaluation of its effectiveness. Finally, section 6 briefly concludes and outlines future work.

2 Static analysis techniques

Analyzing unknown executables is not a new problem. Consequently, many solutions already exist. These solutions can be divided into two broad categories: *static analysis* and *dynamic analysis* techniques. In this section, we discuss static code analysis techniques and point out inherent limitations that make the use of dynamic approaches appealing. In the following section 3, we survey related work in the area of dynamic malware analysis and present advantages of our system compared to previous ones.

Static analysis is the process of analyzing a program's code without actually executing it. In this process, a binary is usually disassembled first, which denotes the process of transforming the binary code into corresponding assembler instructions. Then, both control flow and data flow analysis techniques can be employed to draw conclusions about the functionality of the program.

A number of static binary analysis techniques [2, 3, 8] have been introduced to detect different types of malware. Static analysis has the advantage that it can cover the complete program code and is usually faster than its dynamic counterpart. However, a general problem with static analysis is that many interesting questions that one can ask about a program and its properties are undecidable in the general case. Of course, there exists a rich body of work on static analysis techniques that demonstrate that many problems can be approximated well in practice, often because difficult-to-handle situations occur rarely in real-world software. Unfortunately, the situation is different when dealing with malware. Because malicious code is written directly by the adversary, it can be crafted deliberately so that it is hard to analyze. In particular, the attacker can make use of binary obfuscation techniques to thwart both the disassembly and code analysis steps of static analysis approaches.

The term *obfuscation* refers to techniques that preserve the program's semantics and functionality while at the same time making it more difficult for the analyst to extract and comprehend the program's structures. In the context of disassembly, obfuscation refers to transformations of the binary such that the parsing of instructions becomes difficult. In [9], Linn and Debray introduced novel obfuscation techniques that exploit the fact that the Intel x86 instruction set architecture contains variable length instructions that can start at arbitrary memory address. By inserting padding bytes at locations that cannot be reached during run-time, disassemblers can be confused to misinterpret large parts of the binary. Although their approach is limited to Intel x86 binaries, the obfuscation results against current state-of-the-art disassemblers are remarkable.

Besides obfuscation techniques to increase the difficulty of the disassembly process, the code itself can be obfuscated to make it difficult to extract the control flow of a program or to perform data flow analysis. The basic idea for such obfuscation techniques is that they can be automatically applied, but not easily undone, even if the transformation approach is known. This requirement is similar to the one that inspired the "one-way translation process" introduced in [25], or cryptography. In both cases, a process is suggested that is easy to perform in one direction, but difficult to revert.

One possibility to realize such obfuscation is the use of a primitive called *opaque constants*. Opaque constants are an extension to the idea of opaque predicates, which are defined in [4] as "boolean valued expressions whose values are known to the obfuscator but difficult to determine for an automatic deobfuscator." The difference between opaque constants and opaque predicates is that opaque constants are not boolean, but integer values. More precisely, opaque constants are mechanisms to load a constant into a processor register whose value cannot be determined statically.

Based on opaque constants, one can then build a number of obfuscation transformations that are difficult to analyze statically. For example, one can replace the target of direct control transfer instructions (such as jumps or calls) with indirect variants that use opaque constants as jump targets. Another area of application of opaque constants is data location and data usage obfuscation. The location of a data element is often specified by providing a constant, absolute address or a constant offset relative to a particular register. In both cases, the task of a static analyzer can be complicated if the actual data element that is accessed is hidden. With data usage obfuscation, the tracking of values in registers is complicated by the fact that register content is frequently spilled to, and reloaded from, unknown locations.

Finally, the code that is analyzed by a static analyzer may not necessarily be the code that is actually run. In particular, this is true for self-modifying programs that use polymorphic [22, 26] and metamorphic [22] techniques and packed executables that unpack themselves during run-time [16].

3 Dynamic analysis techniques

In contrast to static techniques, dynamic techniques analyze the code during run-time. While these techniques are non-exhaustive, they have the significant advantage that only those instructions are analyzed that the code actually executes. Thus, dynamic analysis is immune to obfuscation attempts and has no problems with self-modifying programs. When using dynamic analysis techniques, the question arises in which environment the sample should be executed. Of course, running malware directly on the analyst's computer, which is probably connected to the Internet, could be disastrous as the malicious code could easily escape and infect other machines. Furthermore, the use of a dedicated standalone machine that is reinstalled after each dynamic test run is not an efficient solution because of the overhead that is involved.

Running the executable in a virtual machine (that is, a virtualized computer), such as one provided by VMware [24], is a popular choice. In this case, the malware can only affect the virtual PC and not the real one. After performing a dynamic analysis run, the infected hard disk image is simply discarded and replaced by a clean one (i.e., so called *snapshots*). Virtualization solutions are sufficiently fast. There is almost no difference to running the executable on the real computer, and restoring a clean image is much faster than installing the operating system on a real machine. Unfortunately, a significant drawback is that the executable to be analyzed may determine that it is running in a virtualized machine and, as a result, modify its behavior. In fact, a number of different mechanisms have been published [17,20] that explain how a program can detect if it is run inside a virtual machine. Of course, these mechanisms are also available for use by malware authors.

A PC emulator is a piece of software that emulates a personal computer (PC), including its processor, graphic card, hard disk, and other resources, with the purpose of running an unmodified operating system. It is important to differentiate emulators from virtual machines such as VMware. 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 PC emulators, which simulate all instructions in software. Because all instructions are emulated in software, the system can appear exactly like a real machine to a program that is executed, yet keep complete control. Thus, it is more difficult for a program to detect that it is executed inside a PC emulator than in a virtualized environment. This is the reason why we decided to implement TTAnalyze based on a PC emulator.

Note that there is one observable difference between an emulated and a real system, namely speed of execution. This fact could be exploited by malicious code that relies on timing information to detect an emulated environment. While it would be possible for the emulator to provide incorrect clock readings to make the system appear faster for processes that attempt to time execution speed, this issue is currently not addressed by TTAnalyze.

In addition to differentiating the type of environment used for dynamic analysis, one can also distinguish and classify different types of information that can be captured during the analysis process. Many systems focus on the interaction between an application and the operating system and intercept system calls or hook Windows API calls. For example, a set of tools provided by Sysinternals [18] allows the analyst to list all running Windows processes (similar to the Windows Task Manager), or to log all Windows registry and file system activity. These tools are implemented as operating system drivers that intercept native Windows system calls. As a result, they are invisible to the application that is being analyzed. They cannot, however, intercept and analyze Windows API calls or other user functions. On the other hand, tools [6] exist that can intercept arbitrary user functions, including all Windows API calls. This is typically realized by rewriting target function images. The original function is preserved as a subroutine and callable through a trampoline. Unfortunately, the fact that the code needs to be modified can be detected by malicious code that implements integrity checking.

TTAnalyze uses a PC emulator and thus has complete control over the sample program. It can intercept and analyze both native Windows operating system calls as well as Windows API calls while being invisible to malicious code. The complete control offered by a PC emulator potentially allows the analysis that is performed to be even more fine-grain. Similar to the functionality typically provided by a debugger, the code under analysis can be stopped at any point during its execution and the process state (i.e., registers and virtual address space) can be examined. Unlike a debugger, however, our system does not have to resort to breakpoints, which are known to cause problems when used for malicious code analysis [23]. The reason is that software breakpoints directly modify the executable and thus can be detected by code integrity checks. Also, malicious code was found in the wild that used processor debug registers for its computations, thereby breaking hardware breakpoints.

4 System description

TTAnalyze is a tool for analyzing Windows executables (more precisely, files conforming to the portable executable (PE) file format [12]). To this end, the program under analysis is executed inside a PC emulation environment and relevant Windows API and native system calls are logged. In the following sections, we describe in more detail the design and implementation of key components of TTAnalyze.

4.1 Emulation environment

As mentioned previously, TTAnalyze uses a PC emulator to execute unknown programs. When designing our system, we had to choose between different forms of emulation. In particular, we had to decide if the hardware of a complete PC should be emulated so that an actual off-the-shelf operating system could be installed, or if the processor should be emulated and our own implementation of (a subset of) the operating system interface should be provided. Virus scanners typically emulate the processor and provide a lightweight implementation of the operating system interface (both native system calls and Windows API calls). This approach allows a very efficient analysis process. Unfortunately, it is not trivial to make the operating system stub behave exactly like the actual operating system, and the semantics between a real system and the simulated one differs in many cases. These differences could be detected by malware, or simply break the code. Thus, we decided to emulate an entire PC computer system, running an off-the-shelf Windows XP on top. While the analysis is significantly slower compared to a virus scanner, the accuracy of the emulation is excellent. Since our focus is on the analysis of the behavior of the binary, this trade-off is acceptable.

TTAnalyze uses Qemu [1], an open-source PC emulator written by Fabrice Bellard, as its emulator component. Qemu is a fast PC emulator that properly handles self-modifying code. To achieve high execution speed, Qemu employs an emulation technique called *dynamic translation*. Dynamic translation works in terms of basic blocks, where 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. The idea is to first translate a basic block, then execute it, and finally translate the next basic block (if a translation of this block is not already available). The reason is that it is more efficient to translate several instructions at once rather than only a single one.

Of course, Qemu could not be used in our system without modification. First, it had to be transformed from a stand-alone executable into a Windows shared library (DLL), whose exported functions can be used by TTAnalyze. Second, Qemu's translation process was modified so that a callback routine into our analysis framework is invoked before every basic block that is executed on the virtual processor. This allows us to tightly monitor the process under analysis.

Before a dynamic analysis run is performed, the modified PC emulator boots from a virtual hard disk, which has Windows XP (with Service Pack 2) installed. The lengthy Windows boot-process is avoided by starting Qemu from a snapshot file, which represents the state of the PC system after the operating system has started.

4.2 Analysis process

The analysis process is started by executing the (malware) program in the emulated Windows environment and monitoring its actions. In particular, the analysis focuses on which operating system services are requested by the binary (i.e., which system calls are invoked). Every action that involves communication with the environment (e.g., accessing the file system, sending a packet over the network, or launching another program) requires a Windows user mode process to make use of an appropriate operating system service. There is no way for a process to directly interact with a physical device, which also includes physical memory. The reason for this stems from the design of modern operating systems, which prohibit direct hardware access so that multiple processes can run concurrently without interfering with each other. Thus, it is reasonable to monitor the system services that a process requests in order to analyze its behavior.

On Microsoft Windows platforms, monitoring system service requests is not entirely straightforward. The reason is that the actual operating system call interface, called native API interface, is mostly undocumented and not meant to be used directly by applications. Instead, applications are supposed to call functions of the documented Windows API.[2] The Windows API is a large collection of user mode library routines, which in turn invoke native API functions when necessary. The idea is that the Windows API adds a layer of indirection to shield applications from changes and subtle complexities in the native API. In particular, the native API may change between different Windows versions and even between different service pack releases. On a Windows system, the native API is provided by the system file `ntdll.dll`. Parts of this interface are documented by Microsoft in the Windows DDK [10] and the Windows IFS kit [11]. Moreover, Gery Nebbett has written an unofficial documentation of the native API [14], which covers about 90% of the functions.

[2] The Windows API is documented by Microsoft in the Platform SDK [13].

Malware authors sometimes use the native API directly to avoid DLL dependencies or to confuse virus scanner's operating system simulations. For this reason, TTAnalyze monitors both the Windows API function calls of an application and also its native API function calls. The task of monitoring which operating system services are invoked by the program requires solving two problems:

1. We must be able to precisely track the execution of the malware process and distinguish between instructions executed on behalf of the malware process and those of other processes. This is essential because the virtual processor does not only run the malware process, 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 processor instruction whether or not either instruction belongs to the malware process.
2. We need an unobtrusive way for monitoring the accessed operating system services. That is, we have to be able to determine that a native API call or a Windows API call is invoked *without* modifying the malware code. That is, we cannot hook API functions or set debug breakpoints.

We accomplish the precise tracking of the malware process with the help of the `CR3` processor register. The `CR3` register, which is also known as the page-directory base register (PDBR), contains the physical address of the base of the page directory for the current process. The processor uses the page directory when it translates virtual addresses to physical addresses. More precisely, to determine the location of the page directory when performing memory accesses, the processor makes use of the `CR3` register.

Windows assigns each process its own unique page directory. This protects processes (in particular, their virtual memory address space) from each other by ensuring that each process has its own virtual memory space. The page directory address of the currently running process has to be stored in the `CR3` processor register. Consequently, Windows loads the `CR3` register on every context switch. Thus, we simply have to determine which page directory address has been assigned to the malware process by Windows. Then, we are able to efficiently determine whether or not the current instruction belongs to the test subject under analysis by comparing the current value of the `CR3` register to the page directory address of this test subject.

Determining the physical address of the page directory of the test subject is the responsibility of a probe component that is located *inside* the emulated Windows XP environment. This probe serves as a sensor in the emulated environment and consists of a kernel driver and a program that is run in user mode. The task of the kernel driver is to locate the page directory address that belongs to the test subject and report its findings back to the user mode process. The user mode component then informs TTAnalyze. Note that TTAnalyze is outside the emulated environment, thus communication between the probe and TTAnalyze has to take place over the virtual network that connects the emulated environment with its host system. To this end, an RPC server is used that runs inside the emulated PC.

The kernel driver is the most straightforward way to access the page directory address, which is stored in a memory region that is only accessible to the Windows NT kernel and its device drivers. More precisely, the page directory address can be found as an attribute of that `EPROCESS` structure that corresponds to the test subject. The `EPROCESS` structure is a Windows-internal data object that plays a key role in the way Windows manages processes. For each process in the system, a corresponding `EPROCESS` structure exists. Thus, the device driver has to walk the list of system processes (which consists of `EPROCESS` members) until it finds the one corresponding to the process of the test subject. At this point, the appropriate page directory address can be read. Note that the page table address of the test subject's process has to be obtained *before* its first instruction is executed. To this end, the process is created in a suspended state. Only after successfully identifying the page directory address is the test subject allowed to run.

As mentioned previously, the second problem of our analysis is to monitor the invocation of operating system functions.[3] This task can be solved by comparing the current value of the virtual processor's instruction pointer (or program counter) register to the start addresses of all operating system functions that are under surveillance. This comparison is performed in the callback routine of TTAnalyze, which Qemu invokes at the start of each translation block. Note that the start address of a function always corresponds to the first instruction in a translation block. The reason is that a function call is a control transfer instruction, and whenever a control transfer instruction is encountered, Qemu starts a new translation block. At this point, TTAnalyze is invoked and can check the current value of the program counter.

A Windows application typically accesses operating system functions by dynamically linking to system DLLs and calling their exported functions. Thus, we can extract the addresses of interesting functions simply from library export tables. For example, an application calls the Windows API function `CreateFile`, which is implemented in the shared library `Kernel32.dll` when it wants to create a file. In this case, determining the start address of `CreateFile` is easily possible by looking at the corresponding entry in `Kernel32.dll`'s export table (and then adding the base address of `Kernel32.dll` to it, as DLLs may be loaded at a different base address).

4.3 Function arguments

Using the system described in the previous sections, we are in a position of knowing which operating system functions are used by an application. For example, if an application invokes `CreateFile`, we know that a file was created. Unfortunately, we do not have any more details (e.g., the name of the created file). Obviously, we can improve the situation by analyzing the arguments of operating system function calls. To this end, we have extended our analysis framework with the capability to automatically invoke user-specified callback routines in TTAnalyze whenever the test subject calls one of the monitored operating system functions. For each callback routine, the analyst can specify code to process or log the arguments of the corresponding operating system function. For example, if the test subject calls the `CreateFile` function, a TTAnalyze callback routine is invoked where one can access the argument that specifies the name of the file to be created.

To be able to access an argument value of an operating system function, the callback routine has to first read it from the emulated, or virtual system by specifying its memory address and size. To see this, recall that the TTAnalyze callback routine is running in a different memory address space than the process under analysis. Thus, the writer of a callback routine has to know the size and structure of all function arguments. Reading function call arguments in this fashion would be tedious and error-prone, certainly reducing the number of callback functions. To address this problem, we desire a mechanism to automatically generate the required code for reading the values of function arguments from the virtual system. The goal is to have the parameter list of a callback routine mirror the parameter list of its corresponding operating system function. Whenever the callback routine is invoked, all function argument values are automatically extracted from the virtual system and then correctly copied into the arguments of the callback routine. In this fashion, the author of a

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

callback function can access the arguments of an operating system function call by simply reading the arguments of the callback routine.

To achieve the goal of generating the necessary C++ source code for reading the arguments of a function call from the virtual system, we developed the *generator* component. This component is a stand-alone program that can be run independently of TTAnalyze. Its task is to generate the desired callback routine stubs (or more precisely, stubs that include the code to handle the arguments). The generator component requires as input a file containing the declarations of all monitored operating system functions. By parsing the function declarations, the generator is able to determine the sizes and structures of function arguments and can subsequently generate the appropriate C++ code for reading them.

The grammar for the generator's input file resembles the grammar of the C programming language. The difference is that our grammar only supports declarations and no statements. Moreover, we have slightly extended the C-syntax in two ways.

1. Parameter declarations of functions may include the keywords `[out]`, `[in]`, or `[inout]`. These keywords are 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 system function call is invoked, while `out` parameters are read when the function returns. If a direction specification is missing, `in` is assumed by default.
2. Array declarations of the form `[ARGx_B]` or `[ARGx_U]` are possible. Such declarations indicate that the variable in front of `[]` is a dynamic array, and that the size of this array is specified by another function argument. The position of this size-specifying argument in the function parameter list is indicated by the value of `x`. Thus, `x` represents an integer value larger than zero. The postfix `_B` further specifies that the size is given in bytes, while the postfix `_U` states that the size is given in units of the array base type. The special form `[NT]` is used for a null-terminated byte array (e.g., C strings are treated as null-terminated byte arrays).
   The reason for having to annotate array arguments is that TTAnalyze has to know how to determine the number of elements of an array during run-time in order to copy the right amount of data to the callback routine. To this end, TTAnalyze can either be told about an 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. As an example, consider the function `int main(int argc,char *argv[])`. This function should be declared as `int main(int argc, char argv[NT][ARG1_U]` in our header file.

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

There is another problem that we have to deal with when reading the values of function arguments from the virtual system. Unfortunately, it is not always immediately possible to read from the virtual address space of a process in the emulated system. To understand this problem, consider that the physical main memory of the emulated PC system simply is a large malloc'ed memory block on the host system. Thus, TTAnalyze can always read from the emulated main memory when supplying a physical address. When supplying a virtual address in the context of the emulated system, however, this virtual address has to be converted into a physical address first. Unfortunately, the possibility exists that the content referred to by this virtual address is not present in the emulated physical memory, but only on the emulated hard disk (i.e., the content is currently paged out). In this case, reading from the virtual system's memory would result in an error. There are also other cases where one is not able to directly retrieve the content for a virtual address. The Windows MMU (memory management unit) uses lazy evaluation as often as possible to save resources [19]. *Lazy evaluation* means to wait 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 only when the processor first references memory in the shared library.

Failing to read an argument of an operating system function call would be a serious drawback. Thus, TTAnalyze must be able to read the memory contents at any specified virtual address. To solve the problem of memory content that is currently paged out, we can resort to the page fault handler of the emulated operating system. More precisely, whenever we wish to access an address that is not present in the emulated physical memory, we force the test subject to read from this virtual address. This read operation invokes the page fault handler of the emulated operating system, which loads the appropriate memory page into the emulated physical memory. When the handler has done its work, the desired content can be easily obtained.

4.4 Code injection

In the previous section, we mentioned the need of TTAnalyze to force the test subject to perform read operations on its behalf. To this end, TTAnalyze has to change the flow of execution of the test subject. This is achieved by *injecting* read instructions into its instruction stream. However, the ability to change the flow of execution of a program is not only useful for inserting read instructions. It can also be used to call arbitrary functions exported by a DLL (e.g., Windows API functions). This ability to insert function calls can be used to improve the quality of the analysis results in the following situations:

- *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, as `RegCreateKeyEx` can be used for both creating as well as opening a registry key.
- *File or directory*. In several situations, it is not possible to decide if a filename refers to a file or a directory from the function arguments alone.
- *Unknown handles*. TTAnalyze typically monitors all Windows API and native API function calls that return handles. As a result, TTAnalyze knows to what resources these handles refer to. However, handles might be inherited from another process or obtained via a operating system function that is not monitored. In these cases, function call insertion is required to extract information about an otherwise unknown handle.

Because TTAnalyze uses emulation to run the test subject, it is easy to insert additional instructions (such as read instructions) into Qemu's translation blocks. Also, function calls are easy to inject, as 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. The main difficulty when performing a function call in the context of the emulated process is that the arguments expected by this function need to be pushed onto the emulated stack. Pushing necessary arguments requires one to serialize and copy all arguments from the host memory into the memory of the emulated system, possibly involving complex function arguments that contain pointers to other structures. This process is the opposite of reading arguments from the emulated system into the host environment. This allows us to reuse the generator component (described in section 4.3) to automatically generate the necessary code to push the arguments onto the emulated stack.

4.5 Analysis report

TTAnalyze is a tool for analyzing malware. While, in principle, arbitrary functions can be monitored, we provide a number of callback routines that analyze and log security-relevant actions. After a run on a test sample, the recorded information is summarized in a concise report. This report contains the following information:

1. General information. This section contains information about TTAnalyze's invocation, the command line arguments, and some general information about the test subject (e.g., file size, exit code, time to perform analysis, ...).
2. File activity. This section covers the file activity of the test subject (i.e., which files were created, modified, ...).
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, information about the creation or termination of processes (and threads) as well as interprocess communication can be found.
6. Network activity. This section provides a link to a log that contains all network traffic sent or received by the test subject.

An example report that shows the results of the analysis of the `Sober.Y` virus is presented in the Appendix.

5 Evaluation

To demonstrate the capability of TTAnalyze to successfully monitor the actions of malicious code, we ran dynamic tests on current malware samples. Then, we compared the output of our tool to a textual description for each sample. The descriptions that we used were provided by Kaspersky Lab [7]. The goal of the evaluation was to determine to which extent our analysis results match the characterizations provided by this well-known anti-virus vendor.

For the selection of our test subjects, we consulted Kaspersky's list of the most prevalent malware samples published in December 2005. Unfortunately, it was not possible to obtain samples for all entries on these lists. However, we were able to select ten different malware programs that represent a good mix of different malicious code variants currently popular on the Internet. For some of the names on the list, we received a number of different samples. Some of these samples were packed using different executable packer programs, others were not even recognized as valid Windows PE executables. From this pool, we chose one working sample for each malware type. Then, we scanned all samples for our experiments by the online virus scanner provided by Kaspersky and made sure that they were all recognized correctly.

The results of our experiments are shown in Table 1. In this table, a ✓ symbol indicates that the output of our tool exactly matches the provided description. However, in a surprising number of cases (indicated by the ✗ symbol), the output of our tool differed from the provided description. Interestingly, manual analysis confirmed that our system was indeed producing correct results, and that the behavior provided in the textual description was not reproducible. The differences between the output of our tool and the virus descriptions can have several reasons. In many cases, the general behavior reported by TTAnalyze confirmed the textual description, but the details did not match precisely. For example, both sources reported in agreement that a certain file was created in the system directory, but the file names were different. This can occur when the malicious code chooses random filenames or a name from a list of options that are not exhaustively covered by the malware description. Another reason for differences between our output and a textual description could be that the virus scanner identified an executable as a member of a certain malware variant, while in fact, the behavior of our particular malware instance has slightly changed.

In three cases, which are indicated by the ↯ symbol, our analysis failed to recognize the creation of certain Windows registry values. The reason was that these registry entries were created by the client-server subsystem process `csrss.exe` on behalf of the malicious code. Because our analysis was only recording the actions of the malware itself, we only observed the interaction of the sample with the `csrss.exe` process. However, there is no inherent restriction in TTAnalyze's design that prohibits monitoring more than one process. Thus, by also monitoring the actions of those processes that are interacting with (or started by) the malicious code, such cases can be successfully covered.

6 Conclusions and future work

Because of the window of vulnerability that exists between the appearance of a new malware and the point where an appropriate signature is provided by anti-virus companies, every new malware poses a serious threat to computer systems. This paper introduced TTAnalyze, a system to analyze the behavior of an unknown program by executing the code in an emulated environment. The goal of the analysis process is to gain a quick understanding of the actions performed by malicious code with the general aim of reducing the window of vulnerability. To this end, our tool records the invocation of security-relevant operating system functions (both Windows API functions and native kernel calls).

Because the sample program is executed completely in software on a virtual processor, TTAnalyze can tightly monitor the process without requiring any modifications to its code. This allows the system to easily handle self-modifying code and code integrity checks, two features commonly observed in malware. Furthermore, the emulated system presents itself to running processes exactly like a real system. This makes it more difficult for malware to detect the analysis environment when comparing our solution to virtual machine or debugging environments. Finally, TTAnalyze uses a complete and unmodified version of Windows XP as the underlying operating system in which the unknown program is started. Thus, TTAnalyze provides a perfectly accurate environment for malicious code.

During the course of testing TTAnalyze with real malware samples, it became apparent that dynamic analysis alone is often not sufficient to obtain the complete picture of the behavior of an unknown executable. The reason is that only a single execution path can be examined during a particular analysis run. To address this problem, we aim to extend our analysis so that multiple execution paths can be explored. For example, the process under analysis could be cloned when the emulator encounters a conditional branch. Then, the branch predicate is inverted in one process, causing both processes to follow alternative paths of the program. This could enable us to capture the behavior of an executable in different environments, with different inputs, or under special circumstances (e.g., the executable is run at a certain day of the year such as the now infamous Michelangelo virus that becomes active on the birthday of the famous artist).

Table 1 TTAnalyze test results

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

Appendix: Sample virus analysis report

The beginning of December 2005 saw a `Sober.Y` outbreak. `Sober.Y` replicates by mailing itself to other computers. The number of mails infected by `Sober.Y` was extremely high (higher than any other worm in the previous months) and so, most anti-virus vendors classified `Sober.Y` as being a critical threat.

TTAnalyze run 1

According to TTAnalyze, the following events occur when running the virus binary.

- 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 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 TTAnalyze 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 was confirmed by Michael St. Neitzel's very detailed virus description [15] 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`. Thus, we had TTAnalyze analyze this process in a second run. The results are shown below:

- 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
```

Note that we do not show the complete list of read files because it is very long. From this list, however, we observe that the process reads all files that have certain file extensions such as `.ini` and `.txt`.

Kaspersky'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 can see that the 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 were not immediately found by TTAnalyze as they were performed by the `csrss.exe` process (as explained in section 5). 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 mails sent by the worm.

References

1. Bellard, F.: Qemu, a fast and portable dynamic translator. In: Usenix Annual Technical Conference, 2005
2. Christodorescu, M., Jha, S.: Static analysis of executables to detect malicious patterns. In: Usenix Security Symposium, 2003
3. Christodorescu, M., Jha, S., Seshia, S., Song, D., Bryant, R.: Semantics-aware malware detection. In: IEEE Symposium on Security and Privacy, 2005
4. Collberg, C., Thomborson, C., Low, D.: Manufacturing cheap, resilient, and stealthy opaque constructs. In: Conference on Principles of Programming Languages (POPL), 1998
5. Computer Economics. Malware report 2005: the impact of malicious code attacks, 2006. http://www.computereconomics.com/article.cfm?id=1090
6. Hunt, G., Brubacher, D.: Detours: binary interception of Win32 functions. In: 3rd USENIX Windows NT Symposium, 1999
7. Kaspersky Lab: antivirus software, 2006. http://www.kaspersky.com/
8. Kruegel, C., Robertson, W., Vigna, G.: Detecting Kernel-level rootkits through binary analysis. In: Annual Computer Security Application Conference (ACSAC), 2004
9. Linn, C., Debray, S.: Obfuscation of executable code to improve resistance to static disassembly. In: ACM Conference on Computer and Communications Security (CCS), 2003
10. Windows Device Driver Kit 2003, 2006. http://www.microsoft.com/whdc/devtools/ddk/
11. Microsoft IFS KIT, 2006. http://www.microsoft.com/whdc/devtools/ifskit
12. Microsoft PECOFF. Microsoft Portable Executable and Common Object File Format Specification, 2006. http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx
13. Microsoft Platform SDK, 2006. http://www.microsoft.com/msdownload/platformsdk/
14. Nebbett, G.: Windows NT/2000 Native API Reference. New Riders Publishing, indianapolis, 2000
15. Neitzel, M.St.: Analysis of win32/sober.y, 2005. http://www.eset.com/msgs/sobery.htm
16. Oberhumer, M., Molnar, L.: UPX: Ultimate Packer for eXecutables, 2004. http://upx.sourceforge.net/
17. Robin, J., Irvine, C.: Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In: Usenix Annual Technical Conference, 2000
18. Russinovich, M., Cogswell, B.: Freeware Sysinternals, 2006. http://www.sysinternals.com/
19. Russinovich, M., Solomon, D.: Microsoft Windows Internals: Windows Server 2003, Windows XP, and Windows 2000. Microsoft Press, Bellevue (2004)
20. Rutkowska, J.: Red pill... or how to detect VMM using (almost) one CPU instruction, 2006. http://invisiblethings.org/papers/redpill.html
21. Symantec. Internet security threat report, 2005. http://www.symantec.com/enterprise/threatreport/index.jsp
22. Szor, P.: The Art of Computer Virus Research and Defense. Addison Wesley, Reading (2005)
23. Vasudevan, A., Yerraballi, R.: Stealth breakpoints. In: 21st Annual Computer Security Applications Conference, 2005
24. VMware: server and desktop virtualization, 2006. http://www.vmware.com/
25. Wang, C.: A security architecture for survivability mechanisms. PhD Thesis, University of Virginia (2001)
26. Yetiser, T.: Polymorphic Viruses – Implementation, detection, and protection, 1993. http://vx.netlux.org/lib/ayt01.html