MASTER THESIS
COMPUTER SCIENCE

RADBOUD UNIVERSITY

Stealthy and in-depth behavioral malware analysis with ZANDBAK

Author:
Tim van Dijk
`tim.vandijk96@gmail.com`

Internal supervisor/assessor:
dr. ir. Harald Vranken
Radboud Universiteit
`harald.vranken@ou.nl`

External supervisor:
dr. ir. Stefan op de Beek
Ministerie van Defensie
`stefan.opdebeek@gmail.com`

Second assessor:
dr. ir. Erik Poll
Radboud Universiteit
`e.poll@cs.ru.nl`

August 28, 2019

Abstract

A vast amount of malware samples emerges every day. To tackle this, there is a need for automated analysis. This can be achieved through malware analysis sandboxes: isolated environments with analytical capabilities that can monitor the behavior of programs that are run inside them. Malware authors understand the reliance on automated analysis and in response build evasive malware. This type of malware tries to detect sandbox environments and evade analysis, drastically changing the outcome of the automated analysis.

In this thesis, we present ZANDBAK: a malware analysis sandbox with in-depth analytical capabilities that defends against evasive techniques. ZANDBAK resides purely in kernel space, making it nearly undetectable to user space malware which does not have the necessary privileges to detect the presence of ZANDBAK. Furthermore, ZANDBAK has novel approaches and techniques to performing real-time stack walking, snapshotting and infection scope tracking. We describe the implementation of ZANDBAK in detail. We perform a series of experiments and a case study where we analyze an implant of the *PlugX* malware. With this, we demonstrate that ZANDBAK indeed bypasses anti-analysis techniques used in the wild and has the ability to perform in-depth analysis.

Acronyms

APC	Asynchronous Procedure Call
API	Application Programming Interface
APT	Advanced Persistent Threat
C&C	Command and Control
CPU	Central Processing Unit
DLL	Dynamic-Link Library
GUI	Graphical User Interface
IAT	Import Address Table
IDT	Interrupt Descriptor Table
I/O	Input/Output
IP	Internet Protocol
IRQ	Interrupt Request
IRQL	Interrupt ReQuest Level
KMDF	Kernel-Mode Driver Framework
MAC	Media Access Control
MDL	Memory Descriptor List
OS	Operating System
PE	Portable Executable
RAM	Random Access Memory
SEH	Structured Exception Handling
SSDT	System Service Dispatch Table
SSN	System Service Number
SST	System Service Table
VM	Virtual Machine

VMI Virtual Machine Introspection

VMM Virtual Machine Monitor

WDM Windows Driver Model

WMI Windows Management Instrumentation

Contents

Chapter 1

Introduction

Malware, short for malicious software, is a term that refers to software made by attackers that aims to disrupt computer operation, gather sensitive information, or gain access to private computer systems [20]. It is a blanket term that not only describes the traditional categories such as computer viruses, worms, key loggers, rootkits, adware, etc., but also combinations thereof. Unsurprisingly, the increase in popularity of the term malware coincides with the rise of such hybrid malware, exhibiting behaviors of multiple categories [31].

The theoretical preliminary work on malware goes all the way back to 1949, when mathematician John von Neumann's article on the "Theory of self-reproducing automata" was published [22]. The article describes systems that are capable of constructing copies of themselves, passing along their programming. Clearly, this is very similar to how, for example, computer viruses or worms spread.

22 years later, in 1971 von Neumann's theory was first brought into practice by Bob Thomas at BBN as he wrote the experimental computer program *Creeper*. Creeper is considered to be the first computer worm. Creeper displays the message "I'm the creeper, catch me if you can!" and then scans the ARPANET for another mainframe that can serve as a host for the program. Once one is found, it does not copy but effectively moves itself there, where the process is started all over [6].

In those days, developing malware was mostly about showing off one's skills. Since then, malware and the scene surrounding it have undergone a substantial transformation. Tens of billions of devices are connected to the internet and the goals of malware authors have shifted towards far more lucrative objectives, i.e. money, intelligence and power [1]. Today, the cybercrime industry is thriving and acts of cyberterrorism and cyberwarfare are becoming increasingly more commonplace [16, 62].

A vast amount of new malware samples emerges every day. Kaspersky Lab, an antivirus software company, reported detecting 360.000 new malicious files a day in 2016 [49]. The detected malware exhibits a wide array of behavior and to understand and manage the threat they pose, it is quintessential that analysis takes place. This overwhelming amount of malicious files cannot be analyzed through manual reverse engineering efforts alone. Clearly, there is a need for automated malware analysis.

One way to perform behavioral analysis is through the use of a sandbox environment. A sandbox is a security mechanism that restricts a program's access to the host machine [14]. Using a sandbox, we can execute malware while minimizing the risk of damage. A sandbox can be enhanced with analytical capabilities that allow it to monitor the workings of the programs running inside of it.

Because malware authors understand the reliance on automated analysis, malware today has started to look for this and behave differently when it detects this type of environment. Such evasive / environment-aware malware could either refuse to execute or modify its runtime behavior in an attempt to mislead the analysis [11, 2]. The techniques employed by evasive malware have little effect on static analysis, but can drastically change the outcome of behavioral analysis as the collected data is potentially worthless or worse, it could even start the analyst off on the wrong foot. Consequently, it is critical that a sandbox for malware analysis is sufficiently stealthy such that it evades detection.

In this thesis, we present ZANDBAK[1]: a modern, stealthy, kernel-based sandbox with novel malware analysis capabilities. Furthermore, we demonstrate its practical use by analyzing various pieces of malware with it, demonstrating ZANDBAK's stealth and ability to monitor behavior in-depth.

ZANDBAK is made for Windows 10, which as of now is Microsoft's latest operating system for personal computers. By residing purely in the kernel, it is nearly undetectable to user space malware which does not have the necessary privileges to detect ZANDBAK. Still, kernel mode sandboxes are relatively uncommon, presumably due to Windows kernel development being a challenging and extremely time-consuming endeavor for reasons described in Section 5.3.1. ZANDBAK's novel capabilities include real-time stack walking, snapshotting and infection scope tracking, all of which are described in Chapter 5 in detail.

1.1 Research questions

The main goal of developing a modern, stealthy, kernel-based sandbox with novel malware analysis capabilities can be split into two sub-goals:

- Gain insight in techniques used by malware authors to evade sandboxes, and in defenses that counter those techniques.
- Build a sandbox environment for malware analysis that employs such defenses.

These goals translate into the following research question:
Can one build a sandbox environment for the analysis of malware that renders evasive techniques ineffective?

This main research question breaks down into the following sub-questions:

SQ1. What techniques do malware authors use to evade sandboxes?

SQ2. How can one build a sandbox in which malware can be executed?

SQ3. How can defenses against sandbox evasion be implemented?

SQ4. How can one monitor the behavior of malware that is executed in a sandbox?

SQ5. How can one monitor the spread of malware through various forms of injection or low-level modification of processes?

SQ6. To what extend does ZANDBAK render evasive techniques ineffective?

[1]Zandbak is Dutch for sandbox.

1.2 Thesis outline

The remainder of this thesis is organized as follows. Chapter 2 covers some information about the Windows operating system, which is useful because the sandbox is implemented as a driver for the Windows kernel. Chapter 3, describes several types of malware, how they can be analyzed and how malware might evade analysis. Chapter 4 contains information about how sandboxes work in general and in Chapter 5 we explain the design, features, implementation and set-up of ZANDBAK. Chapter 6 describes experiments and a case study that show ZANDBAK's practical use. Chapter 7 describes related work and, where relevant, compares ZANDBAK to it. Chapter 8 suggests future work and finally Chapter 9 provides brief concluding remarks.

Chapter 2

Windows system architecture

Most operating systems implement various privilege levels at which code can run, often using support provided by the underlying hardware. Windows is no exception. In this chapter, we provide background information on these privilege levels and explain how system services work. System services are the mechanism through which lesser privileged code can make use of operating system resources. Programs, including malware, often run with low privilege and can do very little without continuously invoking system services. As such, using the in-depth knowledge of this mechanism provided in this chapter, we will later show that it can be adjusted to monitor malware's behavior.

2.1 Privilege levels

The architectures of x86 and x64 processors define four privilege levels to protect resources from being overwritten by those with lesser privilege. These levels are also referred to as ring 0 (most privileged) through ring 3 (least privileged) and are shown in Figure 2.1. Windows uses only two of these rings: ring 0 for kernel mode and ring 3 for user mode [55]. At any time, the CPU runs at a specific privilege level, determining what code can and cannot do. The processor provides this necessary foundation such that operating system designers can implement mechanisms that ensure misbehaving user applications cannot compromise the system as a whole.

In kernel mode, one has nearly unrestricted access to the underlying hardware. Every CPU instruction can be executed and all memory addresses and I/O ports can be accessed. This is where core operating system components reside, including drivers, although some drivers may run in user mode. If a program crashes in kernel mode and the system is unable to recover, the entire system comes to a halt.

Most programs execute in user mode. It is a 'safer' place to run as a crash does not bring the whole system down, although access to resources is quite limited. When a program is started a process is created for it by the operating system. Among other things, a process provides a private virtual address space, allowing a program to run in isolation from other programs. This is different from kernel mode where all code shares a

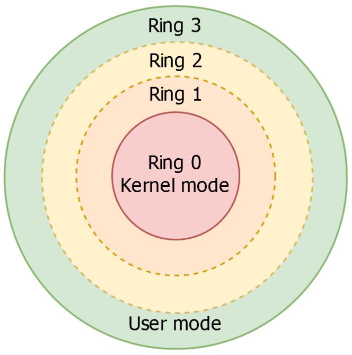


Figure 2.1: Privilege levels as defined in the x86 and x64 architectures.

single virtual address space.

User space programs often require access to resources that are normally only available in kernel mode. In fact, a user mode program can do almost nothing to the outside world without access to these resources. For example, it cannot access the hard drive, send data over the network, allocate memory, or even print to the screen. Windows offers regulated access to some of these resources through a documented interface called the Windows API.

Opposed to documented features, such as the Windows API, there are *undocumented features*. These are features that are not officially documented by Microsoft and are subject to change at any given time. Often some documentation on undocumented features is available by third-parties. This information often is found through the efforts of reverse engineers and potentially is inaccurate or outdated.

2.2 Using operating system services

The Windows API is the user mode system programming interface to functionality provided by the operating system. It is a large set of documented, callable subroutines that serve as a wrapper around the underlying, undocumented native system services (or system calls) [26]. Of course, it is possible to directly use the system calls rather than using the wrapper, but one should be beware of that their functionality might change as Windows gets updated.

To further illustrate how user space programs interact with the OS and what happens behind the scenes, let us consider an example. Assume we create a program in C in which we include `windows.h` and subsequently call its `FindNextFile` function. This results in the following actions being taken: [3, 27]

1. Including `windows.h` causes `kernel32.dll` to be dynamically loaded;
2. Calling `FindNextFile` redirects program flow to corresponding code in `kernel32.dll`;
3. `FindNextFile` needs a list of files in a directory, so it uses the native system service `NtQueryDirectoryFile` in `ntdll.dll` to get one.
4. `NtQueryDirectoryFile` prepares arguments and then transfers control to the kernel using either an `INT 0x2e` or `SYSENTER` instruction.
5. (a) In case of the `INT 0x2e` instruction, the kernel gets interrupted and an interrupt service routine is called, namely the one pointed to by entry 0x2e of the Interrupt Descriptor Table (IDT). Normally, this results in `KiSystemService` being called. We can specify a system call number by storing it in the eax register.

   (b) In case the `SYSENTER` instruction is used, the processor jumps to kernel space code at the address in `SYSTENTER_EIP_MSR (KiFastCallEntry)`, which in turn leads to `KiSystemService` being executed without the overhead of an interrupt [65].
6. Depending on the arguments, `KiSystemService` uses either `KeServiceDescriptorTable` or `KeServiceDescriptorTableShadow`. The `ServiceTable` field of these structures contains pointers to an array of linear addresses, known as the System Service Dispatch Table (SSDT). The entry that corresponds to `NtQueryDirectoryFile` is found, and is used to finally transfer control to the actual implementation of the `NtQueryDirectoryFile` in kernel mode code.

7. `NtQueryDirectoryFile` completes its task by interfacing with the I/O manager which in turn communicates with the underlying hardware through layered drivers. Afterwards, it returns control to the program running in user space.

2.3 System service dispatcher

Whether the user mode code execute `INT 0x2e` or `SYSENTER`, the result is the same: the kernel's system service dispatcher, `KiSystemService`, ends up being invoked. In summary, the system service dispatcher creates a copy of the caller's arguments on the thread's user mode stack to its kernel mode stack prior to locating and executing the system service code. The user cannot manipulate the kernel mode stack, so it ensures the user does not manipulate the arguments while the system service is being executed.

Figure 2.2 provides an overview of how the system service dispatcher navigates the nested data structures in order to redirect execution to the correct kernel routine. Let us take a closer look at how this component of the Windows kernel works.

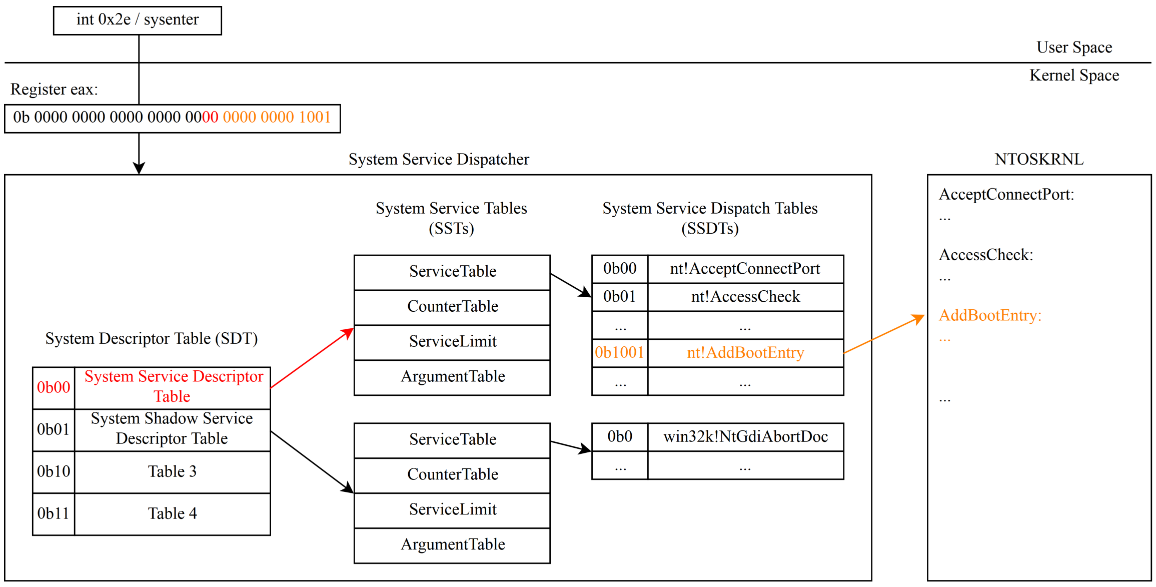

int 0x2e / sysenter
User Space
Kernel Space
Register eax:
0b 0000 0000 0000 0000 0000 0000 0000 1001
System Service Dispatcher
NTOSKRNL
System Service Tables (SSTs)
System Service Dispatch Tables (SSDTs)
ServiceTable
CounterTable
ServiceLimit
ArgumentTable
0b00 nt!AcceptConnectPort
0b01 nt!AccessCheck
...
0b1001 nt!AddBootEntry
System Descriptor Table (SDT)
0b00 System Service Descriptor Table
0b01 System Shadow Service Descriptor Table
0b10 Table 3
0b11 Table 4
0b0 win32k!NtGdiAbortDoc
AcceptConnectPort:
AccessCheck:
AddBootEntry:
...

Figure 2.2: Overview of the system service dispatcher's functionality.

In order to know where the system service code resides and how many bytes of the user mode stack must be copied, it first makes use of the Service Descriptor Table (SDT). The SDT is structured as follows:

```
typedef struct _SERVICE_DESCRIPTOR_TABLE {
    SYSTEM_SERVICE_TABLE ntoskrnl; // System Service Descriptor Table
    SYSTEM_SERVICE_TABLE win32k;   // System Shadow Service Descriptor Table
    SYSTEM_SERVICE_TABLE Table3;   // Reserved for device drivers
    SYSTEM_SERVICE_TABLE Table4;   // Reserved for device drivers
};
```

The value in the eax register determines what system service from what System Service Table (SST) should be used. It is encoded as follows:

- Bits 0-11: system service number (SSN);
- Bits 12-13: system service table (SST);
- Bits 14-31: not used.

Using a kernel debugger, we can see that Windows indeed only uses two out of four possible SSTs:

```
kd> x nt!KeServiceDescriptor*
fffff801`78176880 nt!KeServiceDescriptorTable = <no type information>
fffff801`7815f980 nt!KeServiceDescriptorTableShadow = <no type information>
```

These SSTs use the following structure:

```
typedef struct _SYSTEM_SERVICE_TABLE {
    PDWORD ServiceTable;   // System Service Dispatch Table (SSDT)
    PDWORD CounterTable;   // Not used in Windows free build
    DWORD  ServiceLimit;   // Number of function pointers in SSDT
    PBYTE  ArgumentTable;  // Array of byte counts
} SYSTEM_SERVICE_TABLE;
```

After selecting the appropriate SST, the SSN is used twice as an index. The first time, it is used as an index to find a value in the `ArgumentTable` which tells us the amount of bytes that must be copied from the user-space stack to the kernel-space stack. Subsequently, it is used to extract an entry from the SSDT. The entry we find is a pointer to the routine that implements the requested system service.

2.4 Interrupt Dispatching

Bovet and Cesati [4] define interrupts as:

> "An interrupt is usually defined as an event that alters the sequence of instructions executed by a processor. Such events correspond to electrical signals generated by hardware circuits both inside and outside the CPU chip."

Synchronous interrupts are generated inside of the CPU as result of an instruction's execution. These are also referred to as exceptions or software interrupts. Exceptions can either be generated by the processor (faults, traps, abort) or be triggered by assembler instructions (`INT` X).

Asynchronous interrupts are triggered by an external device and are simply referred to as interrupts. Devices send interrupt requests (IRQs) to the CPU to signal that it requires attention. They are asynchronous because they are generated at arbitrary times with respect to the CPU's clock. Depending on the priority of the interrupt, the CPU eventually postpones its current task to handle the device's request. This system is opposed to polling, where the CPU checks with the external device if it requires attention.

Each interrupt is identified by an interrupt vector which is a number between 0 and 255. Interrupt vectors in the range 0 to 31 are for exceptions and non-maskable interrupts. Non-maskable means the interrupt must be handled right away as opposed to maskable interrupts which can be disabled or ignored. Maskable interrupts fall in the range 32 to 47. IRQ0 to IRQ15 are assigned to this range. The remaining vectors 48 to 255 are for software interrupts.

2.4.1 Interrupt Descriptor Table

The Interrupt Descriptor Table (IDT) is an array of up to 256 `KIDTENTRY` structures that resides in kernel space memory. Each entry in this table is 8 bytes in size and corresponds with a specific interrupt vector. When an interrupt occurs, the processor multiplies interrupt vector by 8 and adds it to the IDT base address. The resulting memory address is verified to be in the IDT. If everything checks out, the structure at the computed address is loaded and information in it is used to further handle the interrupt.

Using the kernel debugger, it is possible to look up the definition of the `KIDTENTRY` structure:

```
kd> dt nt!_KIDTENTRY
   +0x000 Offset           : Uint2B
   +0x002 Selector         : Uint2B
   +0x004 Access           : Uint2B
   +0x006 ExtendedOffset   : Uint2B
```

Which translates to the following structure in C:

```
typedef struct _KIDTENTRY {
    USHORT      Offset;          // Lower part of the offset
    USHORT      Selector;        // Kernel segment selector
    USHORT      Access;          // Gate specific information
    USHORT      ExtendedOffset;  // Higher part of the offset
} KIDTENTRY;
```

The `Offset`, `ExtendedOffset` and `Selector` fields together are used to specify the location of the function handling the interrupt. This address is jumped to when an interrupt occurs.

Each entry can be any of three types of gates: an interrupt gate, a task gate or a trap gate. Which one it is depends on the value of a flag in the Access field: a 16-bit value built up from the bit fields shown in Table 2.1 [64].

Name	Bit(s)	Description
Present	15	Set to 0 for unused interrupts.
Descriptor Privilege Level	14-13	The minimum privilege level the calling descriptor should have to be allowed to call this gate.
Storage Segment	12	Set to 0 for interrupt and trap gates.
Type	11-8	Specifies the gate type this entry represents: Value — Gate type 0x5 — 80386 32-bit task gate 0x6 — 80286 16-bit interrupt gate 0x7 — 80286 16-bit trap gate 0xE — 80386 32-bit interrupt gate 0xF — 80386 32-bit interrupt gate
Reserved	7-0	Specified to be 0.

Table 2.1: Bit field of a gate's Access field.

It depends on the situation what type of gate is required. In the context of system services, trap gates are used. Trap gates are used for code redirection and transitions in privilege level. They can be invoked with the `INT` X instruction. Interrupt gates are nearly identical to trap gates apart from the fact that interrupt gates temporarily disable interrupts whereas traps gates do not. Task gates were designed for hardware task switching but are rarely used due to the existence of faster mechanisms.

The IDT is initialized by Windows at boot time. The processor has a special register (IDTR) in which both the physical base address and the length of the IDT are stored. The x86 instruction set contains two instructions to interact with the IDTR:

- `SIDT M` stores the value in IDTR to memory address M;
- `LIDT M` loads the value at memory address M into IDTR.

Although, the former instruction can only be used from ring 0, unprivileged code can use the `LIDT` instruction and obtain the address of the IDT.

2.4.2 Kernel Patch Protection

Kernel Patch Protection (KPP), nicknamed PatchGuard, is a feature introduced in 2005 for 64-bit Windows operating systems that aims to protect the kernel's overall reliability, performance and security by preventing uncondoned behavior. It is a security mechanism that is intended to prevent both malware and third-party vendors from making unsupported modifications to the kernel of the Windows operating system [19]. At the time of its creation, it was somewhat controversial because kernel patching was not only used by malicious software but also by security products such as firewalls and antivirus software [40]. It is designed to protect the following vital kernel structures: [63]

- SSDT (System Service Descriptor Table)
- GDT (Global Descriptor Table)
- IDT (Interrupt Descriptor Table)
- System images (`ntoskrnl.exe`, `ndis.sys`, `hal.dll`)
- Processor Model Specific Registers

PatchGuard routinely checks the integrity of the kernel by comparing kernel components against known good copies / signatures. In case of a mismatch, it causes a bug check (number: 0x109; code: `CRITICAL_STRUCTURE_CORRUPTION) -` an unrecoverable fatal error after which the system shuts down.

In particular, the protection of the SSDT is troublesome for rootkits (see Section 3.1). This is because SSDT hooking, a technique widely used by rootkits can no longer be used. Before, rootkits could control code flow by intercepting calls to system services. To do so, the rootkit would overwrite the pointer to the implementation system service in the SSDT with one that points to attacker controlled code, redirecting code flow. In case of a rootkit, this code usually serves as a wrapper: it calls the system service and returns some fabricated data afterwards, possibly hides its presence. Although it achieves its goal of making it harder for malware to take control of the system, it has the side effect of preventing legitimate products from working properly. For example, antivirus software often employs SSDT hooking to monitor system behavior.

We must notice that PatchGuard simply runs in ring 0 and has no special privileges compared to the rest of the kernel. Remember that drivers also operate from ring 0 and thus enjoy the same privileges PatchGuard has. Therefore, there is no fundamental barrier that stops drivers from trying to corrupt PatchGuard. The team that developed PatchGuard realized this limitation and resorted to *security through obscurity*, making extensive use of misdirection, and obfuscation techniques. "While many would argue that security through obscurity adds nothing, the authors believe that it's merely a matter of raising the bar high enough so as to eliminate a significant number of people from being able to completely understand something." [63] Unsurprisingly, security researchers managed to see through Microsoft's misdirection and have successfully devised ways to disable PatchGuard. One example of such a project is the UPGDSED project by EP_X0FF and Fyrre [41], which is openly available on GitHub. UPGDSED supports the following versions of Windows: Windows 7 SP1, Windows 8, Windows 8.1, Windows 10 (TH1/TH2/RS1/RS2/RS3). Note that Windows 10 RS3 coincides with the system ZANDBAK is designed for.

Chapter 3

Malware behaviors & analysis

Malware plays a part in most computer security intrusions and incidents. In this chapter, we discuss how malware can be analyzed and how different types of malware usually operate.

3.1 Types of malware

Based on the infection mechanism and behavior, malware can be grouped into various categories. As already described in the introduction, malware increasingly often exhibits behavior of multiple categories. For example, a hybrid virus/worm typically features a virus' ability to alter the code flow of a program with the worm's ability to propagate to other machines.

Every day, new malware samples are being created with seemingly endless capabilities, so we cannot possibly cover all categories of malware. Nevertheless, following is a brief description of common malware categories.

Downloader Downloaders are often used by exploit kits or mail attachments as the first stage of an attack. Downloaders are usually small and are programmed to download and run another piece of malware from the internet [47].

Launcher / Dropper Launchers are malicious software that are used to launch other malware, often using stealthy and unconventional methods to avoid detection. Unlike Downloaders, they often contain the malware they are designed to execute [47].

Virus Computer viruses are programs with the ability to replicate themselves. They often do so by inserting themselves into executable files. This spread is not limited to the local machine, as it might also spread over the network or media such as USB drives [8].

Worm Similar to viruses, worms also replicate and propagate themselves to other machines. The difference lies in that they only replicate over the network and do not need the support of any file [8].

Bot A botnet is a collection of infected internet-connected machines that are being controlled by a third-party. Each such device is known as a "bot". Bots are often used in distributed denial of service attacks and for sending spam [8].

Ransomware Ransomware attempts to extort users by disabling functionality in the system in some way, effectively holding the system hostage [59]. The ransomware then demands payment to restore that functionality, often through anonymous payment mechanisms (e.g. Bitcoin) [28].

Rootkit Hoglund defines rootkits as "a set of programs and code that allows a permanent and undetectable presence on a computer" [15]. An attacker can install a rootkit after having obtained root or administrator access, and is used to gain permanent foothold on a system. Once installed, rootkits take active measures to ensure they remain hidden from detection and can take measures against removal.

Spyware Spyware attempts to silently monitor the behavior of users, record web surfing habits, or steal sensitive information such as passwords or intellectual property [10]. The spyware later reports the collected information back to the spyware distributor for further exploitation.

3.2 Malware analysis

Malware analysis is the process of learning how malware functions. As malware can exhibit a wide range of functionalities, it is essential to know which features a certain sample has.

Malware analysis is usually performed in response to a network intrusion. In this case, the analyst's main goal is to provide the information needed for a correct resolution of the incident. The first step is to determine all infected machines and files, and after a cursory analysis, determine which files require full analysis.

There is a myriad of goals that an analyst can have in mind when performing malware analysis. Some of these goals may include: [47]

- assessing damage;
- identifying a vulnerability;
- finding an indicator of compromise (to detect other victims);
- attributing the attack;
- gaining insight in the techniques used by the malware.

Once the analyst has achieved these goals, there should be a clear picture of what a particular suspect binary can do, how it can be detected on the network, and how damage can be measured and contained [47].

Most often, malware samples are found in the form of an executable or shared library file (e.g. .DLL) rather than source code and will therefore not be human-readable. In order to get a notion of what the program is supposed to perform if it were to be executed, a variety of tools and techniques needs to be employed.

There are two main approaches to malware analysis: static analysis and dynamic or behavioral analysis. For a malware analyst, the fastest path to success usually involves a combination of both.

3.2.1 Static analysis

Static analysis means the malware is examined without running it and is usually performed by extracting useful information from different resources of the binary file.

A good start could be to search the binary file for data that can be interpreted as a string. This might reveal host names and IP addresses for a command & control (C&C) server, copyright strings for third-party libraries, or even information about the developing platform such as a path to debugging symbols (i.e. .pdb files). Embedded paths could reveal host names, usernames or the name of the malware. Such information can be a very valuable resource for attribution.

One of the most useful pieces of information we can gather is the list of imports. Imports are functions used by a program that actually reside in an external source, such as a shared library.

Most often, malware imports functions from Windows DLLs that contain functionality to interact with the operating system. These functions are often well-documented, which saves the analyst the substantial amount of effort as one can simply look up the functionality. For example, if we see that it imports `OpenProcess`, `GetCurrentProcess` and `GetProcessHeap`, we can assume that the software can open and manipulate processes.

Sometimes malware also uses third-party DLLs, for example, a Mozilla Firefox DLL could be used to connect back to the server. Whenever this happens, it is safe to infer that the malware uses this program to achieve its goals.

One can use a disassembler such as *GDB*, *IDA* or *Radare2* to translate the machine code into assembly instructions that can be read and understood by humans. The resulting assembly can be analyzed to figure out the functionality of portions of the program. The analyst should be rather selective in what parts they want to manually determine the functionality of, as this is a time consuming process and most assembly is not important for the purpose of understanding the malware.

Some tools, like the *Hex-Rays decompiler* and *Ghidra* take disassembly one step further and actually try to reconstruct the source code. This process is supposed to be the inverse of compilation and is hence called decompilation. In some cases, the product is almost as easy to understand to the analyst as the real source code was to the malware author. For example, Java's .jar files contain enough information to nearly achieve full source code recovery. In case of C / C++ however, a lot of information about the source code is lost during compilation.

Static analysis at a machine code level can be extremely cumbersome because malware often uses anti-analysis techniques. Malware often uses code obfuscation techniques such as compression, packing, encryption, or self-modification to complicate static analysis [32].

3.2.2 Behavioral analysis

Behavioral analysis executes malware in a controlled and monitored environment to observe its behavior. The environment, often called a sandbox, allows the malware to perform all of its malicious operations. Typically, observing the behavior involves recording which system calls are invoked, together with their parameters [21]. After analysis is complete, most sandboxes generate a report based on the results of the automated analysis. Such a report contains rich information about runtime behaviors and shows what operating system resources were used such as files, Windows registry and network connections.

A major advantage is that the sandbox can automate many tasks that would normally be done by human analysts and reverse engineers, leading to vastly improved scalability. Using the recorded data, analysts can obtain the sample's unpacked code, detect botnet C&C servers and generate signatures for C&C traffic as well as remediation procedures for malware infections [18].

Obviously, behavior-based malware analysis only works if the malware actually performs relevant malicious operations during its analysis. Moreover, unlike static malware analysis where all possible executions are taken into account, here it analyzes only a single malware execution at a time. This is a substantial drawback as this approach has the potential to miss interesting behavior that the sample might exhibit under different circumstances. Determining whether malware shows its true colors is actually shown to be undecidable [25]. Nevertheless, approaches that increase execution path coverage exist, such as the one by Moser et al. [21]. Their approach takes advantage of virtualization - making snapshots on branches in code execution, returning to the snapshots and manipulating memory objects to ensure different paths are taken. Doing so allows them to build a more complete view of the sample's potential behaviors.

Despite its shortcomings, behavioral analysis is a widespread approach as it has the potential to scale well and it largely sidesteps the difficulties involved in the static analysis of obfuscated code [18].

3.3 Evasive malware

Sophisticated malware often employs a series of techniques to maintain stealth or hinder efforts for analysis [1]. We refer to malware that uses such techniques to make analysis systems, such as sandboxes, ineffective, as *evasive* malware. In order for malware to successfully evade analysis, it must distinguish between running in an analysis environment and in a plain machine and adjust its behavior accordingly.

Interesting to note is that evasive techniques can prove to be a malware's Achilles heel. An example of this is the WannaCry ransomware that infected more than 230.000 computers across 150 countries in a worldwide cyberattack in 2017 [12]. As an evasive technique, WannaCry probes a domain that it knows should not exist. Consequently, it expects the operation to fail. If the probe unexpectedly succeeds, it would know something is out of the ordinary. WannaCry assumes that this indicates the presence of a sandbox performing a naive form of network emulation. To prevent analysis, the malware would then shut down. When security researchers discovered this evasive technique, they registered the domain that WannaCry probes. Now, whenever WannaCry probes this domain, the operation succeeds, which makes it believe it runs in a sandbox and causes it to shut down. By registering the domain, the researchers effectively flipped WannaCry's kill switch.

Grouped by which part of the analysis system they attempt to circumvent, we consider three types of evasive techniques:

- Anti-debugger
- Anti-virtualization
- Anti-sandbox

In the remainder of this section, we will take a closer look at the last two of these types of evasive techniques and see how they relate to ZANDBAK. We do not consider anti-debugging techniques because they are irrelevant as ZANDBAK is not a debugger. The possibilities for evasion are only bound by creativity. Consequently, it is impossible to provide an exhaustive list of techniques. Instead, we will describe of few interesting techniques.

3.3.1 Anti-virtualization

Malware analysts often conduct their research in isolated environments, such as VMs. A VM allows the flexibility to run malware or run it in a debugger without fear of infecting the host. Furthermore, after infection, the VM can quickly be reverted to a clean snapshot to continue analysis.

VMs also are used by a security products, in particular, antivirus solutions. Historically, antivirus solutions work using signature scanning, which detects threats by searching for patterns. The majority of antivirus solutions on the market today also employ some type of heuristic detection [30]. Most heuristic antivirus systems run the suspect binary in a sandbox / virtual environment and see what happens. Afterwards, a rule or weight-based system is applied to assess the danger that the suspect binary poses. If the danger level exceeds a predetermined threshold, the suspect binary is flagged as malicious and appropriate measures are taken.

Malware developers are well aware of this reliance on virtualization and started to invent ways circumvent these systems. If malware appears benevolent in a VM, it complicates research by malware analysts and can circumvent heuristic detection, while still performing malicious activities on *real* machines.

For malware developers, evading virtual machines also has drawbacks. This is due to the incorrect assumption that VMs are only used for malware analysis. In reality, it is not uncommon for valuable targets to run in a VM. For example, consider companies that run their servers through a cloud service such as Amazon Web Services (AWS) or the Microsoft Azure Platform. These servers are most definitely valuable targets: they can be mission critical for the company or could contain company secrets, customer data, etc. However, as these cloud services make use of virtualization, the malware will not trigger, severely limiting its range of potential targets. Fully aware of this, malware developers sometimes choose to refrain from using anti-virtualization techniques.

Evasive techniques are based on the assumption that systems of interest carry characteristics that differentiate them from plain systems [7]. Let us now take a look at several techniques for detecting VMs.

Example anti-virtualization techniques

RUNNING PROCESSES & SERVICES Malware enumerates all running processes on the system. Subsequently, it compares process names against a list of process names that are known to identify the presence of a virtual machine.

REGISTRY ARTIFACTS The Windows registry is checked for keys that reveal the presence of a VM. The key HKLM\HARDWARE\Description\System\SystemBiosVersion, for example, could be checked for “VMWARE” or “VBOX”.

Filesystem artifacts Virtualization solutions, such as VirtualBox, often offer a set of device drivers and system applications that optimize the guest operating system for better performance and usability. Malware could check the filesystem for the existence of files that indicate the installation of such software that normally would not be present on non-virtualized systems.

Backdoor detection In most virtualization solutions, there is a channel over which host and guest OS can communicate. In case of VMWare, this is implemented as an I/O operation to a specific port address. Malware can also try to query this backdoor. If successful, it indicates the presence of a hypervisor.

MAC address Virtual machines often use virtualized network adapters. These adapters naturally have a MAC address. The problem is however that the VendorID part of these MAC address correspond to values that are known to belong specific brands of VMs.

Number of cores Malware analysts commonly allocate a single processor core to their VMs. However, today's processors, to which plain machines have access, will almost always have multiple cores.

Username & hostname Malware can check the username and hostname of the system for (sub)strings that would suit a VM for malware analysis. Strings such as "vm", "malware", "sandbox" are commonly used, but so are the names of anti-virus companies or even the usernames of specific malware analysts.

Remote timing test Chen et al. [7] have shown that remote hosts can be fingerprinted using the TCP timestamp option to measure the clock skew. They observed that virtual machines can be identified because they exhibit more perturbed clock skew behavior. This is because virtual machines do not get as accurate timing information as plain machines get. Plain machines get regular hardware interrupts generated by hardware oscillators whereas virtual machines rely on VMM-generated software interrupts, which can be lost or delayed.

3.3.2 Anti-sandbox

Instead of trying to detect the virtual machine the sandbox relies on, malware authors also have the option to directly try to detect the sandbox. This clearly is a much more fine-grained approach to avoiding behavioral analysis and, for the malware author, comes with the benefit of not missing out on potentially valuable targets. Furthermore, it no longer relies on the false assumption that sandboxes must run on virtual machines: sandboxes function equally well on plain machines - only cleanup gets more complicated. To evade behavioral analysis, malware can either behave differently in the presence of a sandbox or circumvent the analysis mechanisms of the sandbox.

In the next section, we describe a number of anti-sandbox mechanisms. Some of the techniques we look at are only used to detect sandboxes whereas others are aware of the mechanisms sandboxes use or the context they are used in.

Example anti-sandbox techniques

Agent artefacts Sandboxes often have an agent running on the system. This agent handles communication and data exchange with the host. This agent might be detected, either as a running process or through files on disk.

Sleep architecture Sandboxes are commonly used to analyze large amounts of malware samples around the clock. To improve throughput, sandboxes are commonly designed to monitor each sample for a limited amount of time. Malware authors are aware of this context and try to circumvent analysis by postponing execution, for example by sleeping. In response, sandbox developers engineered various methods of preventing such delays. Normal systems do not try to prevent delays so malware authors, in turn, try to detect sandboxes by finding discrepancies in these methods.

Window classes A window class is a set of attribute that Windows uses as a template to create a window. Malware can check if any of the windows that are currently open use a window class that belongs to a known sandbox.

Loaded DLLs Sandboxes such as Cuckoo load a DLL into the target process to perform monitoring. Malware can list which modules are loaded in its process and check if any unexpected modules appear.

Parent process Each process has a parent, including malware processes. When a sandbox agent starts a malware sample in a naive manner, it becomes the parent of the malware process. This means that malware can check if its parent is the agent of a known sandbox.

Current path Sandboxes and analysts often give malware samples obvious names and place them on straightforward places on the filesystem. For example, the malware can easily check if its filename contains “malware”, “sample” or the hash of itself.

Unhooking In some cases, malware can completely sidestep a sandbox’ analysis by removing the hooks with which it monitors the sample. Hooks placed by the malware replace the addresses of certain functions. Unhooking involved manually resolving the addresses of these functions to restore their original value.

Chapter 4

Malware analysis sandboxes

In 1996, one of the first papers on sandboxing was published by Goldberg et al. [14] In this paper, they coin the need for a secure environment wherein untrusted helper applications (e.g. browser add-ins) can securely be run. Being led by the basic assumption that "an application can do little harm if its access to the underlying operating system is appropriately restricted", they built Janus: a proof-of-concept sandbox that confines untrusted software and data by monitoring and restricting the system calls it performs.

Despite both being sandboxes, there are fundamental differences between restrictive sandboxes such as Janus and sandboxes for malware analysis. The goal of a restrictive sandbox is to restrict the capabilities of untrusted programs in such a manner that they pose no danger to the rest of the system. Contrarily, malware analysis sandboxes should aim to be as transparent as possible towards the programs running inside it while still ensuring security. Besides that, malware analysis sandboxes focus much more on monitoring behavior than restricting actions.

In this chapter, we take a look at existing sandboxes to see what behavior they monitor and how they do it. To be more specific, in Section 4.1 we show how sandboxes operate in general. Then in Section 4.2, we look at various types of hooks that sandboxes can use to monitor behavior.

4.1 Overview

Looking at the basic architecture of sandboxes for malware analysis, three recurring components can be identified:

1. the server / host
2. multiple sandboxes / analysis guests
3. a (virtual) network that connects the machines

The server is responsible for guest and analysis management. It starts analysis and generates reports. Often a user interface is provided in which samples can be deployed and reports can be viewed.

The guests run an agent that follows the server's instructions. After running a sample, they are reverted to a clean state. These machines run malware samples and monitor their behavior. Afterwards, the recorded behavior is sent back to the server, which in turn uses that information to generate a report.

4.2 Potential hooking locations

In computer programming, the term hooking refers to a technique that allows for viewing, interacting with, or changing something that is already running in a system. It provides a straightforward mechanism that can easily alter a program's behavior without having the source code available, it has a wide range of possible purposes. For example, hooking can be used to aid in debugging, extend functionality, patch vulnerabilities, or benchmark programs (e.g. by measuring frame rate). There also are quite nefarious applications. For example, hooking is widely used in video game hacking. A rather simple example would be to hook the random number generator such that it always returns 'optimal' values instead of random ones. Certain pieces of malware, such as rootkits, also make use of hooking; often to fake output of routines that would otherwise reveal their existence.

Sandboxes for malware analysis often use hooking to monitor programs that are running on the machine. Still, there is a myriad of ways and places to hook. Looking back at Section 2.2 and considering the steps that are taken when a system service is invoked, we can identify various candidate locations that can be hooked in order to monitor system service usage. In the remainder of this section, we will try to answer the question "what hooking technique would best suit our needs?".

4.2.1 User space hooks

A tried and tested approach is to hook the Windows API in user space [33]. The easiest strategy would be to directly patch the executable or DLLs. For example, if we want to hook `function A` and redirect it to `function B`, we can do so by simply inserting a relative jump at the start of `function A` containing the offset to `function B`. A downside is that it is simple to detect: simply checking if the first instruction is a relative jump suffices since this is a rather unusual variation of the function prologue.

A popular type of user space hook is one in the Import Address Table (IAT). The IAT is a lookup table which is used when the program calls a function from a different module (i.e. DLLs). Hooking the IAT is usually achieved via DLL injection. When the DLL is loaded into the program's memory, it can overwrite pointers in the IAT to handlers within the DLL [51].These hooks can be detected by verifying that each entry in the IAT points to the appropriate module.

Of course, there are many more user space hooking techniques and tricks for hooks to prevent detection. This battle between detection and evasion can be thought of as a never-ending arms-race between hook developers and detectors. Fundamentally, all user space hooks are detectable by user space programs as they are all on equal footing regarding privileges. Machine code, including hooks, can only maintain integrity and stealth against opponents who are less privileged. To compromise stealth, all an adversary needs to know is where to look. Similarly, a program cannot maintain integrity against an opponent with the same privileges, even if it actively tries to verify its integrity because even the verification routines can be corrupted as well.

Nevertheless, reliably detecting user space hooks can still require significant effort, sometimes making their use worthwhile even though they effectively rely on security through obscurity.

4.2.2 Kernel space hooks

In certain contexts, in particular that of a sandbox for malware analysis, integrity and stealth are so important that relying on security through obscurity is unacceptable. Therefore, a better solution is desired.

As already explained in Chapter 2, user space programs can do very little without invoking system services. In Section 2.2 we see that at one point control is transferred to the kernel. To the program invoking the system service, the kernel handling its request is like a black box. As in user space, in the kernel there are plenty opportunities to place a hook. Naturally, hooks in the kernel operate in ring 0 and therefore are higher privileged than their counterparts in user space. These privileges can be leveraged to maintain integrity and stealth against opponents in user space.

Remember, there are two methods through which user mode programs can make system calls: `INT 0x2e` and `SYSENTER`. `INT 0x2e` is the legacy way of performing user to kernel mode transitions and is no longer used on modern x64 systems (although it can be enabled through modification of a registry key). This entails that IDT hooks, or hooks on functions and structures associated with interrupt dispatching, e.g. `KiInterruptTemplate` hooks by mxatone and ivanlef0u [57], nowadays are useless.

Today, system services are called with the faster `SYSENTER` instruction which does not have the overhead of an interrupt. `SYSENTER` causes the processor to jump to the address in `SYSENTER_EIP_MSR`. One could set the value of this register to the address of another handler, hooking the `SYSENTER` instruction. This is called a *SYSENTER hook*.

Alternatively, one could modify the lookup table that the default system service handler uses. In order words, perform SSDT hooking as described in Section 2.4.2.

Finally, one could dive much deeper in the operating system and place hooks to monitor specific activities. For example, one could place a minifilter driver in the filesystem driver stack to monitor reads and writes.

For a general-purpose sandbox, SYSENTER and SSDT hooks would be a better match as it allows for monitoring a broader range of actions. The SYSENTER hook has the advantage that only one place needs to be modified to gain control over the execution of all system services, but has the disadvantage that it can be bypassed by malware using the legacy `INT 0x2e` instruction. On the other hand, one modification per system service is needed when hooking the SSDT, but it ensures execution is redirected regardless of how the system service was invoked. Furthermore, one still has access to all parameters that were passed to the system service.

4.2.3 Virtual machine introspection

Virtual Machine Introspection (VMI) is a technique for monitoring the runtime state of a VM [13]. VMI can either take place from within the VM or from outside the VM. The former requires communication with an agent in the VM that monitors its environment. The latter may be implemented using some type of low-level information provided by the virtual machine monitor like raw bytes of the VM's memory.

When used for analyzing malware, implementing VMI outside of the VM has several advantages. Firstly, there is no need for a functional OS to run an introspection agent. This allows for monitoring even before machine has completely booted or when the OS hangs. Secondly, it can be implemented in a way that introduces no artifact in the VM at all, making it very difficult to detect. Lastly, it is strongly isolated from the guest it is monitoring. This gives it a high degree of attack resistance and allows it to continue observing with integrity, even after the guest is completely compromised. This means that it can reliably be used to monitor any type of malware - even kernel mode rootkits.

However, it also comes with several challenges. To begin with, the low-level view of the VM must be converted into meaningful information. Then there also is the issue that monitoring the guest is cumbersome and can be very resource-intensive.

Chapter 5

Zandbak

This chapter focuses on the product of this thesis: Zandbak. Firstly, we describe the design and explain the requirements set for Zandbak. Secondly, we describe the individual components, features and techniques that are used in the sandbox and provide details on their implementation. Thirdly, we show how these components are combined into the kernel driver that is Zandbak. Lastly, we provide a brief guide on setting up the system.

5.1 Design and requirements

The goal of Zandbak is to provide a system for automated behavioral analysis of malware. We intend for Zandbak to step in when analysis fails in other sandboxes due to evasive malware detecting the presence of a sandbox. Therefore transparency to prevent evasion is crucial and the behavior must be observed in-depth.

Accordingly, the requirements we set out for Zandbak are as follows:

1. **Security**: the damage programs can cause running in the sandbox must be limited;
2. **Stealth**: programs must be unable to distinguish if they are running on a plain machine or in Zandbak;
3. **Fidelity**: analysis must provide deep and accurate insight in the workings of the programs running in the sandbox.

Looking at other sandboxes, we see that scalability commonly is a requirement. With Zandbak, we chose to instead focus on increased fidelity. This allows us to forsake some performance and implement stack walking and snapshotting: valuable yet resource-intensive analytical features. In the remainder of this section we will explain Zandbak's design and how it satisfies these requirements.

5.1.1 Security

It is a standard assumption to consider the Virtual Machine Manager (VMM) trustworthy [17]. We rely on the VMM to keep the sandbox isolated from the rest of the system. Nevertheless, caution must be exercised as the past shows numerous vulnerabilities that allow for virtual machine escapes i.e. breaking out of the virtual machine and interacting with the host operating system.

Furthermore, the VM in which Zandbak runs should not be connected to computer networks, including the internet. The design decision to discard the networking aspect of malware analysis sandboxes was made to limit the scope of this thesis. Besides that, connecting infected machines (including sandboxes) to networks, puts yourself and others at risk and could jeopardize an investigation, as the malware could use the network to propagate and infect other machines, or tip the malware developer off that the malware is under analysis. Although having no network connection could be seen by the malware as an indicator of being in a sandbox, allowing access to the internet opens up a whole new world of possibilities for the malware to detect the sandbox' presence. To limit network connectivity as much as possible, there should only be a direct connection with the server that controls Zandbak. Keep in mind that this still exposes the server to the malware, so appropriate defensive measures must be taken.

5.1.2 Stealth

For Zandbak to stay under the radar, it is designed as a driver that resides purely in kernel space. The core principle behind its stealth relies on the difference in privileges between ring 0 and ring 3. Its stealth follows from user space malware simply lacking the privileges to access the resources that would reveal its presence. Furthermore, we hide existence of the driver itself by unlinking it from the list of drivers.

In this thesis, the detectability of the virtual environment is considered less important because Zandbak can also function on a plain machine. Nevertheless, some effort tweaking the virtual machine configuration to increase stealth of Zandbak as a product was still made. The details of this configuration are provided in Section 5.4 as part of the guide on setting up Zandbak.

5.1.3 Fidelity

For Zandbak to perform in-depth analysis of the malware that runs in it, we implemented a number of analytical features. The constraints of residing purely in kernel space severely complicate seemingly simple tasks such as process creation, reading process memory and moving files into the sandbox. Numerous library functions are readily available to accomplish such tasks in user space yet they cannot be used in kernel space.

Being purely in the kernel provides attack resistance against user space malware, allowing Zandbak to continue observing with integrity, even after user space is compromised.

We divided Zandbak's analytical features into three categories.

Standard sandbox functionality The techniques in this category are relatively standard in malware analysis sandboxes. As we did not want to reinvent the wheel, part of the code that implements these features is borrowed from existing projects. This category contains the following features:

- Hooking the SSDT to monitor system calls
- Callback for process creation
- Callback for registry actions

Existing techniques adjusted to fit in Zandbak This category includes techniques that have been applied in different contexts and had to be modified significantly in order to work in the kernel as part of Zandbak. The techniques we refer to are:

- Using WinSock Kernel for real-time communication between Zandbak and the server that controls it
- Starting samples through APC injection

Novel techniques These are techniques that, to the best of our knowledge, have not been applied in malware analysis sandboxes before or are applied in a new way. These features are:

- Real-time stack walking
- Snapshotting
- Infection scope tracking

To the best of our knowledge, there are no sandboxes that perform real-time stack walking. Neither could we find a kernel mode sandbox that creates snapshots of processes and would reveal the technique it used to accomplish this. Similarly, we were unable to find a sandbox that would monitor the spread of malware by monitoring system service invocations.

The decision to include these specific novel techniques in Zandbak stems from a discussion within the team, asking ourselves what features we would like to have or see improved in a sandbox.

5.2 Components and implementations details

In this section, we provide an in-depth view of Zandbak's features and their implementations.

As mentioned earlier, the sandbox targets the 64-bit version of Windows 10 (build 1709) and is implemented as a Windows Driver Model (WDM) driver. We chose WDM over the newer Kernel-Mode Driver Framework (KMDF) for two reasons. Firstly, it is lower level than KMDF and offers a higher level of control. This is essential for some of the features we wanted to implement. Secondly, WDM is more portable as it is supported on older systems whereas KMDF is only supported on newer versions of Windows. If one would want to port Zandbak to Windows XP, their task would be easier if it was written as a WDM driver.

We target this slightly older version of Windows 10 as it allows us to easily bypass PatchGuard without limiting the range of malware that we can analyze: (nearly) all malware that runs on the most recent version of Windows 10 is also able to run on build 1709. Targeting a specific build of Windows also eases development as it allows for hard coding offsets to fields in undocumented structures whose layout might change between builds.

5.2.1 SSDT Hooks

To monitor system behavior, our sandbox employs the SSDT Hooking technique as described in Section 2.4.2. In that same section, we explained the difficulties involved with hooking the SSDT on a modern installation of Windows. Indeed, the SSDT is read-only and is monitored by PatchGuard for changes. Circumventing PatchGuard was quite straightforward: we simply used UPGDSED [41] to disable it altogether. In the remainder of this section, we will describe our implementation of SSDT hooking.

Disabling Write Protection

There are two common approaches to bypassing the read-only restriction on the SSDT, which we shall both discuss briefly.

The naive solution that follows from reading Intel's developer's manual [46] is to unset the "Write Protection" bit in the CR0 register. The manual states:

> CR0.WP allows pages to be protected from supervisor-mode writes. If CR0.WP = 0, supervisor-mode write accesses are allowed to linear addresses with read-only access rights; if CR0.WP = 1, they are not. User-mode write accesses are never allowed to linear addresses with read-only access rights, regardless of the value of CR0.WP.)

This is not a very stable solution however. Each CPU core has its own set of control registers, and we only unset the WP bit of the current core: write protection still is in effect in the other cores. The scheduler can switch the thread to another core at any moment, after which writing to read-only memory would trigger a page fault.

Fortunately, a more stable solution exists - one that is implemented in TitanHide, a driver that is intends to hide debuggers from certain processes [60]. The operating system uses a memory descriptor list (MDL) to describe the physical page layout for a virtual memory buffer. TitanHide allocates an additional MDL for the physical memory in which the SSDT resides and locks the describes pages in RAM to prevent paging. Subsequently, it maps the locked pages to a virtual address range for which there are no restrictions regarding access rights. We chose to adopt TitanHide's solution because of its increased stability.

Finding the SSDT

Now that we are capable of modifying the SSDT, we are left with the issue of finding the SSDT in the first place. In x64 versions of Windows, the address of the SSDT is not exported by NTOSKRNL, so we will need to locate it ourselves through memory scanning.

In Section 2.3, we learned that the system service dispatcher, `KiSystemService`, uses the SSDT to redirect code flow to the correct system service. Therefore, at some point, the system service dispatcher must reference the location of the SSDT. Indeed, the routine contains an instruction that loads the relative address of the SSDT into register R10. TitanHide exploits this knowledge and scans the entire kernel space memory using a signature to find the instruction, after which it extracts the offset from the instruction. Trying not to reinvent the wheel, our solution is based on the one used in TitanHide.

Creating the hook

To create the hook, we define a function with the same parameters as the original. Then we modify the corresponding entry in the SSDT such that it points to the hook. An important detail of the SSDT on x64 systems is that it encodes the function's addresses as offsets from the base of the SSDT rather than absolute addresses.

To make sure the sure offset of the hooks fits in the SSDT, we make use of a hook stub. This hook stub moves the target address into the CPU's instruction pointer register via the stack and consists of the following instructions:

```
movabs  rax, hookAddr   // Moves function pointer into register rax
push    rax             // Pushes function pointer to the top of the stack
ret                     // Pops function pointer into the instruction
                        // pointer
```

We scan the memory behind the SSDT in search for a code cave: a writable and executable region in memory in which executable code can reside. Finding one is relatively easy as the hook stub is only a few bytes in size.

5.2.2 Snapshotting

We register a callback for process creation and termination using `PsSetCreateProcessNotifyRoutineEx`. The operating system calls the registered callback whenever a new process is created or when the last thread of a process is about to exit. We seize this opportunity to create snapshots of the process:

- On process creation, we do not snapshot the process' memory right away. Instead, we create a new thread that waits for a configurable amount of time, which we currently fixed at 50 milliseconds, while the process continues to initialize, after which it creates a copy of the process' memory. The delay gives the process some time to start and initialize data structures.
- Similarly, we create a snapshot of the process' memory right before it exits. This time, however, we take the snapshot right away while stalling the last thread of the process to postpone process termination and the following cleanup.

To create a snapshot, we employ the following procedure. We start by opening a handle to the process we want to snapshot using `ZwOpenProcess`. Then, we attach to the address space of the process using `KeStackAttachProcess`. Next we want to enumerate all allocations made by the process. We can do so using `NtQueryVirtualMemory`. Provided a `BaseAddress`, this function returns a `MEMORY_BASIC_INFORMATION` structure describing the allocation (range of pages) that address is in. By providing a `BaseAddress` of 0, we can obtain an initial `MEMORY_BASIC_INFORMATION` structure. Although this region is not actually used by the program, it still exists albeit in a `MEM_FREE` state, allowing us to gather the information to find allocations that are in use. The `MEMORY_BASIC_INFORMATION` structure is defined as follows:

```
typedef struct _MEMORY_BASIC_INFORMATION {
  PVOID   BaseAddress;
  PVOID   AllocationBase;
  ULONG   AllocationProtect;
  SIZE_T  RegionSize;
  ULONG   State;
  ULONG   Protect;
  ULONG   Type;
} MEMORY_BASIC_INFORMATION, *PMEMORY_BASIC_INFORMATION;
```

The RegionSize field is particularly helpful in finding the next allocation. We simply add that value to the BaseAddress we used previously and call NtQueryVirtualMemory again. This ought to return information on the next allocation, allowing us to repeat the process until NtQueryVirtualMemory returns a STATUS_ACCESS_VIOLATION error code, which means we finished traversing all allocations.

The state of an allocation is denoted in the State field and must be one of three values: [54]

1. MEM_COMMIT: indicates committed pages for which physical storage has been allocated, either in memory or in the paging file on disk;
2. MEM_FREE: indicates free pages not accessible to the calling process and available to be allocated;
3. MEM_RESERVE: indicates reserved pages where a range of the process' virtual address space is reserved without any physical storage being allocated.

Processes cannot store data in allocations that are in a MEM_FREE or MEM_RESERVE state, so it is only necessary to copy data of allocations that are actually in use i.e. in the MEM_COMMIT state.

Copying data from a process' virtual address space to kernel space is not very straightforward. The most stable way would be to first map the allocations to kernel address space, check if the buffer resides in the user portal of address space and is correctly aligned, lock the relevant pages, hoping that the process has not terminated yet, and subsequently copy a portion of the memory. This can be achieved using a combination of IoAllocateMdl, MmProbeAndLockPages, MmMapLockedPagesSpecifyCache and RtlCopyMemory. We opted to sidestep this by using the undocumented MmCopyVirtualMemory routine. It is much easier because it serves as a wrapper around the steps above and is very stable, but has the downside that it is undocumented and thus might break at any time in the future when Windows updates.

Another interesting field of the MEMORY_BASIC_INFORMATION structure is Allocation Protect. It indicates what the permissions (read, write, execute, etc.) of the allocation are.

The snapshots or memory dumps can contain valuable information by themselves: encryption keys, IP addresses and any other interesting data that might reside in them. However, most information in these memory dumps is constant and likely to be less interesting than parts that change over the course of the process' execution. By computing the difference between the creation and termination snapshots, we can highlight parts in memory of increased importance.

5.2.3 Logging

An important aspect of every sandbox for malware analysis is its ability to store recorded behavior. We considered two candidate architectures:

1. A standalone driver that collects information, performs additional processing, and yields logs afterwards;
2. A Client-server model where the client simply collects and transfers data to the server, that in turn will perform the other tasks.

We eventually opted for a client-server model because it would require fewer computations at client-side, reducing the analysis' impact on performance. Besides that, it comes with the advantage that the collected data can be processed at server-side, in a high-level programming language which arguably is easier than the subset of C++ that is used for driver development. Moreover, this solution eliminates a potential limit on the size of the recorded data. If we were to store everything in the sandbox, it would need to fit in RAM as writing to disk would generate artifacts that can be detected by the malware, compromising ZANDBAK's stealth.

For connecting to the server, we use Winsock Kernel, a kernel mode Network Programming Interface (NPI). It is an interface with which network I/O operations can be performed from the kernel. It is similar to user space NPIs as it allows for socket creation, binding, data transfers, etc., except for one major difference which is that it is all asynchronous.

Initially, ZANDBAK creates one socket and connects with it to the server. The server can then use this connection to upload files and send commands to ZANDBAK. Then, several more threads are created - each with their own socket. These are all dedicated to sending the contents of the message queue. Experimentally, we found that 10 threads are can easily to keep up with the rate at which the message queue gets filled.

At first, whenever an event was logged, it was sent to the server right away in a blocking manner. This meant that the system continuously had to wait for network operations to complete before continuing execution. Unsurprisingly, this led to severe performance issues. Now, whenever there is data to send or behavior to log, a message is created and then pushed to the start of the message queue, after which the thread can continue execution right away.

The message queue is implemented as a double-ended queue (deque) with fixed size and was added to ZANDBAK in order to improve performance. We thought a deque would be the right data structure because, when used a queue, First-In-First-Out behavior results, which in turn leads to messages more or less being sent in order. We actually had to implement the double-ended queue data structure ourselves, as existing implementations are not available in kernel space. A mutex is used to synchronize accesses to the message queue.

The messages use a Type-Length-Value (TLV) encoding scheme and consist of two parts: the header and the content.

```
struct MESSAGE {
  HEADER   Header;
  UINT8*   Content;
};
```

Upon transmission, the regions in memory that these pointers reference are copied into a contiguous buffer.

The header consists of three fields: the length of the message's content, the type of the message and the time at which the message was created.

```
struct HEADER {
  ULONG           MsgLen;
  UCHAR           Type;
  LARGE_INTEGER   SystemTime;
};
```

The message type in the header indicates how the information in the content of the message, which is simply a series of bytes, is structured. For example, a message of the `PROCESS_CREATION_TYPE` would take the following structure in place of the `Content` field:

```
struct PROCESS_CREATION_CONTENT {
  ULONG ParentId;
  ULONG ProcessId;
};
```

This message is used when a new process is created and contains the process IDs of the parent and child respectively.

To receive a message, the server first reads 5 bytes from the connection, containing the type and length of the message. This provides the server with all the information it needs to retrieve the remainder of the message and to parse it. As of now, the server simply prints the received messages.

5.2.4 Sample deployment

To allow automation and improve scalability, Zandbak facilitates in the automated deployment of malware samples. To that end, we need to programmatically copy a binary to the analysis guest and then execute it. Unfortunately, Microsoft provides no supported method for starting a (user space) process from within the kernel at all. Therefore, we rely on a process injection technique often used by malware called *APC Injection*. We will explain this technique later on. This section describes how samples can be deployed to an existing Zandbak setup. Instructions on setting up Zandbak itself are provided in section 5.4.

Copying the binary

To copy the binary from the server to the sandbox, we added a command that does so using the already established *command connection*. Its implementation is rather straightforward.

```
NAME
    upload - upload a file to the sandbox

SYNTAX
    upload SANDBOX_PATH LOCAL_PATH

DESCRIPTION
    Uploads the local file found at LOCAL_PATH to the sandbox where it will
        be stored at SANDBOX_PATH.
```

The server sends a message containing `SANDBOX_PATH` and the file's data. The sandbox receives this, creates a new file at `SANDBOX_PATH` and stores the received data in it.

APC Injection

Asynchronous Procedure Calls (APCs) can direct a process' thread to execute code prior to executing its regular execution path [29]. Each thread has its own APC queue. Whenever a thread is in an alertable state, the contents of its APC queue are processed (executed) in the context of the thread. A thread can enter an alterable state in various ways, for example when functions such as `WaitForSingleObjectEx`, `WaitForMultipleObjectsEx` and `Sleep` are called. Although APCs can be injected in any thread, using one that already is alertable is a simple way to ensure timely processing of the APC.

An APC must be initialized using the undocumented `KeInitializeApc` routine shown in the listing below:

```
NTKERNELAPI
NTAPI
VOID KeInitializeApc (
    PRKAPC              Apc                 // Structure to be initialized
    PRKTHREAD           Thread              // Thread whose queue the APC
                                            // will be inserted into
    KAPC_ENVIRONMENT    Environment,        // Thread environment to be run in
    PKKERNEL_ROUTINE    RundownRoutine,     // Executed in kernel mode when done
    PKNORMAL_ROUTINE    NormalRoutine,      // Executed when APC is processed
    KPROCESSOR_MODE     ProcessorMode,      // Run in UserMode / KernelMode
    PVOID               NormalContext       // Argument for NormalRoutine
);
```

After initialization, we can use the undocumented `KeInsertQueueApc` routine to insert the APC into any thread's APC queue.

`explorer.exe`, or File Explorer, is an application that is shipped with all releases of Windows from Windows 95 onwards. It provides a GUI for accessing file systems and is responsible for many user interface elements, such as the desktop. This application is the perfect candidate for APC injection as it runs on (nearly) all systems and has plenty of alertable threads.

Using `NtQuerySystemInformation`, we are able to obtain a list of `SYSTEM_PROCESS_-INFORMATION` structures - one for each process that is currently running. We loop over this list until we find the one that belongs to `explorer.exe`. We can recognize this entry by checking if the value of the `ImageName` equals `explorer.exe`. Once we find the correct structure, we iterate its array of `SYSTEM_THREAD` structures until we find one that has the alertable flag set.

Now that we have found a target thread to inject, we can start preparing the APC. Remember, the goal of the APC is to get the target process to execute our code, which we provide as shellcode. Getting the shellcode to run is a three-stage process.

First, we customize the shellcode by replacing the path `placeholder` variable with the null-terminated path to the target executable. Since the APC will run in user mode in the context of the thread, the shellcode must be made available there. To do so, we inject a kernel mode APC which allocates a page of memory with read/write/execute permissions in the process' memory space using `NtAllocateVirtualMemory`. Afterwards, we copy the shellcode there. Now the shellcode resides in the process' memory and we have a pointer to it. Finally, the kernel mode APC injects a user mode APC, passing the pointer to the shellcode as the `NormalRoutine` parameter such that it will be executed when the APC is processed, causing the target executable to start.

To perform APC injection and start a program, one can use the “run” command:

```
NAME
     run - run a file in the sandbox

SYNTAX
    run SANDBOX_PATH

DESCRIPTION
    Performs APC injection on explorer.exe in order to start the program that
         is stored in the sandbox at SANDBOX_PATH.
```

File execution shellcode

Shellcode refers to a chunk of machine code that is capable of being executed from an arbitrary location in memory, and without relying on services provided by the operating system as a normal executable would. Historically, such chunks of code were used to spawn a shell from which the attacker could control the machine, hence the name “shellcode”. Today, shellcode is more likely to download and execute another program than spawn a shell, but the term remains.

The goal of our shellcode is to start an executable given the file’s path. When writing a program that performs this task, one would simply use a Windows API function such as `WinExec` or `CreateProcessA` and refer to them by name. When the program is compiled and linked, the linker puts information in the resulting executable that ensures relevant DLLs are loaded and that addresses of called functions are resolved.

In Windows, a program can load additional DLLs and lookup functions they contain at runtime using the `LoadLibraryA` and `GetProcAddress` APIs in `kernel32.dll`. For the shellcode to be independent of its host program, we need to use those functions for all API functions it uses. Luckily, we only need `WinExec` which is part of `kernel32.dll` - a library that always is loaded in the address space of every process. To locate this function, we use the `lookup_api` procedure described in a blogpost by McDermott Cybersecurity [38]. In this blogpost, the author solves the “catch-22” of needing `GetProcAddress` to resolve the address of `GetProcAddress` by implementing their own version of that function. Like the “real” `GetProcAddress`, it traverses the data structures of a DLL in memory, in particular the `IMAGE_EXPORT_DIRECTORY`, and fetches the address of the function whose name we provide.

The following listing is an excerpt from the shellcode that executes the program that the path in the placeholder variable points to:

```
main proc
    enter   28h, 0
    mov     r15, 0fffffffffffffff0h
    and     rsp, r15

    lea     rdx, winexec
    lea     rcx, kernel32_dll
    call    lookup_api

    mov     rdx, 5
    lea     rcx, placeholder
    call    rax

    leave
    ret
main endp

kernel32_dll    db  'KERNEL32.DLL', 0
winexec         db  'WinExec', 0
placeholder     db
        'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA',
        'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA',
        'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA',
        'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA',
        'AAAA', 0
```

In the prologue, we allocate some space on the stack and make the stack 16-byte aligned. Afterwards, we use the API to resolve the address of `WinExec` in `kernel32.dll`. Finally, we make a call equivalent to `WinExec(placeholder, SW_SHOW)` and subsequently return.

Note that the placeholder variable points to a null-terminated string of 260 A's. 260 coincides with the maximum length for a path in Windows, `MAX_PATH`, which is defined as 260 characters. As we already mentioned in the previous section, the kernel mode APC modifies the generated machine by replacing the sequence of A's with the path to the target binary such that `WinExec` is called with the actual path.

Self-evidently, `lookup_api` can resolve the address of `LoadLibraryA`. Using calls to this function, we can load additional DLLs expanding the repertoire of APIs to call. For example, we can make the shellcode show a message box using the following sequence of operations:

- lookup `LoadLibraryA` from `kernel32.dll`;
- use `LoadLibraryA` to load `user32.dll`;
- lookup `MessageBoxA` from `user32.dll`;
- prepare arguments and call `MessageBoxA`.

5.2.5 Stack Walking

Zandbak monitors the system at the level of system service calls. Looking only at what system services a program invokes, results in limited insight in what the program is trying to achieve. By considering the arguments that are passed to the system service, we can gain additional insight, but we aim to go even further than that.

If we inspect the user mode stack of the process invoking the system service, unwinding the stack frames, we can reconstruct the program flow that caused it end up invoking the system call. Indeed, unwinding the stack allows us to see what series of function calls lead to the invocation of the system service. Depending on how parameters are passed, we can sometimes determine what arguments are passed to functions within the program and to the Windows API.

On x64 Windows, two types of processes can run: 64-bit processes that run natively and 32-bit processes that run using the WOW64 emulation layer. Walking the stack requires an approach specific to the type of process. Because of limited time that was available for the completion of this thesis, Zandbak only supports stack walking for 64-bit processes. In the remainder of this section, we will take a closer look at the implementation of this feature.

Stack unwinding for x64 processes

Traditionally, walking the stack frames of a process was as simple as traversing a linked list. Consider the typical x86 function prologue as produced by GCC:

```
push ebp
mov  ebp, esp
sub  esp, N
```

This prologue:

- pushes the current base pointer onto the stack, so it can be restored later;
- sets the base pointer to the stack pointer, such that the new frame starts at the bottom of the previous one;
- make room in the current stack frame for local variables.

At the end of the function, this prologue is followed by an epilogue that reverses the actions of the prologue:

```
mov  esp, ebp
pop  ebp
ret
```

The actions are reversed by:

- setting the stack pointer to the end of the previous frame;
- popping the saved base pointer back into the base pointer register;
- jumping to the program counter that was saved by the function that called this function.

As we illustrate in Figure 5.1, such prologues allow for easy stack frame traversal using EBP chaining. This technique is unfortunately ineffective against x64 processes because Windows x64 adheres to a new software convention titled “x64 Software Conventions” [56]. Under the x64 Software Conventions, traditional use of the RBP (the 64-bit version of EBP) is completely optional. By default, a function accesses local variables as an offset from the top of the stack, pointed to by the RSP register. As a result, RBP is no different from the other normal non-volatile general-purpose registers.

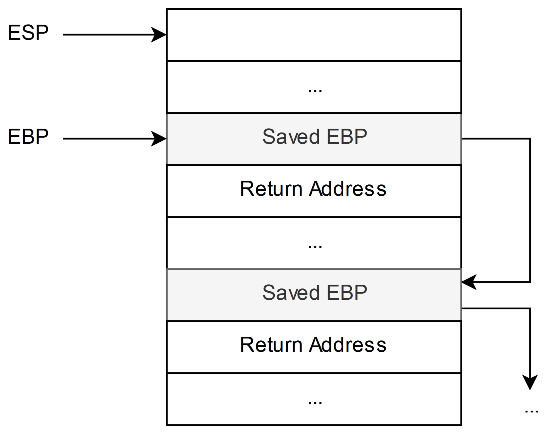


Figure 5.1: The stack layout of a x86 process.

Nevertheless, the PE32+ file contains metadata to support structured exception handling (SEH): the native exception handling mechanism for Windows. When an exception occurs, the stack is linearly searched for an exception handler, and all entries before the function with the exception handler are discarded from the call stack. So, in cases where the exception is not handled in the same function as it was thrown, unwinding of the stack takes place.

Zandbak makes use of the same structures that SEH uses to unwind the stack: `RUNTIME_FUNCTION`, `UNWIND_INFO` and `UNWIND_CODE`. The relation between these structures is highlighted in Figure 5.2.

For each function chunk in the image, a `RUNTIME_FUNCTION` chunk is prepared. The structure holds three relative virtual offsets (RVAs): offsets from the base of the image in virtual address space. Two indicate the start and end of the function chunk and the other indicates `UNWIND_INFO` struct regarding that function chunk can be found. This information is stored in the `.pdata` section of the PE file which consists of an array of `RUNTIME_FUNCTION` structures and is sorted by function chunk address.

The `UNWIND_INFO` structure is used to record the effects of a function on the stack pointer and where the nonvolatile registers are saved on the stack. Most importantly, it contains an array of `UNWIND_CODE` structures. This array precisely describes how and where in the function chunk the stack pointer and nonvolatile registers are manipulated. For example, the `UNWIND_CODE` of type `UWOP_PUSH_NONVOL` describes a push of a nonvolatile register and the `UWOP_ALLOC_LARGE` code describes an allocation on the stack. In order to unwind a stack frame, one needs to interpret the `UNWIND_CODE` structures and apply their inverse. Doing so restores not only RSP, but some other registers as well. While unwinding the stack, one can collect a bounty of information regarding previous intermediate states of the program by looking at the restored registers and the local variables of functions.

Function names

By unwinding stack frames, we obtain a series of return pointers. Programs frequently make use of external libraries that offer high-level APIs, for example the Windows API, and rarely make direct system calls themselves. As a result, a substantial portion of the return pointers will point to shared libraries. Given the base address of each shared library in a program’s memory, we can easily pinpoint which shared library a return pointer points to,

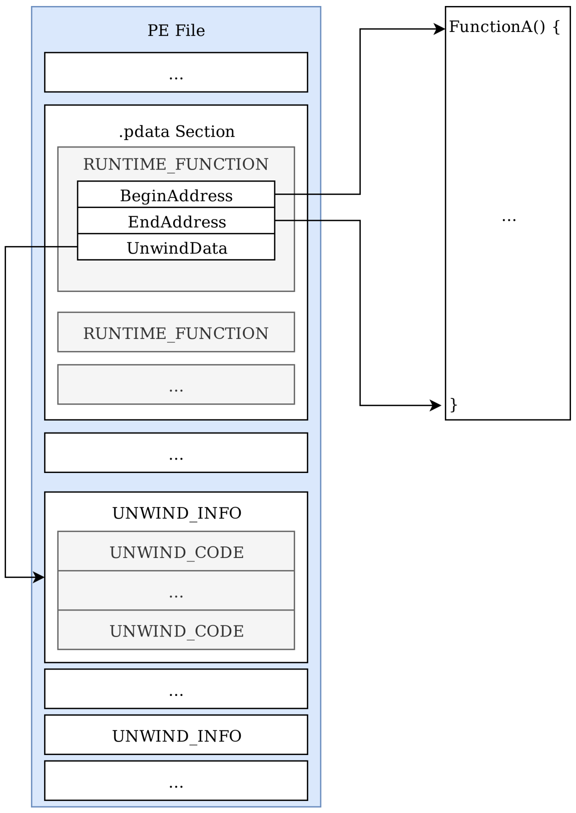


Figure 5.2: The relation between SEH related structures and their layout in the PE file.

and rewrite each return pointer as an offset from the start of their respective library.

Certain PE files, for example shared libraries, contain an export table: a table with information about the functions that the PE file makes accessible to other programs. Using the `.pdata` section, we can usually find in which chunk the return address resides. If by coincidence, the chunk is at the start of an exported function, we can simply query the export table to find a function name. Unfortunately functions often consist of multiple chunks and there is no straightforward way to figure out which groups of chunks constitute a function. Furthermore, only a subsection of the function in a shared library is exported. Therefore, we need to look for another way of obtaining symbols.

For some shared libraries, especially those shipped with Windows by default, debugger symbols are available from which information such as function names and parameter types can be extracted. With this information we can write a return pointer as an offset from the start of a function in an image. To further clarify the differences in representation, an example is shown in Table 5.1.

Format	Example
Raw return pointer	0x7fffca48 d7fee0b0
Offset from image	ntdll.dll+0x1234
Offset from function in image	ntdll.dll!RtlCreateUserProcess+0x37

Table 5.1: The different representations of a return pointer.

Implementation

To implement these techniques into ZANDBAK, we first need to locate the user space stack. When a user-to-kernel mode transition occurs, all registers are saved in the threads execution context structure. This structure is called `KTRAP_FRAME` and is accessed through the `TrapFrame` field of the undocumented Thread Control Block (TCB) of the thread invoking the system service. Indeed, the execution context contains the value of the stack pointer right before the user-to-kernel transition, revealing the top of the user mode stack.

The top of the stack contains the return pointer to the function that invoked the system call. For this address, we can look up the corresponding `RUNTIME_FUNCTION` structure by means of a binary search and subsequently unwind the stack frame. Now RSP points to the function's caller, whose stack frame we can now unwind. We repeat this process until we find an invalid return address, such as a null pointer.

This provides us with enough information to represent the call trace in terms of return pointers. For a richer representation we need to gather additional information.

As we mentioned earlier, a program rarely invokes system services itself, but rather uses libraries and APIs that do so, such as `ntdll.dll`. To find the right `RUNTIME_FUNCTION` structure, the `.pdata` sections of those images must be queried as well.

To enable this, ZANDBAK keeps track of some metadata. Using `PsSetLoadImageNotifyRoutine`, we registered a callback that is triggered whenever a process loads an image (e.g. .dlls), including its own image. In this callback, we store:

- a copy of that image;
- the base address of that image;
- the process it belongs to;
- the path to the image.

It is important to create a copy of the image rather than the version a process has in memory, which can be tempered with at runtime in an effort to trick us.

Using the information in these structures, we can determine which image a return pointer points to, and its offset within that image. To actually find a function name, we aim to use debugger symbols. There is however one major hurdle we need to overcome: the DbgHelp library with which one normally retrieves debugger symbols works in user space only.

We looked for ways to use this library anyway, but all involved some kind of user space component, which is an unacceptable compromise regarding ZANDBAK's stealth.

As the target OS is fixed (Windows 10 x64 build 1709), we can assume that the shared libraries in the Windows directory are relatively static. Therefore we built a user space application that generates a lookup table in the form of a (90.000+ lines) C++ header file, which is then included in ZANDBAK. We generate this lookup table as follows:

1. Recursively list all .dll files in the Windows directory;
2. For each .dll file, query symbols for each function chunk in the .pdata section;
3. Combine adjacent chunks that belong to the same function;

4. Generate the corresponding part for the lookup table in the C++ header.

Using this lookup table, we can give a detailed representation for return addresses that reside in system libraries. To illustrate Zandbak's final version of stack walking with an example, in Figure 5.3 we show a stack trace that is generated during the startup of the `winver.exe` utility. This stack trace was started in Zandbak's `NtReadFile` hook and shows how `gdi32full.dll`, which is part of the Microsoft Graphics Device Interface, attempts to draw text on the screen but needs to load a font first.

```
00007FFF417C2C66   \Windows\System32\KernelBase.dll!ReadFile+0x76
00007FFF4093576A   \Windows\System32\gdi32full.dll!CreateFontIndirectExW+0x51a
00007FFF4093986C   \Windows\System32\gdi32full.dll!GetObjectA+0xb3c
00007FFF40953322   \Windows\System32\gdi32full.dll!LpkDrawTextEx+0x2ad2
```

Figure 5.3: Example stack trace of `winver.exe` loading a font file.

5.2.6 Infection tracking

Process injection is a widespread defensive technique employed by malware that entails injecting and running pieces of their own malicious code in other processes. Using process injection, a malware infection can spread from one process to the next. Malware can apply this technique to hide that it is running code and to bypass security filters. For example, if the web browser is whitelisted to send traffic over the network, the malware inject code into the web browser and piggyback on its privileges to communicate with a C&C server over the network.

It is important to keep track of what processes are infected, such that the scope of analysis can be adjusted and the malware continues to be monitored.

Zandbak's strategy is based on the assumption that malware, at some point, needs a handle to a process / thread in order to inject itself. To this end, three rules were designed with which Zandbak tracks potential spread of the malware:

1. If infected process X creates process Y, process Y becomes infected;
2. If infected process X opens a handle to process Y, process Y becomes infected;
3. If infected process X creates a thread in process Y, process Y becomes infected.

These rules were implemented by hooking the kernel mode functions `NtCreatePro cess(Ex)`, `NtOpenProcess` and `NtCreateThread(Ex)` respectively.

5.3 Implementation

The Zandbak malware analysis system comprises of two main components:

1. an analysis VM running a WDM driver;
2. a server for communicating with the driver.

In Section 5.3.1 we look at how the components described in Section 5.2 are combined into a single WDM driver. Then, in Section 5.3.2, we look at how the Python server is implemented and communicates with the driver.

5.3.1 WDM Driver

WDM drivers are usually written in C or C++. ZANDBAK is primarily written in C++ with small parts in C and assembly. However, when using C++ for kernel development, one must take into account that many of the language's advanced and even basic features cannot be used. In the kernel development commmunity, it is common knowledge that a number of C++ features can introduce more problems than they can solve and therefore should be avoided. For starters, the following features are absent or should be avoided:

- exception handling
- constructors and destructors
- templates
- inline functions
- virtual functions, multiple inheritance and class deriviation in general

Additionally, when kernel mode programming, one must be very careful in the use of memory: reading from a paged-out region in memory already causes a blue screen showing the `PAGE_FAULT_IN_NONPAGED_AREA` error. Furthermore, one must be wary of the Interrupt ReQuest Level (IRQL). At any given time, Windows runs at a specific IRQL that determines what interrupts are enabled at that time. Code that is running at elevated IRQL has some restrictions on what it can and cannot do. Failing to comply with these restrictions cause blue screens, this time showing the `IRQL_NOT_LESS_OR_EQUAL` error.

To troubleshoot such problems, one can use a kernel debugger, which typically requires a second computer. For other versions of Windows, this requires special cables and has slow performance. From Windows 8 onwards, there is support for kernel debugging over a network connection, simplifying kernel debugging using virtual machines. Our setup uses two virtual machines where the VM that is being debugged connects over an internal network to the other VM running the WinDbg kernel debugger. In our experience, this has very workable performance.

The amount of information about Windows kernel development that can be found is rather limited. The Microsoft Developer Network (MSDN) and Windows Internals books [26] offer high quality information about the documented part of the kernel. However, when taking a deep dive into the undocumented part of the kernel, for example to implement APC injection, these sources offer little help. Instead, we have search online, only to find game hacking sites and obscure Russian / Chinese hacker forums that usually contain unreliable, low quality information.

To a WDM driver, the `DriverEntry` function is what the `main` function is to a "normal" C++ program: the point at the start of execution where control is handed over to the programmer. Drivers often remain in the background and only become active when certain events occur. In `DriverEntry`, initialization takes place and the actions that a driver takes in certain situations are registered. This can be done, for example, by registering callbacks and configuring handlers to I/O request packets that are used to communicate with the driver. In case of ZANDBAK, roughly the following actions take place in `DriverEntry`:

1. Handlers to I/O requests are configured;
2. `ntdll.dll` is read from disk so that it can be used for exports;
3. A number of undocumented functions gets manually imported;
4. Network connections are established with the server;
5. SSDT hooks are placed;
6. Callbacks are registered for process creation, process deletion, images being loaded and actions that modify the registry;
7. The driver is unlinked from the list of drivers.

The driver then continues to linger in the background and is activates momentarily when either a callback is triggered, it receives a command from the server or one of the functions it hooked is called.

5.3.2 Python server

On another machine, the server is running with which the driver communicates. It is a relatively simple application written in Python and consists of three files: `ZandbakServer.py`, `Logger.py` and `Commander.py`.

The application is started by running `ZandbakServer.py`. When started, it creates two sockets: one for sending commands to the sandbox and one to receiving data back from the sandbox. Both sockets are handled in their own thread and use the `struct` library to pack and unpack messages.

The thread for sending commands waits for command line input. Upon receiving input, it translates it to a corresponding command packet using `Command.py` and sends it to the sandbox.

The thread for receiving data waits for the sandbox to send a message. This is where all message queue handling threads in the driver connect to. It first reads and parses the header, telling it the type and length of the message. The remainder of the message is received and passed to `Logger.py` along with the type of the message for further processing. By means of a jump table, control is then handed over to the correct handler for that type of message. The logger unpacks the message into its individual fields and subsequently constructs and prints a formatted string using those fields.

5.4 Setting up Zandbak

In this section, we show how the Zandbak malware analysis system is set up.

5.4.1 Step 1: Preparing VirtualBox

For Zandbak to remain undetected, the presence of the virtual machine it runs in must be hidden as well. In this thesis, we made use of Oracle VM VirtualBox for virtualization. By default, it is very easy for malware to detect that it is running in a VirtualBox VM.

Depending on the OS running on the host, different options for hardening the VirtualBox installation are available. Linux users can use VBoxHardening by Cisco's Talos Group [43].

VBoxHardening is a collection of scripts and binary files that makes changes to the source code of a VirtualBox installation, making VirtualBox hard to detect by malware.

Windows users can use VBoxHardenedLoader instead [44]. VBoxHardenedLoader uses a driver that patches VirtualBox DLL files at runtime. Furthermore, it includes a script that patches the virtual machine's files.

We opted to use Windows because VBoxHardenedLoader is more mature and still actively being developed whereas VBoxHardening has been abandoned for about two years. To use VBoxHardenedLoader, we simply followed the instructions provided the installation guide in their GitHub repository.

5.4.2 Step 2: Configuring the VM

We create a VM with 2 cores, 4 GB RAM and a 120 GB HDD for ZANDBAK to run in. These hardware specifications are chosen to successfully bypass certain anti-virtualization checks in Al-Khaser, which are discussed in detail in section 6.2. We further configured the VM as described in step 2 of VBoxHardenedLoader's installation guide.

Afterwards, we install Windows 10 (build 1709) on the VM and disable Windows Update. Next, we install several applications, runtimes and utilities using Ninite. This serves two goals:

1. it makes the VM resemble a normal machine;
2. it might help malware samples run as they might have some of that software as a dependency.

The following software was installed: Chrome, Firefox, x64 Java Runtime, .NET 4.8, Silverlight, LibreOffice, Dropbox, OneDrive, 7-Zip, WinRAR, Skype, Thunderbird, Paint.NET, GIMP, IrfanView, Python, FileZilla, Notepad++, VLC, TeamViewer 14, ImgBurn and TeraCopy.

Finally, we run UPGDSED [41] to disable PatchGuard, which will allow ZANDBAK to make the changes necessary in the kernel to monitor malware.

5.4.3 Step 3: Starting analysis

The analysis guest is now ready to start analysis. First, we must start the server on the host by running `python ZandbakServer.py`. Next, we open the OSRLoader on the guest to load and start ZANDBAK's driver. If the IP address and port of the server are correct, either by changing the source code or changing the host's settings, the driver and server should now be connected. Using the `upload` and `run` commands in the server's command-line interface, we can now deploy a malware sample for ZANDBAK to analyze.

Chapter 6

Evaluation

This chapter consists of two parts. In the first part (Sections 6.1 and 6.2), we perform a series of experiments. The first experiment highlights ZANDBAK's infection tracking abilities. The second experiment is used to evaluate ZANDBAK's stealth. The second part (Section 6.3) involves a case study where we use ZANDBAK to recover the encrypted configuration file of an implant of the PlugX malware family.

With these experiments and the case study, we want to verify that ZANDBAK indeed:

1. adjusts the scope of analysis accordingly when other processes get infected;
2. passes anti-sandbox checks;
3. can obtain unpacked / decrypted malware components from memory.

6.1 Experiment 1 - detecting process injection

As is explained in Section 5.2.6, process injection is defensive technique commonly used by malware. Injecting code spreads the infection from one process to the next, which means that ZANDBAK's scope of analysis must be expanded to include the newly infected process.

For this first experiment, we perform various process injection techniques to see if ZANDBAK can keep track of the potential spread of the malware. We reuse code from the Al-Khaser, Windows-Process-Injection and InjectProc projects [58, 37, 48] on GitHub to perform the following types of injections:

- Classic DLL injection [37];
- PE injection [37];
- Process hollowing [48];
- APC injection [58].

These techniques are popular among malware developers and differ substantially from one another, forming a varied selection of injection techniques.

In this thesis, we will describe these methods of process injection only superficially. For a more detailed explanation, we refer to an excellent blog post by Endgame [45] and the source code in the GitHub repositories that implements these techniques [58, 37, 48].

6.1.1 Classic DLL injection

With this injection technique, a new thread is created remotely in the target process. The entry point for this thread is set to the address of `LoadLibrary` and the parameter is set to the DLL we want to inject. This causes the specified DLL to be loaded in the target process, executing its `DLLMain` function.

The sample we prepared for this experiment injects `HelloWorldDLL.dll` into the process of the Firefox web browser. We sent `ClassicDLLInjection.exe` and `HelloWorldDLL.dll` to ZANDBAK with the `upload` command and started Firefox. Next, we used the `run` command to start `ClassicDLLInjection.exe` and indeed a message box showing "Hello World!" appeared, indicating a successful DLL injection.

Upon inspection of logs sent to ZANDBAK's server, we note that the malware's spread was detected. The logs show that ClassicDLLInjection's process opened a handle to Firefox's process on multiple occasions and used `CreateThreadEx` to create a new thread in that process. Both events are expected considering to steps of the DLL injection and were registered by ZANDBAK as potential spreading of the malware, causing Firefox's process to be flagged as infected.

The logs also show that before it infected Firefox, the program tried to open a handle to `conhost.exe`, and that when it failed, it started `conhost.exe` itself. The same behavior also showed up for the other process injection techniques we tested. As it turns out, the `conhost` process brokers all GUI activity on behalf of non-GUI applications [53]. Therefore, it makes sense that the console applications that implement the injection techniques interact with it.

6.1.2 PE injection

Here, the malware allocates memory in a target process and subsequently relocated its own image into the target process. Then, a new thread is created in the target process with the `startAddress` parameter set to a (relocated) function address that is intended to execute.

We configured our sample to inject itself into `notepad.exe`. The sample contains a function that spawns a message box showing "Hello World". After injection, it computes the new address of that function and creates a thread in `notepad.exe` that executes that function. We copied PEInjection.exe to the target machine with `upload` and started Notepad. Then, we executed the sample with the `run` command, and the message box appeared.

The logs showed similar results as in the case of classic DLL injection, except for the fact that Notepad was being manipulated instead of Firefox. In these logs, we see that PEInjection's process opened a handle to Notepad's process on multiple occasions and that it used `CreateThreadEx` to spawn a new thread in Notepad's process. These events too were registered as the malware spreading to Notepad's process.

6.1.3 Process hollowing

A bootstrap application creates an innocent process in suspended state, for example notepad.exe. Then, it unmaps the process' entire image and replaces it with a malicious image that is to be hidden. The new image is rebased if its preferred image base does not coincide with that of the original image. Finally, the thread is set to continue execution at the new entry point and is resumed, executing the malicious image.

For this experiment, we use a sample that performs process hollowing on Notepad. As with the previous experiments, we upload the sample, run it, behold a "Hello World" message box and inspect the log file that was generated. As expected, the logs show that ProcessHollowing's process creates a new process, starting notepad.exe and opens a handle to that process on multiple occasions. This led to ZANDBAK automatically adjusting the scope of analysis to include the hollowed out process.

6.1.4 APC injection

Finally, we perform an experiment to see if ZANDBAK can track APC injections. The type of APC injection that is performed here is similar to the one ZANDBAK performs to start processes, but takes place in user space, leading to fewer hurdles to overcome. Memory is allocated in a target process and in it, the path to the target DLL is placed. Afterwards, the process is searched for an alertable thread. When one is found, `QueueUserAPC` is called and an APC is inserted. The APC instructs the target process to run `LoadLibrary`, passing the path to the target DLL as parameter.

We uploaded the `APCInjection.exe` binary along with `HelloWorldDLL.dll`. APCInjection.exe is engineered to perform its injection on `Calculator.exe`, so we started that prior to executing the sample. The "Hello World" message box showed, so we shut down the analysis and inspected the logs. The logs show that `APCInjection.exe` first opens a handle to Calculator's process and subsequently opens a handle to one of Calculator's threads. This lead to ZANDBAK flagging the Calculator as infected.

6.1.5 Results

In these experiments, we performed four types of process injection. In all cases, ZANDBAK was able to identify the spread of the malware.

As we already described in the previous chapter, ZANDBAK's strategy is based on the assumption that malware, at some point, needs a handle to a process / thread in order to inject itself. The experiments show that this is quite a robust approach and because of its underlying assumption, we expect it to detect all types of process injection. However, this comes with the downside that false positives are expected occasionally. We have already seen this with the `conhost` process being flagged as infected in each experiment.

Still, one can devise methods of spreading malware that would bypass ZANDBAK's infection tracking mechanism. For example, one could overwrite an executable file on disk that is run on startup, although this would still show up in the logs.

6.2 Experiment 2 - assessing stealth using Al-Khaser

To assess the stealthiness of ZANDBAK, we use Al-Khaser: an open-source application that implements a large collection of anti-analysis techniques found in real-world malware [58]. Using this application, we want to see what evasive techniques ZANDBAK in practice successfully defends against and what techniques it has difficulty with. In Subsection 6.2.1, we argue which of Al-Khaser's evasion techniques are relevant to test against ZANDBAK.

Then, in Subsection 6.2.2, we run the relevant evasive techniques in Zandbak and discuss the results.

6.2.1 Relevant categories of evasive techniques

In Al-Khaser's code, its anti-analysis techniques are separated in 15 categories. The checks of each category can be activated or deactivated by setting a Boolean value at the start of the program. In this experiment, we set them to the following values:

```
BOOL    ENABLE_TLS_CHECKS              = FALSE;
BOOL    ENABLE_DEBUG_CHECKS            = TRUE;
BOOL    ENABLE_INJECTION_CHECKS        = TRUE;
BOOL    ENABLE_GEN_SANDBOX_CHECKS      = TRUE;
BOOL    ENABLE_VBOX_CHECKS             = TRUE;
BOOL    ENABLE_VMWARE_CHECKS           = FALSE;
BOOL    ENABLE_VPC_CHECKS              = FALSE;
BOOL    ENABLE_QEMU_CHECKS             = FALSE;
BOOL    ENABLE_XEN_CHECKS              = FALSE;
BOOL    ENABLE_WINE_CHECKS             = FALSE;
BOOL    ENABLE_PARALLELS_CHECKS        = FALSE;
BOOL    ENABLE_CODE_INJECTIONS         = FALSE;
BOOL    ENABLE_TIMING_ATTACKS          = TRUE;
BOOL    ENABLE_DUMPING_CHECK           = TRUE;
BOOL    ENABLE_ANALYSIS_TOOLS_CHECK    = TRUE;
```

The **TLS checks** (Thread Local Storage) are anti-debugging techniques that are irrelevant due to Zandbak not being a debugger. We would have disabled the **debug checks** category as well if not for one technique that checks if the parent process is `explorer.exe`. This particular check is interesting because it verifies if the APC injection indeed works as expected. The **injection checks** inspect the process in various ways to see if suspicious modules are loaded. These checks could show if Zandbak indeed does not load modules into the malware's process. Next up are **generic sandbox checks** and checks for specific types of sandbox. Since we use VirtualBox, we chose to enable the generic checks and the **VirtualBox checks** that specifically try to detect VirtualBox. Zandbak's ability to track various types of process injection was already verified in experiment 1, so we disabled those checks here. Although Zandbak does not implement time acceleration / sleep skipping techniques, we enabled the **timing attacks** category because it implements two techniques that could still possibly detect the VM rather than Zandbak. The **dumping checks** modify the PE header in an attempt to prevent process dumping. We enabled these to see how Zandbak handles these modifications. Finally, we enabled the **analysis tools checks** to see if we accidentally install suspicious software on the analysis VM.

6.2.2 Results

After building Al-Khaser with the configuration in the previous section, we uploaded and started it in the analysis VM through the server's console. In the remainder of this section, we will discuss the output of Al-Khaser per category.

Debug checks : 36 / 36 evaded Unsurprisingly, all anti-debugging checks are evaded, including the check that sees if the parent process is `explorer.exe`, which shows that our implementation of APC injection indeed results in `explorer.exe` being the malware sample's parent process.

Injection checks : 8 /8 evaded ZANDBAK does not inject any modules. As expected, all checks that search for module injection were evaded.

Generic sandbox / VM checks : 31 / 51 evaded ZANDBAK failed to evade a large number of checks in this category. The following listing shows what these checks are.

```
[*] Checking Local Descriptor Table location                  [ BAD ]
[*] Checking ProcessId using WMI                              [ BAD ]
[*] Checking power capabilities                               [ BAD ]
[*] Checking CPU fan using WMI                                [ BAD ]
[*] Checking Win32_CacheMemory with WMI                       [ BAD ]
[*] Checking Win32_PhysicalMemory with WMI                    [ BAD ]
[*] Checking Win32_MemoryDevice with WMI                      [ BAD ]
[*] Checking Win32_MemoryArray with WMI                       [ BAD ]
[*] Checking Win32_VoltageProbe with WMI                      [ BAD ]
[*] Checking Win32_PortConnector with WMI                     [ BAD ]
[*] Checking Win32_SMBIOSMemory with WMI                      [ BAD ]
[*] Checking ThermalZoneInfo performance counters with WMI [ BAD ]
[*] Checking CIM_Memory with WMI                              [ BAD ]
[*] Checking CIM_Sensor with WMI                              [ BAD ]
[*] Checking CIM_NumericSensor with WMI                       [ BAD ]
[*] Checking CIM_TemperatureSensor with WMI                   [ BAD ]
[*] Checking CIM_VoltageSensor with WMI                       [ BAD ]
[*] Checking CIM_PhysicalConnector with WMI                   [ BAD ]
[*] Checking CIM_Slot with WMI                                [ BAD ]
```

Firstly, this shows that Al-Khaser was able to detect a discrepancy in the location of the Local Descriptor Table. The following comment taken from Al-Khaser's source code describes the technique:

> "This trick involves looking at pointers to critical operating system tables that are typically relocated on a virtual machine. [...] On real machines, the LDT is located lower in memory than it is on guest (i.e., virtual) machines."

Secondly, we see that we cannot evade a check for power capabilities. The source code contains the following comment about this technique:

> "Check what power states are enabled. Most VMs don't support S1-S4 power states whereas most hardware does, and thermal control is usually not found either."

Thirdly, we see that we fail to evade a number of checks that use Windows Management Instrumentation (WMI) to query the status of various hardware components.

VirtualBox checks : 51 / 51 evaded This shows that VirtualBoxHardenedLoader works as intended and successfully mitigates all VirtualBox detection techniques used in Al-Khaser.

Timing attacks : 9 / 11 evaded As the listing below shows, we failed to evade the two timing attacks that do not check for time acceleration or sleep skipping but rather for the presence of a VM.

```
[*] Checking RDTSC Locky trick                      [ BAD ]
[*] Checking RDTSC which force a VM Exit (cpuid)[ BAD ]
```

Both techniques use the RDTSC instruction to get the CPU's cycle counter. The first check is based on the assumption that a call to `GetProcessHeap()` takes significantly longer on a virtual machine than on bare metal. For an in-depth explanation of this technique, we refer to a blogpost by Forcepoint [42]. The second check creates a situation that on a VM would a trigger a VM Exit. Then it looks at the cycle counter to look for overhead caused handling the VM Exit.

DUMPING CHECKS : 2 / 2 EVADED ZANDBAK was able to dump all allocations of Al-Khaser when it closed even though its PE header has been modified. Dumping still works because `NtQueryVirtualMemory`, which ZANDBAK uses, does not rely on the contents of the PE header, and even if it did, ZANDBAK has stored a clean copy that it could use.

ANALYSIS TOOLS CHECKS : 25 / 25 EVADED No analysis tools were detected. This shows we did not accidentally have any known analysis tools running on the analysis guest.

These results show that ZANDBAK succesfully defends against most, but not all, of Al-Khaser's checks. Some of Al-Khaser's techniques manage to detect the presence of the virtual machine, mostly through discrepancies in hardware states. Most of the checks in Al-Khaser aim to detect artifacts introduced by specific sandboxes. Only the checks in the generic sandbox category are truly applicable to the direct detection of ZANDBAK, none of which managed to indeed detect ZANDBAK.

6.3 Case study - Extracting a decrypted config file from a PlugX implant

PlugX is a remote access tool and was first identified in 2012 [35, 52, 50]. This piece of malware was used to target government institutions and allows attackers to remotely take control over infected systems and perform data theft. The software is of good quality, appears to be modularized and is easily extensible.

A PlugX implant consists of three parts:

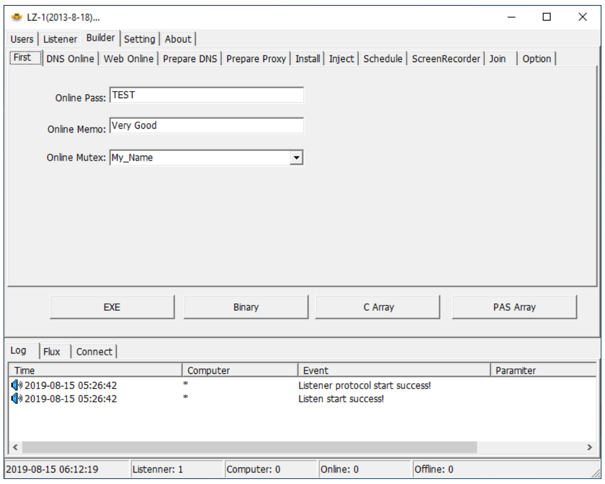
1. A legitimate, signed executable;
2. A custom .dll file;
3. A compressed and encrypted binary file containing malicious code and a configuration.

The executable is a valid signed file that the malware author took from Kaspersky Lab's *Kaspersky Internet Security*: `avp.exe`. When executed, it attempts to load `ushata.dll` from the same directory, which usually is another component of *Kaspersky Internet Security*. The malware author, however, bundled `avp.exe` with its own version of `ushata.dll` which gets loaded instead. The custom dll then reads `ushata.DLL.818` and subsequently decompresses and decrypts it using `RtlDecompressBuffer` and a custom decryption routine. Its contains the malicous code, additional modules and configuration data. All this remains in memory and is not written to disk. In this case study, we try to extract the configuration file from memory using ZANDBAK's memory dumping feature.

The C&C software for PlugX is a controller / builder. This means that it can be used to both control infected computers and to build implants with a specified configuration. A screenshot of the C&C software is shown in Figure 6.1. We use the builder to create an implant and deploy it to ZANDBAK, which is set to create a snapshot of each process 50 milliseconds after it starts and when it terminates.

Upon inspection of the log, we see that the malware indeed loads the dll as expected and then reads the binary file in its entirety. Shortly after, we see snapshots being created of each of the malware process' allocations. Each of the process' 124 allocations gets dumped to a separate file for a total of 13.1 MB.

We know that IP addresses of DNS servers and of the C&C server are encoded in the configuration file. Additionally, the allocation must be executable and writable. Therefore, we narrow down our search by looking only for allocations with `PAGE_EXECUTE_READWRITE` permissions that contain an IP address. Only one file satisfies these criteria. We open the file in a hexeditor and jump to the IP address we found. In the region around this offset, we can indeed recover a bounty of information regarding the implant's configuration. We can see IP addresses and ports, registry keys, the list of processes it tries to inject, where it stores screenshots, the password it sends to the C&C server, etc.



LZ-1(2013-8-18)...
Users | Listener | Builder | Setting | About
First | DNS Online | Web Online | Prepare DNS | Prepare Proxy | Install | Inject | Schedule | ScreenRecorder | Join | Option
EXE
Binary
C Array
PAS Array
Log | Flux | Connect

Online Pass: TEST
Online Memo: Very Good
Online Mutex: My_Name

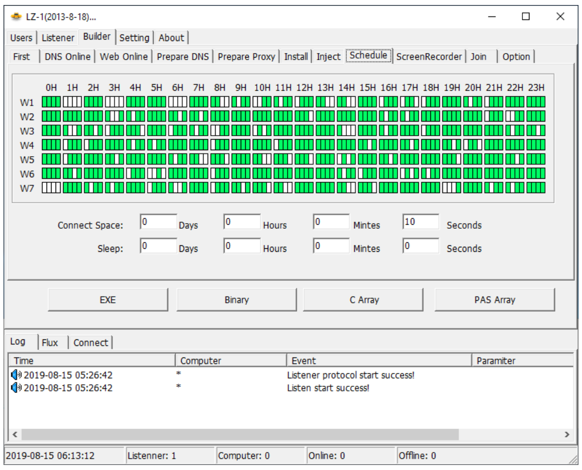
Time	Computer	Event	Parameter
2019-08-15 05:26:42	*	Listener protocol start success!	
2019-08-15 05:26:42	*	Listen start success!	

2019-08-15 06:12:19 Listenner: 1 Computer: 0 Online: 0 Offline: 0

Figure 6.1: The user interface of the PlugX C&C software.

The builder also allows for configuration of a schedule that determines when the malware becomes active. At offset 0x2c0c4, we can see the schedule encoded as an array of $672 (= 7 * 24 * 4)$ elements, each corresponding to one quarter in the week. Each element is either 0 or 1, depending on the whether the malware should be active during the corresponding quarter or not. In Figure 6.2, we show a screenshot of the schedule in the builder and what it looked like when we recovered it from the memory dump.

In conclusion, we have successfully recovered the configuration of the PlugX implant

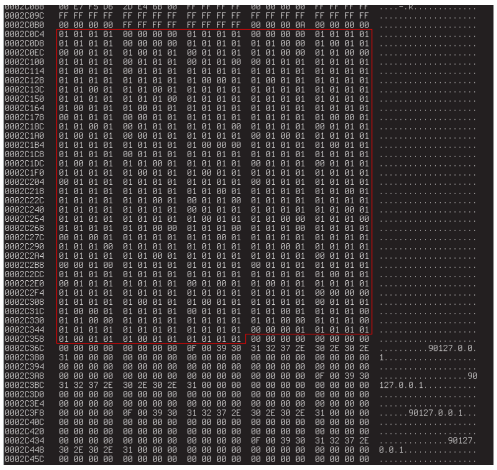


LZ-1(2013-8-18)...
Users | Listener | Builder | Setting | About
First | DNS Online | Web Online | Prepare DNS | Prepare Proxy | Install | Inject | Schedule | ScreenRecorder | Join | Option
0H 1H 2H 3H 4H 5H 6H 7H 8H 9H 10H 11H 12H 13H 14H 15H 16H 17H 18H 19H 20H 21H 22H 23H
W1
W2
W3
W4
W5
W6
W7
EXE
Binary
C Array
PAS Array
Log | Flux | Connect

Connect Space: 0 Days 0 Hours 0 Mintes 10 Seconds
Sleep: 0 Days 0 Hours 0 Mintes 0 Seconds

Time	Computer	Event	Parameter
2019-08-15 05:26:42	*	Listener protocol start success!	
2019-08-15 05:26:42	*	Listen start success!	

2019-08-15 06:13:12 Listenner: 1 Computer: 0 Online: 0 Offline: 0



```
0002C088  00 E7 F5 D6  2D E4 6B 00  FF FF FF FF  00 00 00 00  FF FF FF FF  ....-.k.............
0002C09C  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF  ....................
0002C0B0  00 00 00 00  FF FF FF FF  FF FF FF FF  00 00 00 0A  00 00 00 00  ....................
0002C0C4  01 01 01 01  00 00 00 00  01 01 01 01  00 00 00 00  01 01 01 01  ....................
0002C0D8  01 01 01 01  00 00 00 00  01 01 01 01  01 01 00 00  01 00 01 01  ....................
0002C0EC  00 00 01 01  01 00 01 01  00 01 01 01  01 01 00 00  01 01 00 00  ....................
0002C100  01 01 01 01  00 01 01 01  00 01 01 00  00 01 01 01  01 01 01 01  ....................
0002C114  01 00 01 01  00 01 01 01  01 01 01 01  01 01 01 01  01 01 01 01  ....................
0002C128  01 01 01 01  01 01 01 01  01 00 00 01  01 00 01 01  01 01 01 01  ....................
0002C13C  01 01 00 01  01 01 00 01  01 01 01 01  01 01 01 01  01 01 01 01  ....................
0002C150  01 01 01 01  01 01 01 00  01 01 01 01  01 01 01 01  01 01 01 01  ....................
0002C164  01 00 01 01  01 00 01 01  01 01 01 01  01 01 01 01  01 01 01 01  ....................
0002C178  00 01 01 01  00 00 01 01  01 01 01 01  01 01 01 01  01 00 00 01  ....................
0002C18C  01 01 00 01  00 01 01 01  01 01 01 00  01 01 01 01  00 01 01 01  ....................
0002C1A0  01 00 01 01  00 00 01 01  01 01 01 01  00 01 00 01  01 01 01 01  ....................
0002C1B4  01 01 01 01  01 01 01 01  01 00 00 00  01 01 01 01  01 00 01 01  ....................
0002C1C8  01 01 01 01  00 01 01 01  01 01 01 01  01 01 01 01  01 01 01 01  ....................
0002C1DC  01 00 01 01  01 01 00 01  01 01 01 01  00 01 01 01  00 01 01 01  ....................
0002C1F0  01 01 01 01  01 00 01 01  01 00 01 01  01 01 01 01  01 01 01 01  ....................
0002C204  00 01 01 01  01 01 01 01  01 01 01 01  00 01 01 01  01 01 01 01  ....................
0002C218  01 01 01 01  01 01 01 01  01 01 00 01  01 01 01 01  01 01 00 01  ....................
0002C22C  01 01 01 01  01 01 00 01  00 01 01 00  01 01 01 01  01 01 01 01  ....................
0002C240  01 01 01 01  01 01 01 01  00 01 01 01  01 01 01 01  01 01 00 01  ....................
0002C254  01 01 01 01  01 01 01 01  01 00 01 01  01 01 00 00  01 01 01 00  ....................
0002C268  01 01 01 01  01 01 00 00  01 01 01 00  01 01 01 00  01 01 01 01  ....................
0002C27C  00 01 00 01  01 01 01 01  01 01 00 01  01 01 01 01  01 01 01 01  ....................
0002C290  01 01 01 00  01 01 01 01  01 01 01 01  01 01 00 01  01 01 01 01  ....................
0002C2A4  01 01 01 01  01 01 00 01  01 01 01 01  01 01 01 01  00 01 01 01  ....................
0002C2B8  00 00 01 00  01 01 01 01  01 01 01 01  00 01 01 01  01 01 01 01  ....................
0002C2CC  01 01 01 01  01 01 01 01  01 01 01 01  01 01 01 01  01 00 01 01  ....................
0002C2E0  00 01 01 01  01 01 01 01  00 01 00 01  01 00 01 01  01 01 01 01  ....................
0002C2F4  01 01 01 01  01 01 01 01  01 01 01 01  01 01 01 01  00 00 00 00  ....................
0002C308  01 01 01 01  01 00 01 01  01 00 01 01  01 01 01 01  00 01 01 01  ....................
0002C31C  01 00 00 01  01 01 01 01  00 01 01 01  01 01 00 01  01 01 01 00  ....................
0002C330  01 01 00 00  01 01 01 01  01 01 01 01  01 01 00 00  01 01 01 00  ....................
0002C344  01 01 01 01  01 01 01 01  01 01 01 01  00 00 00 01  01 01 01 01  ....................
0002C358  01 00 01 01  01 00 01 01  01 01 01 01  00 00 00 00  00 00 00 00  ....................
0002C36C  00 00 00 00  00 00 00 00  0F 00 39 30  31 32 37 2E  30 2E 30 2E  ..........90127.0.0.
0002C380  31 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  1...................
0002C394  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ....................
0002C3A8  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  0F 00 39 30  ..................90
0002C3BC  31 32 37 2E  30 2E 30 2E  31 00 00 00  00 00 00 00  00 00 00 00  127.0.0.1...........
0002C3D0  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ....................
0002C3E4  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ....................
0002C3F8  00 00 00 00  0F 00 39 30  31 32 37 2E  30 2E 30 2E  31 00 00 00  ......90127.0.0.1...
0002C40C  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ....................
0002C420  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ....................
0002C434  00 00 00 00  00 00 00 00  00 00 00 00  0F 00 39 30  31 32 37 2E  ..............90127.
0002C448  30 2E 30 2E  31 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  0.0.1...............
0002C45C  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ....................
```

Figure 6.2: The malware's schedule in the builder (left) and in the memory dump (right).

using ZANDBAK's snapshotting feature. This shows that ZANDBAK indeed can be used to recover unpacked / decrypted malware components from memory.

6.3.1 Conclusion

We can relate the output of the experiments and the case study to *stealth* and *fidelity*: two of ZANDBAK's core requirements.

Experiment 2 shows that ZANDBAK is indeed capable of defending against evasive techniques. None of Al-Khaser's techniques directly detect ZANDBAK's presence. Still, using VBoxHardenedLoader alone is not sufficient to make the virtual machine invisible - some of Al-Khaser's techniques manage to detect the presence of the VM that ZANDBAK runs in. This is less of an issue than the direct detection of ZANDBAK for reasons already described in Section 3.3.1. Firstly, ZANDBAK can run without a VM. Secondly, Malware authors increasingly often refrain from using anti-virtualization techniques. Although there is some room for improvement, we would argue that ZANDBAK's defences are sufficient and its stealth requirement is met.

In experiment 1, we saw that ZANDBAK's infection tracking rules indeed are effective, but are rather coarse and sometimes yield false positives. The case study highlighted the value of ZANDBAK's snapshotting feature. Using it, we were able to recover a configuration file that would have been very difficult to obtain through static analysis. These examples show that ZANDBAK's snapshotting and infection tracking features contribute to its fidelity.

Chapter 7

Related Work

Being released in 2007, CWSandbox was one of the first malware analysis sandboxes [32]. CWSandbox monitors malware by injecting DLLs that hook all exported Windows API functions. This is very similar to how Cuckoo Sandbox [39], which currently is the most popular open source malware analysis sandbox, performs its analysis. CWSandbox and Cuckoo are both highly automated and can analyze a high throughput of malware samples. However, both sandboxes are easily evaded due to the multitude of artifacts they introduce [36, 5]. Nearly no stealth and tamper resistance is provided, which could potentially lead to incorrect results.

To make it harder for malware to detect Cuckoo Sandbox, Correia, Chevalier and Moreau developed Zer0m0n [61], a kernel mode driver for Cuckoo to perform analysis from kernel space. Although this is a step in the right direction, it completely misses the mark by still relying on a user space Cuckoo components, for example to communicate with the Cuckoo server.

Joebox is a sandbox that hooks library calls in user space as well as system service invocations in kernel space [11, 34]. It does so by placing export address table and SSDT hooks. In an attempt to hide itself, Joebox employs rootkit techniques. In particular, to prevent detection, it installs a page-fault handler and marks the memory page containing its executable code as “not present”. Whenever a process tries to access that page, the page-fault handler is called and returns a fake version of that page.

A more recent approach to stealthy sandboxing is Ether [9] which makes use of Virtual Machine Introspection. Ether is very hard to detect because it completely resides outside of the analysis environment, in a modified version of Xen. Ether leveraged the fact that page-faults can be configured to trigger VMEXITs on specific occasions to trace system calls in the analysis VM at the cost of significant overhead. By using various hardware virtualization extensions, such as Intel VT, Ether can hide side-effects that are introduced by the analyzer. Despite significant effort being put into hiding the side effects, Pek et al. devised ways to detect out-of-the-guest malware analyzers and for Ether specifically [24].

In 2014, Lengyel et al. built on VMI-based approaches with DRAKVUF [17]. DRAKVUF uses the break point injection technique (#BP) to trap kernel functions. It is faster than Ether, but still causes a significant slowdown of the analysis machine.

Zandbak's stack walking technique builds on a recent paper by Otsuki et al. [23] In this paper, they explain how one can build stack traces from the memory dump of a x64

Windows machine. We expand on their work by implementing a variant of their technique in ZANDBAK. Our variant does not work on a memory dump but rather on a live system from within the Windows kernel. With this, we offer a method of stack analysis from the kernel that previously was not possible.

Chapter 8

Future work

Currently, Zandbak is more of a prototype than a finished product. We believe this is reasonable considering the time constraints and goals of this thesis.

During analysis, Zandbak captures a bounty of information. This information gets written to a log file that can be difficult and time-consuming to comprehend. To turn Zandbak into a fully fledged product, mainly its usability should be improved. A good start would be to perform post-processing and generating a report. Post-processing could include listing modified files and registry keys and automated comparison of process' snapshots over time.

One way to implement this would be to integrate it with an existing product. For example, Zandbak could be modified such that the driver provides the same interface as Cuckoo's user mode agent. That way, Cuckoo's interface and report generator could be reused and there would be no need to implement a similar product that would be functionally the same.

From a research & development standpoint, it would be interesting to further tap into the potential of real-time stack walking. As of now, we only generate a call trace, ignoring any data on the stack that is not a return pointer. Among other things, the data we ignore contains allocations on the stack, such as arrays and buffers, whose content might be valuable for an analyst.

Furthermore, the results of the call trace can be expanded upon. In some situations, we see that a system service is invoked as the result of a program calling a Windows API function. We can directly see what parameters were passed to the system service. Unfortunately, parameters passed to functions are usually not stored on the stack but in registers that get overwritten continuously. Still, it might be possible to establish a mapping between system service parameters and Windows API functions that allows us to reconstruct the parameters that were passed to the Windows API function.

The infection tracking mechanism currently uses a very coarse ruleset for determining if a process potentially gets infected. To reduce the amount of false positives, one could narrow down situations in which infection can take place and make the rules more specific. For example, one could change the rule "If infected process X opens a handle to process Y, process Y becomes infected" to "If infected process X opens a handle to process Y with permissions Z, process Y becomes infected".

To detect certain types of malware and identify behaviors, YARA could be integrated into ZANDBAK, applying its rules to the memory dumps that ZANDBAK makes.

Some malware only runs with elevated privileges i.e. as administrator or SYSTEM. ZANDBAK currently uses `explorer.exe` as a target for APC injection. This process runs with user privileges. In an attempt to run samples with elevated privileges, we tried to inject into `svchost.exe` instead. This failed however because of it being a Windows Protected Process which does not allow for allocating new executable memory. Although this issue can certainly be worked around given ZANDBAK enjoys the privileges of ring 0, we abandoned the issue because of time constraints of the thesis.

To limit the scope of this thesis, we completely left out an important aspect of malware analysis, namely network traffic generated by the malware. In future work, one can expand ZANDBAK to include this type of analysis. This could either take the shape of a module within the sandbox itself or as a proxy that the sandbox' network traffic is routed through. If one chooses the former option, it could be implemented in such way that only traffic generated by infected processes gets recorded, reducing the amount of captured irrelevant traffic considerably, reducing complexity of analysis.

Besides that, support for common malware file formats beside *.exe*, such as *.dll*, *.xls*, *.vbs* and *.pdf* would greatly increase the range of malware it could analyze.

Although we verified ZANDBAK's ability to tracking infection spreading for a varied selection of popular code injection techniques in experiment 1 (Section 6.1), there exist many more code injection techniques that we have not tested. To get a more complete view, one could identify a wider range of code injection techniques and variants and expand the experiment with those.

Finally, the results of experiment 2 in Section 6.2.2 show that there is room for improvement regarding the virtual machine's stealth. Spoofing the output of WMI queries to reflect real hardware states would greatly reduce the number of techniques that can detect the presence of a virtual environment. The experiment itself also can be improved. Currently, most of Al-Khaser's sandbox detection checks are tailored towards the detection of specific artefacts of known sandboxes. Only the generic sandbox checks were applicable to ZANDBAK. To gain better insight in the detectability of ZANDBAK, checks that specifically target ZANDBAK are needed.

Chapter 9

Conclusions

In Chapter 1 and Section 3.2 of this thesis, we saw that there is a need for stealthy sandboxes that can analyze evasive malware. In Section 3.3, we looked into the techniques that evasive malware utilize in an attempt to prevent analysis and doing so, answered sub-question SQ1. We considered three categories of evasive techniques: anti-debugger, anti-virtualization and anti-sandbox. Of these three, defending against anti-sandbox techniques has highest priority. Today, malware authors have realized that virtual machines are no longer only used for malware analysis and that it is not uncommon for valuable targets to run in a VM. For this reason, defending against anti-virtualization techniques is important, yet not as important as defending against anti-sandbox techniques. Anti-debugger techniques are out of scope for this thesis because malware analysis sandboxes generally do not use debuggers.

As we saw in Chapter 7, a number of malware analysis sandboxes already exist, of which Ether and Drakvuf are particularly stealthy due to their usage of VM introspection. However, as we saw in Chapter 4, it is difficult to perform in-depth analysis using VM introspection compared to methods that rely on in-guest components.

To this end, built Zandbak, a modern, stealthy kernel-based sandbox for Windows 10 that sets itself apart from other sandboxes with several novel malware analysis capabilities. This required a deep dive into both documented and undocumented features of the Windows kernel. We provided the necessary background information for this in Chapters 2 and 4. Notably, Section 4.2 compares various methods of monitoring malware behavior, providing an answer to SQ4.

Kernel mode sandboxes are relatively uncommon, presumably due to Windows kernel development being a challenging and extremely time-consuming endeavor for a reasons we discussed in Section 5.3.1. In summary, there are many restrictions on how code must be programmed and what it can do at any given time. Making even a minor mistake results in a blue screen and debugging is difficult. Additionally, many aspects of the kernel are not officially documented and no high-level library functions are available.

In Section 5.1, we specified *security*, *stealth* and *fidelity* as Zandbak's requirements. The security requirement is satisfied by relying on the VMM to keep the sandbox isolated from the rest of the system and not connecting the analysis VM to computer networks. Zandbak stays under the radar by residing purely in kernel space. This leads to user space malware being unable to detect the sandbox, simply by lacking the privileges needed to access the resources that would reveal its presence. In addition, we used VBoxHardenedLoader and installed decoy software to harden the VM that Zandbak runs in against detection. By residing purely in the kernel and hardening the VM against detection, we showed how defenses against sandbox evasion can be implemented. This answers SQ3. Unlike other sandboxes, we prioritized *fidelity* over *scalability*. As scalability is less of a concern, it allows us to implement powerful yet resource-intensive features, such as snapshotting and real-time stack walking. These two, as well as infection scope tracking are ZANDBAK's novel features. Using these novel features alongside standard sandbox functionality and modified existing techniques enables ZANDBAK to perform in-depth analysis of the malware that runs in it.

In hindsight, snapshotting is not as resource-intensive as we initially expected. Using the message queue described in Section 5.2.3, we observe that its impact on performance is minimal. Stack walking indeed is resource-intensive. When used, we observe a significant slowdown. This is probably due to our suboptimal implementation of the function name lookup table.

In Section 5.2 we explain the workings of each of ZANDBAK's features and components in detail. Subsequently, in Section 5.3 we show how these features and components are combined into a WDM driver and how the server that controls ZANDBAK functions. These sections provide an answer to SQ2. Section 5.2.6 describes a method for infection tracking that answers SQ5.

We evaluated the effectiveness of ZANDBAK in Chapter 6, where we performed several experiments and a case study analyzing an implant of the PlugX malware. The results of the experiment in Section 6.2 provide an answer to SQ6 and show that ZANDBAK indeed is capable of defending against evasive techniques. Likewise, the experiment in Section 6.1 and the case study in Section 6.3 demonstrate ZANDBAK's ability to monitor the behavior of malware accurately and in-depth.

Further improvements could be made to this work, as mentioned in Chapter 8. Although ZANDBAK's technology is very interesting from a technical standpoint, a lot of work is still required to achieve good usability. There are many options to increase the scope of analysis, add more features or to improve on existing ones. It would be particularly interesting to expand on the real-time stack walking to gain even more insight in the malware's behavior with it.

Bibliography

Academic

[1] Amir Afianian, Salman Niksefat, Babak Sadeghiyan, and David Baptiste. “Malware Dynamic Analysis Evasion Techniques: A Survey”. In: *arXiv preprint arXiv:1811.01190* (2018).

[2] Davide Balzarotti, Marco Cova, Christoph Karlberger, and Giovanni Vigna. “Efficient Detection of Split Personalities in Malware.” In: NDSS. 2010.

[3] Bill Blunden. *The Rootkit arsenal: Escape and evasion in the dark corners of the system.* Jones & Bartlett Publishers, 2012.

[4] Daniel P Bovet and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management.* O’Reilly Media, Inc., 2005.

[5] Alexei Bulazel and Bülent Yener. “A survey on automated dynamic malware analysis evasion and counter-evasion: PC, mobile, and web”. In: *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium.* ACM. 2017, p. 2.

[6] Thomas M Chen and Jean-Marc Robert. “The evolution of viruses and worms”. In: *Statistical methods in computer security* 1 (2004), pp. 1–16.

[7] Xu Chen, Jon Andersen, Z Morley Mao, Michael Bailey, and Jose Nazario. “Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware”. In: *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN).* IEEE. 2008, pp. 177–186.

[8] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. “On the feasibility of online malware detection with performance counters”. In: *ACM SIGARCH Computer Architecture News* 41.3 (2013), pp. 559–570.

[9] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. “Ether: malware analysis via hardware virtualization extensions”. In: *Proceedings of the 15th ACM conference on Computer and communications security.* ACM. 2008, pp. 51–62.

[10] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. “Dynamic spyware analysis”. In: *Proceedings of the USENIX Annual Technical Conference.* June 2007, pp. 233–246.

[11] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. “A survey on automated dynamic malware-analysis techniques and tools”. In: *ACM computing surveys (CSUR)* 44.2 (2012), p. 6.

[12] Jesse M Ehrenfeld. “Wannacry, cybersecurity and health information technology: A time to act”. In: *Journal of medical systems* 41.7 (2017), p. 104.

[13] Tal Garfinkel, Mendel Rosenblum, et al. “A Virtual Machine Introspection Based Architecture for Intrusion Detection.” In: *NDSS*. Vol. 3. 2003, pp. 191–206.

[14] Ian Goldberg, David Wagner, Randi Thomas, and Eric Brewer. “A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker”. In: *Proceedings of the Sixth USENIX UNIX Security Symposium, Focussing on Applications of Cryptography*. Vol. 6. San Jose, California, July 1996.

[15] Greg Hoglund and James Butler. *Rootkits: subverting the Windows kernel*. Addison-Wesley Professional, 2006.

[16] Lech Janczewski. *Cyber warfare and cyber terrorism*. IGI Global, 2007.

[17] Tamas K Lengyel, Steve Maresca, Bryan D Payne, George D Webster, Sebastian Vogl, and Aggelos Kiayias. “Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system”. In: *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM. 2014, pp. 386–395.

[18] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. “Detecting environment-sensitive malware”. In: *International Workshop on Recent Advances in Intrusion Detection*. Springer. 2011, pp. 338–357.

[19] Desmond Lobo, Paul Watters, Xin-Wen Wu, and Li Sun. “Windows rootkits: Attacks and countermeasures”. In: *2010 Second Cybercrime and Trustworthy Computing Workshop*. IEEE. 2010, pp. 69–78.

[20] Nikola Milošević. “History of malware”. In: *arXiv preprint arXiv:1302.5392* (2013).

[21] Andreas Moser, Christopher Kruegel, and Engin Kirda. “Exploring multiple execution paths for malware analysis”. In: *2007 IEEE Symposium on Security and Privacy (SP’07)*. IEEE. 2007, pp. 231–245.

[22] John von Neumann and Arthur W Burks. *Theory of self-reproducing automata*. Champaign, IL, USA: University of Illinois Press, 1966.

[23] Yuto Otsuki, Yuhei Kawakoya, Makoto Iwamura, Jun Miyoshi, and Kazuhiko Ohkubo. “Building stack traces from memory dump of Windows x64”. In: *Digital Investigation* 24 (2018), pp. 101–110.

[24] Gábor Pék, Boldizsár Bencsáth, and Levente Buttyán. “nEther: In-guest Detection of Out-of-the-guest Malware Analyzers”. In: *Proceedings of the Fourth European Workshop on System Security*. ACM. 2011, p. 3.

[25] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. “Polyunpack: Automating the hidden-code extraction of unpack-executing malware”. In: *2006 22nd Annual Computer Security Applications Conference (ACSAC’06)*. IEEE. 2006, pp. 289–300.

[26] Mark Russinovich, David A. Solomon, and Alex Ionescu. *Windows Internals, Part 1*. 6th ed. Microsoft Press, 2012.

[27] Nuno Santos. *Rootkits and Malware Analysis - Part III. Advanced Techniques and Tools for Digital Forensics*. University Lecture. 2018.

[28] Nolen Scaife, Henry Carter, Patrick Traynor, and Kevin RB Butler. “Cryptolock (and drop it): stopping ransomware attacks on user data”. In: *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2016, pp. 303–312.

[29] Michael Sikorski and Andrew Honig. *Practical malware analysis: the hands-on guide to dissecting malicious software*. No Starch Press, 2012.

[30] Orathai Sukwong, Hyong S Kim, and James C Hoe. “Commercial Antivirus Software Effectiveness: An Empirical Study.” In: *IEEE Computer* 44.3 (2011), pp. 63–70.

[31] Fred Touchette. “The evolution of malware”. In: *Network Security* 2016.1 (2016), pp. 11–14.

[32] Carsten Willems, Thorsten Holz, and Felix Freiling. “Toward automated dynamic malware analysis using cwsandbox”. In: *IEEE Security & Privacy* 5.2 (2007), pp. 32–39.

Non-Academic

[33] Jurriaan Bremer. *x86 API Hooking Demystified.* July 2012. URL: `http://jbremer.org/x86-api-hooking-demystified` (visited on 03/27/2019).

[34] S. Buehlmann and C. Liebchen. *Joebox: a secure sandbox application for Windows to analyse the behaviourof malware.* URL: `https://joebox.org/` (visited on 08/19/2019).

[35] NJ Cybersecurity & Communications Integration Cell. *PlugX.* Apr. 2017. URL: `https://www.cyber.nj.gov/threat-profiles/trojan-variants/plugx` (visited on 08/15/2019).

[36] Alexander Chailytko and Stanislav Skuratovich. “Defeating sandbox evasion: how to increase the successful emulation rate in your virtual environment”. In: *ShmooCon 2017.* 2017.

[37] CptGibbon. *Windows Process Injection.* GitHub repository, `https://github.com/CptGibbon/Windows-Process-Injection`. 2017.

[38] McDermott Cybersecurity. *Windows x64 Shellcode.* Jan. 2011. URL: `mcdermottcybersecurity.com/articles/windows-x64-shellcode` (visited on 06/24/2019).

[39] Cuckoo DevTeam. *Cuckoo Sandbox Automated Malware Analysis.* URL: `https://cuckoosandbox.org/` (visited on 08/19/2019).

[40] Scott Dorman. *Kernel Patch Protection aka “Patchguard”.* Oct. 2006. URL: `https://scottdorman.blog/2006/10/30/kernel-patch-protection-aka-patchguard/` (visited on 08/12/2019).

[41] EP_X0FF and Fyrre. *Universal PatchGuard and Driver Signature Enforcement Disable.* GitHub repository, `https://github.com/hfiref0x/UPGDSED`. 2018.

[42] Nicholas Griffin. *Locky returned with a new Anti-VM trick.* URL: `https://www.forcepoint.com/blog/x-labs/locky-returned-new-anti-vm-trick` (visited on 08/20/2019).

[43] Cisco Talos Group. *VBoxHardening.* GitHub repository, `https://github.com/Cisco-Talos/vboxhardening`. 2017.

[44] hfiref0x. *VBoxHardenedLoader.* GitHub repository, `https://github.com/hfiref0x/VBoxHardenedLoader`. 2019.

[45] Ashkan Hosseini. *Ten Process Injection Techniques: A Technical Survey of Common and Trending Process Injection Techniques.* July 2017. URL: `https://www.endgame.com/blog/technical-blog/ten-process-injection-techniques-technical-survey-common-and-trending-process` (visited on 07/25/2019).

[46] *Intel 64 and IA-32 Architectures Software Developer’s Manual - Volume 3A.* Intel Corporation. Sept. 2016.

[47] Kris Kendall and Chad McMillan. “Practical malware analysis”. In: Black Hat Conference (USA). 2007.

[48] Lasha Khasaia. *InjectProc*. GitHub repository, `https://github.com/secrary/InjectProc`. 2019.

[49] Kaspersky Lab. *Kaspersky Lab detects 360,000 new malicious files daily - up 11.5% from 2016*. URL: `https://www.kaspersky.com/about/press-releases/2017_kaspersky-lab-detects-360000-new-malicious-files-daily` (visited on 10/04/2018).

[50] Lastline. *An Analysis of PlugX Malware*. Dec. 2013. URL: `https://www.lastline.com/labsblog/an-analysis-of-plugx-malware/` (visited on 08/15/2019).

[51] John Leitch. *IAT hooking revisited*. 2011. URL: `http://www.autosectools.com/IAT-Hooking-Revisited.pdf` (visited on 03/27/2019).

[52] Computer Incident Response Center Luxembourg. *Analysis of a PlugX variant (PlugX version 7.0)*. Mar. 2013. URL: `http://circl.lu/assets/files/tr-12/tr-12-circl-plugx-analysis-v1.pdf` (visited on 08/15/2019).

[53] Craig Marcho. *Windows 7 / Windows Server 2008 R2: Console Host*. Mar. 2019. URL: `https://techcommunity.microsoft.com/t5/Ask-The-Performance-Team/Windows-7-Windows-Server-2008-R2-Console-Host/ba-p/374219` (visited on 07/23/2019).

[54] Microsoft. *MEMORY_BASIC_INFORMATION structure*. Apr. 2019. URL: `https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntifs/ns-ntifs-_memory_basic_information` (visited on 07/18/2019).

[55] Microsoft. *User mode and kernel mode*. Apr. 2017. URL: `https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/user-mode-and-kernel-mode` (visited on 03/18/2019).

[56] Microsoft. *x64 software conventions*. Dec. 2018. URL: `https://docs.microsoft.com/en-us/cpp/build/x64-software-conventions?view=vs-2019` (visited on 07/26/2019).

[57] mxtatone and ivanlef0u. “Stealth hooking: Another way to subvert the Windows kernel”. In: *Phrack Magazine* 12.65 (Nov. 2008).

[58] Lord Noteworthy. *Al-Khaser*. GitHub repository, `https://github.com/LordNoteworthy/al-khaser`. 2019.

[59] Gavin O’Gorman and Geoff McDonald. *Ransomware: A growing menace*. Symantec Corporation, 2012.

[60] Duncan Ogilvie. *TitanHide*. GitHub repository, `https://github.com/mrexodia/TitanHide`. 2019.

[61] Conix Security. *zer0m0n*. GitHub repository, `https://github.com/angelkillah/zer0m0n`. 2016.

[62] Intel Security. *Net losses: Estimating the global cost of cybercrime*. June 2014. URL: `https://csis-prod.s3.amazonaws.com/s3fs-public/legacy_files/files/attachments/140609_rp_economic_impact_cybercrime_report.pdf` (visited on 08/19/2019).

[63] skape and Skywing. *Bypassing PatchGuard on Windows x64*. Dec. 2005. URL: `http://www.uninformed.org/?v=3&a=3&t=txt` (visited on 05/03/2019).

[64] OSDev Wiki. *Interrupt Descriptor Table*. Sept. 2018. URL: `https://wiki.osdev.org/Interrupt_Descriptor_Table` (visited on 04/03/2019).

[65] OSDev Wiki. *SYSENTER*. June 2017. URL: `https://wiki.osdev.org/SYSENTER` (visited on 04/03/2019).