Introduction to Malware Analysis Techniques

Florian Hāber

February 2015
# Contents

1 Introduction .................................................... 4

2 Preparation ...................................................... 6
   2.1 Assembly on x86 ............................................. 6
       Recognizing high-level constructs .......................... 11
   2.2 Windows NT fundamentals .................................. 17
       Key system layers ............................................ 18
       Address space layout ....................................... 20
       API & native API ............................................ 20
       Interrupts & exceptions .................................... 21
       Processes & threads ......................................... 23
   2.3 Setting up a laboratory ..................................... 24
       Virtualbox .................................................. 24
       OllyDbg .................................................... 25
       WinDbg .................................................... 25
       Other useful tools .......................................... 27
       Malware samples ........................................... 27

3 Malware Analysis ............................................... 29
   3.1 Static Analysis ............................................. 29
       Examining the PE format .................................... 29
       Strings ...................................................... 31
       Virustotal .................................................. 32
       Example: Static analysis of Brontok ....................... 32
   3.2 User-mode debugging ...................................... 33
       General orientation ......................................... 34
       Navigation through the code ............................... 36
       Example: Analyzing W32.Koobface malware ............... 37
   3.3 Decryption and unpacking .................................. 49
Chapter 1

Introduction

With the growing importance of digital technologies for such areas as private communication or banking, there are also growing incentives for unauthorized third parties to obtain that data. Malware, like trojans or backdoors, poses a serious threat to the security and integrity of computer systems, be it home computers or server infrastructures with millions of customers. For the security branch as a whole but also for responsible employees it becomes crucial to understand malware’s capabilities in order to take the appropriate counter measures. This applies for scenarios before attacks happen as well as for settings after successful attacks, where incident response is necessary to clean up and restore affected systems. In particular, also everyone who is involved in developing malware detection and removal tools relies on these insights.

Since existing work on malware analysis either focuses on single-issue topics in great detail (e.g., a single malware) or is way too encompassing but does exclude real-world examples, this paper tries to fill that gap. It aims to provide a brief but not superficial overview of the most common manual malware analysis techniques, thereby focusing on a reverse engineering approach. This is the only way for the analyst to reveal “an invaluable insight into the inner-working of [...] the malware binary” [4] because source code is not available in proper scenarios. For this reason, the paper introduces user-mode debugging with OllyDbg. Furthermore, it discusses kernel-mode debugging with Microsoft’s WinDbg.

According to Peter Szor, it “is the environment – not the malicious code – that is the difficult part to understand” [21]. Therefore, the paper starts with providing not only the most necessary basics of assembly programming on x86 processors but also the fundamentals of the Windows NT architecture (chapter 2). Throughout the paper, more assembly language details are explained. In addition, the building of a controlled research environment is touched. Notably, all featured tools are available for free.
In the analysis part (chapter 3), static and dynamic techniques are introduced with a clear focus on the latter (i.e., debugging techniques in our case). Encryption and packing methods, that are used by malware to hide functionality and to elude static analysis, are also discussed. At the end of each section, there are real-world malware samples examined, applying the respective techniques. All examples are presented to be practically retrievable by the readers themselves as a hands-on inside a controlled laboratory environment. Throughout the paper, there are code snippets provided to assist the explanation wherever possible.

The paper is mainly written for students that are new to the field of malware analysis but have basic computer science knowledge (e.g., operating system principles and experiences with high-level programming languages). It is assumed that readers are able to work with Linux. The purpose of the paper is to provide an understandable introduction for those who are not familiar with x86 assembly language and the use of Windows debuggers, both motivated by the concrete goal of malware analysis. Since these topics are usually not part of the education, assembly on x86 and disassembling are introduced from scratch. Who has missing knowledge of only one of the topics – either assembly, Windows internals or malware – will presumably also benefit from reading.

The paper does not cover any systematic classification of malware types and functionalities. It also omits network activity analysis as well as more sophisticated anti-disassembly and anti-debugging techniques. Moreover, automated analysis methods are not covered at all. Also all kinds of vulnerability exploitation are beyond the scope of the paper. I want to thank Dr. Wolf Müller very much for overseeing the work. I also want to thank Dr. Xiang Fu for answering my questions regarding the decryption of the Max++ malware.

The paper uses the following formattings:

*italic* when important terms occur the first time, for important original terms from other sources, for emphasis,

*monospace* for assembly instructions, register names, memory addresses, values, source code references (e.g., parameter names), file names, kernel structures and variables, URLs,

*bold mono* for shell commands, function names, menu bar entries.
Chapter 2

Preparation

2.1 Assembly on x86

The x86 processor has eight general purpose registers (listed in Table 2.1). These registers can be used to hold data such as variables or memory addresses. One can read from and write to them. Each register has a length of 32 bit (which is 4 bytes). Therefore each register can hold a complete address of any point in a 4 GB memory. It is also possible to access only one half of these registers. For this purpose, one has to use AX, BX, CX, DX, SI, DI, BP, SP as identifiers. In doing so, one will always access the lower 16 bits of the register. For EAX, EBX, ECX, and EDX it is even possible to halve this half again: To access the lowest eight bits, one has to use AL, BL and so on (L for low). To access the next 8 bits, use AH, BH and so on (H for high). The general purpose registers can also be used for some special tasks. Examples are also listed below.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>Holds the return value of function calls</td>
</tr>
<tr>
<td>EBX</td>
<td>Used as base for some arithmetics</td>
</tr>
<tr>
<td>ECX</td>
<td>Used for incrementing indeces in loops</td>
</tr>
<tr>
<td>EDX</td>
<td>Used for input/output operations</td>
</tr>
<tr>
<td>ESI</td>
<td>Used for string operations as source</td>
</tr>
<tr>
<td>EDI</td>
<td>Used for string operations as destination</td>
</tr>
<tr>
<td>EBP</td>
<td>Used as base pointer in stack frames</td>
</tr>
<tr>
<td>ESP</td>
<td>Points to the top of the stack</td>
</tr>
</tbody>
</table>

Table 2.1: Some special tasks of general purpose registers

Another important register is the EIP. It points to the instruction that will be executed
In contrary to general purpose registers, it cannot be overwritten directly. The x86 architecture also knows so-called segment registers that could be used to point to special memory parts. These are CS, DS, FS, ES, and SS. But besides FS, they are not used by the Windows operating system [9] and will therefore not be described in further detail. The x86 architecture also provides special purpose registers like the FPU (floating point unit) and the MMX (multi media extension) registers. Moreover, there are registers that are only used by the operating system, like the memory management registers. They are introduced in later sections as far as they are needed.

Another place where a program can store data is the stack. It is a region in memory that is used as a last-in-first-out (LIFO) data structure. To put a new value on the stack is called to push it. To take a value from it is called to pop it. Values are always pushed on or popped from the top of the stack.

To understand what the processor is doing, it is necessary to know the flag bits and how they are changing during the execution of a program. They represent the current status of the processor and provide useful information about results of prior operations. Flags are heavily used for conditional code like branching inside a program (think of if-else-statements in high-level programming languages). Flags are saved in another register called EFLAGS. Each can be set (1) or cleared (0). The flags most important for now are listed below. The highlighted row shows a flag that is accessible only from privileged level (see section 2.2).

<table>
<thead>
<tr>
<th>CF</th>
<th>Carry Flag</th>
<th>Set if an operation caused a carry out of the most-significant bit. Cleared otherwise.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZF</td>
<td>Zero Flag</td>
<td>Set if result became zero. Cleared otherwise.</td>
</tr>
<tr>
<td>SF</td>
<td>Sign Flag</td>
<td>Set equal to the most-significant bit (sign bit) of the result.</td>
</tr>
<tr>
<td>TF</td>
<td>Trap Flag</td>
<td>Set to enable single-stepping (for debugging).</td>
</tr>
<tr>
<td>OF</td>
<td>Overflow Flag</td>
<td>Set if result became too big or too small for the available range (sign bit excluded).</td>
</tr>
</tbody>
</table>

Table 2.2: Some flags from EFLAGS and their meaning [15]

At its heart an executable binary just consists of byte sequences that correspond to instructions of the processor. In an assembly program these instructions are referred to
by mnemonics [15]. These are nothing but identifiers that make an assembly program human-readable. For example the byte sequence B8 D8 refers to the instruction \texttt{MOV EAX,EBX} which moves the content of EBX to EAX.

In the following, some very common assembly instructions (or rather to their mnemonics) are listed and outlined exemplarily.

\begin{itemize}
\item \texttt{MOV EAX, EBX} \quad Copies the content of EBX to EAX.
\item \texttt{MOV ECX, 0} \quad Overwrites ECX with zeros.
\item \texttt{ADD EDX, EBX} \quad Adds two values and saves the result in the first operand.
\item \texttt{JMP 0x11223344} \quad Unconditional jump to address 0x11223344
\item \texttt{Jcc} \quad A conditional jump. Here, \textit{cc} is a placeholder for conditions like \texttt{E} (equal), \texttt{A} (above), \texttt{Z} (zero) and some more. Negations also exist, e.g., \texttt{NE} (not equal), \texttt{NA} (not above), \texttt{NZ} (not zero).
\item \texttt{JZ} \quad Jumps if the Zero Flag is set.
\item \texttt{LEA ESI, [EBP+8]} \quad Means \textit{load effective address}. The first operand has to be a register, the second a memory address. The memory address in the second operand is calculated and written to the first operand. Because the memory is not accessed, LEA is often used for fast mathematical operations.
\item \texttt{CMP EAX, EBX} \quad Compares two values and modifies flags based on the result. It is often followed by a conditional jump.
\item \texttt{CALL} \quad Calls a function by jumping to it and pushing the address of the instruction after \texttt{CALL} onto the stack (this is the so-called \textit{return address}).
\item \texttt{RET} \quad Inverts the \texttt{CALL} instruction. Jumps back to the address that is on top of the stack – which should be the return address. Usually used at the end of a subroutine.
\item \texttt{POP EDX} \quad Pops the value that is on top of the stack into a register, EDX here. The ESP is automatically adjusted (decreased) by the length of the value.
\item \texttt{PUSH EDX} \quad Pushes a value on top of the stack. The ESP is also automatically adjusted (increased) by the length of the value.
\end{itemize}
**INT** Triggers a software interrupt (explained in section 1.2). Right before, it saves the EFLAGS register and the return address on the stack.

**IRET** Returns from the the interrupt. Gets the return address from the stack similar to **RET**. But also restores the EFLAGS register from the stack.

Bear in mind, that these examples do not show all possible combinations of operands for any instruction. Sometimes operands are restricted to be registers or memory addresses. Some instructions allow multiple constellations, others do not. To be sure how to use an instruction, the Intel manual is a reliable place to look it up\(^1\). Volume 2 [16] contains the complete instruction set reference. A slightly more understandable overview of the most common instructions is given in [7]. In contrast to [16], it also contains short examples.

This whole paper uses the so-called *Intel syntax* for assembly language. There is also an *AT&T syntax* which is especially used in Unix contexts [9] (and that is only slightly different). However, since this paper will focus on Windows and all the tools presented here use the Intel syntax, we will work with this one\(^2\). Moreover, all registers and instructions are spelled with upper-case letters in this paper. But this is just for readability reasons.

```
00003c0: BB050000 0089D866 53665058 83C00190 .......fSfPX....
00003d0: 55575653 E8690000 0081C31B 1C000083 UWVS.i........
00003e0: EC1C8B6C 24308DBB 20FFFFFF E8A3FEFF ...l$0.. .......
00003f0: FF8D8320 FFFFFF29 C7C1FF02 85FF7429 ... ...t)
0000400: 31F68DB6 00000000 8B442438 892C2489 1........D$8.,$.
```

Figure 2.1: A hex dump

To understand disassembled code, one often has to calculate hexadecimal offsets to navigate through the program. Be it to calculate jump targets or for accessing data structures such as linked lists: A single byte makes the difference. Therefore it is necessary that one has a clear idea about what is happening inside the memory. In a *hex dump*\(^2\)

---

\(^1\)The Intel documentation is assumed to be the most reliable source because it is an official documentation. That this is not always the case, is shown by Ange Albertini’s x86 oddities project which can be found at [http://x86.corkami.com/](http://x86.corkami.com/) (last accessed 26.01.2015).

\(^2\)For example, although [7] focuses on Linux when explaining operating-system-dependent content, it uses the Intel syntax, too.
(the hexadecimal view of a part of the memory) a single byte is represented by two signs. Since 1 byte is 8 bits and 8 bits represent $2^8 = 256$ different values, one byte ranges from 00 to FF (which is 255 as hexadecimal). In a “classic” hex-dump view (like shown in Figure 2.1), there are 16 bytes shown in one row. The first column contains memory addresses and the last column shows the ASCII value for every single byte. If there is no printable ASCII character for that byte, a “.” will be shown.

The address shown in the beginning of the row always refers to the first byte after the colon. In our example, 00003c0 is the address of BB. Therefore the address of 05 is 0x3c1, of 00 it is 0x3c2, ..., of the last byte in the first row (which is 90) it is 0x3cf. The next line starts with 0x3d0. Notice, that leading zeros very often are left out for shortness, and 0x is used to signal that a hexadecimal value will follow. Therefore, 00003c1 is written as 0x3c1. This notation is also very common in debuggers and other tools.

To assemble a first program, this paper uses nasm which is a standard tool on Linux. first.asm (shown in Figure 2.2) will do nothing more than proceeding a simple addition in three steps. Type nasm -f elf first.asm to get the object file first.o. Now one can use gcc -o first first.o to get an executable and run it with ./first. It will say: nothing, because it did not generate any output. To do so, one could assemble the program helloworld.asm (shown in Figure 2.3) which will write Hello world! to the console.

nasm has the option -l to also generate a listing file. This is very helpful to retrace how each assembly instruction is translated into machine code. To use this option, one has to write nasm -f elf -l first.lst first.asm. When looking at first.lst with an arbitrary editor, one will discover the instructions line by line. It is observable that some instructions, like POP EAX, are representend by just one byte whereas others need more space. This is for optimization reasons in order to construct shorter programs. POP EAX is a frequently used instruction. When comparing first.lst with the hex dump from Figure 2.1, one can discover that all these instructions are part of the first rows (besides, they are written as a sequence). This is because the hex dump was created through xxd -g 4 -u first, which looks at the executable as a hex file. Notice, that in both, hex view and listing file, the value 5 in MOV EBX, 5 takes four bytes (i.e., 05000000). But it is obviously written “the other way around”. That is correct and up to the little-endian byte-order of the x86 CPU. It inverts the byte-order of immediate values – and hence – writes the least significant byte first. This can be confusing sometimes, so one should have it in mind.
Recognizing high-level constructs

The goal of disassembling a suspicious file is to understand its behaviour and in case it is malware, to find characteristics to develop appropriate counter measures. Understanding its behaviour does not mean to comprehend every single instruction every time, but to get an idea what the program is doing on a bigger scale. Since nowadays computer science students are probably more convenient with reading high-level programming languages, it is wise to figure out high-level constructs in the disassembled code. This subsection is based on [14]. However, the code examples are made by the author.

The look of machine code created by compiling high-level code (such as C source code) can vary between different environments. Of course, it depends on the architecture and its instruction set (but we always assume a x86 CPU in this paper). But even on the same architecture the look differs. It highly depends on which compiler is used and the compiler settings. Thus, the same C source code compiled in two different environments
will most likely lead to two different machine codes. Also compiler optimization is one big reason for different instruction orders or even the use of varied instructions in the resulting program. Luckily however, one can discover high-level constructs in assembly code because of their specific program logic.

**Variables**

```c
#include <stdio.h>

int global1 = 5;
int global2 = 7;

void main() {

    int local1 = 3;
    int local2 = 6;

    local1 = global1 + global2;
}
```

**Figure 2.4: globallocal.c**

```
push ebp
mov ebp, esp
sub esp, 8
mov [ebp-8], 3 ; reference to stack
mov [ebp-4], 6 ; reference to stack
mov eax, dword_40C000 ; reference to memory
add eax, dword_40C004 ; reference to memory
mov [ebp-8], eax
```

**Figure 2.5: Assembly code for globallocal.c**

The first construct covered here are variables. Variables can be *local*, when declared inside a function context or *global*, when declared outside. This difference is also present in assembly code. Local variables are put on stack and referenced by the EBP whereas global variables lie in memory and are referenced by addresses. Notice, how EBP with offsets is used to reach variables on the stack. These offsets are numbers of bytes. Local variables are put below the EBP and here the offsets are −4 and −8 because we created integers which are each 4 bytes long.
If-else-branches

```c
#include <stdio.h>

void main() {
    int greater;
    int local1 = 8;
    int local2 = 9;

    if (local1 > local2) {
        greater = local1;
    } else {
        greater = local2;
    }
}
```

Figure 2.6: ifelse.c

```assembly
401000 push ebp
401001 mov ebp, esp
401003 sub esp, C ; creates space for three integers (C is twelve)
401006 mov [ebp-8], 8
40100D mov [ebp-4], 9
401014 mov eax, [ebp-8] ; move local1 to EAX
401017 cmp eax, [ebp-4]
40101A jle 401024 ; jump to else, if ebp-8 is less or equal
40101C mov ecx, [ebp-8]
40101F mov [ebp-C], ecx
401022 jmp 40102A ; jump to end, means skip else branch!
401024 mov edx, [ebp-4] ; this is the else branch
401027 mov [ebp-C], edx
40102A ... ; end
```

Figure 2.7: Assembly code for ifelse.c

Our next minimalistic example C program (Figure 2.6) tests whether `local1` or `local2` is greater. It then writes the respective value to the local variable called `greater`. But assembly code is sequential and does not know delimiters (like e.g., curly brackets in C) to define parts of code. For this reason, `jumps` are used to navigate through the code.

In the compiled code (Figure 2.7), there are two jumps. One is a conditional `JLE` (jump if less or equal) and one is an unconditional `JMP`. The first one comes right after a
CMP EAX, [EBP-8] instruction, which compares our two local variables. Depending on the result of the comparison, the program decides how to proceed. If the first operand is less or equal, code execution will jump to address 401024. Otherwise, the execution will continue as usual with the next instruction which is at 40101C. In this case, the value of local1 is written to greater (lines 40101C and 40101F). Now the unconditional jump is taken (at 401024). The next instruction would have been the beginning of the else-branch (at 401022). Thus, this jump is necessary to skip what is the else-part in the C source code.

Loops

```c
#include <stdio.h>

void main() {
    int i;
    for (i=10; i>0; i--) {
        // do something
    }
}
```

Figure 2.8: for.c

How to recognize loops is shown by means of a for loop that runs ten times until i is zero (Figure 2.8). i is decremented by 1 each round. The assembly codes might look a bit confusing (Figure 2.9). Checking for jumps reveals that there are three of them. The first one is always executed because it is unconditional (and the code is executed from 401000 to bottom, here). It just skips some lines (in a minute, one will understand why). The execution is now at 401016 where a CMP and a JLE are executed. This is the same mechanism as in the assembly of ifelse.c (Figure 2.7). In contrary to this case, the counter is compared to zero, which is our condition to stop the loop. If the counter is still above zero, then one will enter the inside of the loop. In both Figures, it is left out what happens inside the loop because it is not necessary for understanding the rough structure. After that, a JMP has to be taken which will bring the execution back to 40100D. This is the first instruction of the part that was skipped at the very beginning. Now we see why: The next four lines simply decrement the counter by 1. This happens after the execution of each round. Also notice, that just because some instructions are located earlier in memory, this does not imply that they are executed earlier. It is important to
recognize the jumps to follow the program’s control flow.

Notice, that if the loop would use a counter starting with 0 and increment it until 10 (instead of decrementing from ten to zero), the look of the assembly code would be exactly the same – with four slightly differences. The changed lines are listed in Figure 2.10. However, the structure of instructions and the jumps between them stay unchanged.

Calling conventions

Calling conventions serve the purpose that functions calls across different programs become possible. They describe the policy how and where parameters and return values are passed to enable interoperability of programs compiled with different compilers or written by different authors. As with all standards, there is a variety. Here, two major
C calling conventions are described in short. The program that calls a function is named the caller and the subroutine is the callee.

The CDECL is the convention C compilers use by default. The caller pushes the parameters in a right-to-left manner onto the stack. The callee gets executed and when it is finished, it puts the return value into EAX. The control flow switches back to the caller, who now has to remove the parameters from the stack ("it cleans up the stack").

The STDCALL is the convention used for example for Windows API calls (see section 1.2). It is similar to CDECL, except that here the callee has to clean up the stack before returning.\(^3\)

Figure 2.12 shows the assembly of sum.c in Figure 2.11. It also (in the comments) highlights the two lines that will differ when compiling with STDCALL calling convention. Regard, that memory addresses are left out here because they are not necessary for the understanding. At the one situation where a call occurs (which results in a jump internally), the disassembler inserts a label for us. Notice, that removing data from stack is done via adding the appropriate number of bytes to the current value of ESP. This involves that the data is still there but will be overwritten as soon as new values are pushed onto the stack. Applying STDCALL, the callee has to clean up the stack. This is done by passing the number of bytes as an argument to the RET instruction (8 in our example). This has a convenient side-effect for the reverse engineer. Just by looking at the RET and dividing its parameter by four, one can get an idea about how much parameters this subroutine requires.

\(^3\)The right-to-left-order for STDCALL is stated in [14] and works properly with Microsoft’s C compiler with /Gz flag enabled (Version 15.00.21022.08). But [9] states a left-to-right-order. [22] writes that Microsoft’s documentation erroneously claims a left-to-right-order. Maybe this has been true, however it is out of date, because http://msdn.microsoft.com/en-us/library/zxk0tw93.aspx (last accessed 26.01.2015) specifies right-to-left.
```c
#include <stdio.h>

int x = 5;
int y = 100;

void main() {
    sum(x,y);
    return 0;
}

int sum(int a, int b) {
    return(a+b)
}
```

Figure 2.11: sum.c

```assembly
push ebp
mov ebp, esp
push ecx
mov eax, dword_40C004
push eax
mov ecx, dword_40C000
push ecx
call sub_401030 ; calls the subroutine sum which starts at 401030
add esp, 8 ; this line misses in STDCALL mode!
xor eax, eax ; epilog
pop ebp
ret

sub_401030: ; this is just a label inserted by the disassembler
push ebp ; this instruction is at address 401030
mov ebp, esp
mov eax, [ebp+8]
add eax, [ebp+C]
pop ebp
ret ; ret 8 in STDCALL mode!
```

Figure 2.12: Assembly code for sum.c compiled with CDECL

## 2.2 Windows NT fundamentals

Windows NT is Microsoft’s first true 32-bit system and was introduced with Version 3.1 as Windows NT 3.1 in 1993. It was designed to provide a high portability on different hardware architectures. Until today, there were many updates and meanwhile, the 64-bit
architecture is fully supported. Since Windows 2000 (NT Version 4.0) the product did not contain the NT Version in its name anymore. This paper focuses on Windows XP in the 32-bit version which internal version number is NT 5.1 and was released in 2001. All information from now on refer to that version (running on a x86 uniprocessor in protected mode) and is based on [20], unless explicitly remarked.

Windows runs in two different modes that are clearly separated from each other: the user mode and the kernel mode. The purpose of this is to separate user applications and operating system structures. This separation is allowed by the underlying hardware architecture. Even though the x86 processor offers four different privilege levels [15] (also called rings), Windows only uses two of them. Level 0 is used for kernel mode, level 3 for user mode (level 1 and 2 are unused).

Notice, that the control flow of an application usually does not rely either on user-mode code only or on kernel-mode code only, but it often switches between both. The latter provides e.g. access to hardware, input/output devices, or to the file system – functionalities nearly every application relies on. In case a user application wants to execute such an operation, it will make a call to the operating system for this special task and ask it to take over. This is named a system service call [20] or usually just system call (e.g., in [9]). After this work is done, the system switches back to user mode and the application is continuing.

Key system layers

The design of Windows is based on a layered structure. Figure 2.13 (taken from [20], but modified) shows the Windows architecture in a very simplified manner. Above the horizontal line the user mode is modeled, underneath the kernel mode.

User-mode processes can be divided into four basic types.

- System support processes are fixed processes like the session manager (smss.exe), logon process (winlogon.exe) or local security authentication server (lsass.exe).

- Service processes are processes that run independently of user logons and often have no GUI. They are like daemons in Unix.

- User applications are all kind of executables that one actually has in mind when talking about programs. Examples are the notepad, the explorer or a browser.

- Environment subsystems are the highly separated Windows subsystem and POSIX subsystem, that can run POSIX conformed executables. Applications can only
run in either of them. This paper always talks about the Windows subsystem. Important components of it are the associated process (csrss.exe) and the driver win32k.sys. Until Windows 2000 there also was an OS/2 subsystem.

The subsystem DLLs together with the ntdll.dll form the place where transitions from user mode to kernel mode are organized. The subsystem DLLs consist of gdi32.dll, user32.dll and kernel32.dll for instance. These and ntdll.dll contain functions that call into the kernel mode and export them for applications in user mode. They are explained in greater detail in the subsection API & native API.

The executive includes components that manage the registry, processes and threads or the input/output. It also provides the management of objects which is the key shape how system resources like processes or synchronization objects are represented. They are reached out via object handles. The executive forms the upper half of ntoskrnl.exe whereas the kernel is the lower half of it.

The kernel (in a narrow sense) isolates the executive and device drivers from the hardware. It also provides primitives that other kernel-mode components use to develop their mechanisms. The kernel limits himself to providing primitives whereas the executive
implements policies. Moreover, it operates with so-called *kernel objects* which is like an atomic shape of the objects introduced above that resign e.g. some security overhead.

Usually, processes do not interact with hardware directly but they go through the *hardware abstraction layer (HAL)*. It is responsible for Window’s portability property and implemented in hal.dll.

**Address space layout**

In a 32-bit system where the smallest addressable unit is a byte, there are $2^{32}$ Bytes = 4 GB of memory to address. In Windows, these 4 GB are divided into two parts: By default, each user process gets 2 GB of memory and the remaining 2 GB are reserved for the operating system. When talking about memory, this always means *virtual memory*. This is the mechanism how an operating system maps physical memory to virtual addresses. When dealing with addresses inside the operating system (e.g., while debugging), these are always virtual addresses. It allows providing more memory to processes than physically is there. For this purpose, some parts of memory are *paged* out to hard disk when they are not needed, and reloaded when they are needed again. How virtual memory management works in detail is beyond the scope of this paper and can be looked up in operating systems books or chapter seven in [20].

Each process has its own address space of size 2 GB, but the 2 GB part of the kernel address space is shared. This means, it persists, regardless of which process is executed. Since there are no read/write protections inside the kernel mode, code in kernel mode can access the whole kernel address space. Needless to say, the kernel space is neither writable nor readable from user mode applications. The 2 GB kernel space always starts at address 0x80000000 and ends at 0xFFFFFFFF. In case while reverse engineering one discovers such an address, where the most significant bit is set, one can be sure that it refers to kernel space [9].

**API & native API**

The Windows *Application Programming Interface (API)*\(^4\) is the way Microsoft provides functions that access system resources to software developers. They are well documented and have the advantage to be callable from user mode (but also from kernel mode). Their main use is to provide entry points for transitions to kernel mode. They will call functions (that the developer does not have to know) that in turn trigger system calls.

\(^4\)It was formerly called *Win32 API*. This name can often be found in literature. All information about the API are online at [http://msdn.microsoft.com/library/windows/desktop/hh920508(v=vs.85).aspx](http://msdn.microsoft.com/library/windows/desktop/hh920508(v=vs.85).aspx) (last accessed 26.01.2015)
The key components of the API are the following [9]:

- **GDI API**: It includes low-level graphic primitives like drawing simple elements on screen. They are exported via gdi32.dll.

- **USER API**: These are high-level graphic related functions that allow showing windows, menues and dialog boxes. They are exported via user32.dll and build on gdi API.

- **BASE API**: This API provide stubs to all non-GUI-related functions like them listed above as functionalities of the executive. It is exported through kernel32.dll.

- **Native API**: It is “the actual interface to the Windows NT system” [9]. Its functions are undocumented and exported via ntdll.dll (for user-mode callers) and ntoskrnl.exe (for kernel-mode callers).

As modeled in Figure 2.13, a function call to an API function (which are exported by one of the subsystem DLLs) will trigger a chain of several more calls to other functions. For example, a very common call chain could start with a user-mode call to **WriteFile** (exported by kernel32.dll), which then triggers **NtWriteFile** (exported by ntdll.dll). This will invoke a so-called **system service call** (see next subsection) which switches the system into kernel mode. In this case, the system service that will be called is another function with the same name, **NtWriteFile**, but now taken from ntoskrnl.exe. These system service calls, that are invoked after switching to kernel mode, are organized in a kernel structure called the **system service descriptor table (SSDT)**.

Interrupts & exceptions

Interrupts and exceptions are fundamental concepts of the Windows operating system. Interrupts are **asynchronous** events whereas exceptions are **synchronous**. **Interrupts** are caused by events that occur independently of the running program. Whether a running task may be interrupted, can be disabled by the operating system. It also classifies who can be interrupted by whom. In contrast, **exceptions** are caused by instructions of the

---

5[9] also specifies **kernel API** as a second name, which might be a bit misleading from my point of view.

6It is officially undocumented, but there is an almost complete documentation by Gary Nebbett in his book *Windows NT/2000 Native API Reference*.

7There is also a second version of this table, the **shadow SSDT**. In contrary to the first SSDT, it also contains system service calls for graphic-related APIs. Also notice, that calls to the GDI or USER API will not go through ntdll.dll. But because malware usually does not come with graphic elements, we will not discuss them here further.
running program. Therefore, running the same program under the same conditions will usually reproduce the exception. Both, interrupts and exceptions, can be triggered by either software or hardware. When an interrupt or an exception occurs, the operating system looks up the appropriate handler in the so-called IDT. IDT stands for interrupt descriptor table.\textsuperscript{8}

<table>
<thead>
<tr>
<th></th>
<th>Hardware</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupts</td>
<td>I/O device</td>
<td>Thread switching</td>
</tr>
<tr>
<td>Exceptions</td>
<td>Illegal instruction</td>
<td>Divide-by-zero error</td>
</tr>
</tbody>
</table>

Table 2.4: Examples that trigger interrupts or exceptions

### Interrupt handling

Windows assigns priorities of interrupts (hardware and software) in the interrupt request levels (IRQL). While executing code, the processor is assigned to a specific IRQL, which can be lowered or raised by the operating system.\textsuperscript{9} The execution can only get interrupted by a source which has an higher IRQL than the current. The highest IRQLs are processor related, then device-driven interrupts follow. Software interrupts are located below and user-mode code always runs with the lowest IRQL (level 0, also called passive level).

### Exception handling

Exceptions are treated by Windows with a mechanism called structured exception handling (SEH). It defines an order how the exception is handled. Each part of the SEH can decide whether it does or does not handle the exception. If it is still unhandled, the next handler in line is asked. First, exception handlers inside the process that threw the exception are looked up. Subsequently, the debugger and the kernel follow. If no one can handle the exception, the kernel will terminate the corresponding process.

### System service calling

On older Intel processors a system service call was handled similar to an exception. It was triggered by the instruction INT 0x2E. Nowadays, it is treated differently and in the strict sense, it is neither an interrupt nor an exception. Since Pentium II, Intel provides a special instruction which is designed for this purpose. The goal was to have a faster operation

\textsuperscript{8}This is the term the Intel documentation uses [17]. [20] uses interrupt dispatch table. Consider, that despite the name it handles both, interrupts and exceptions. The IDT is a per-processor data structure.

\textsuperscript{9}The Windows kernel uses KeRaiseIrql and KeLowerIrql to do so.
because of less overhead for this heavily used mechanism. Its mnemonic is SYSENTER\textsuperscript{10}. When calling SYSENTER (e.g., from \texttt{NtWriteFile} in \texttt{ntdll.dll}), the processor will switch into privileged mode and transfer the execution to the routine \texttt{KiFastCallEntry} whose address is saved in the MSR special purpose registers. How system service calls in general and the SYSENTER instruction in particular work, and how Windows NT and the x86 CPU interact, is well explained in [12] and [13].

**Processes & threads**

A program is a set of machine code instructions lying on disk and waiting for execution. However, a process is a container of system resources provided by the operating system. The operating system allocates space in memory, that is loaded with the executable’s image, and ensures running time. In Windows, every process can have multiple threads and must at least have one of them. Otherwise, a process cannot execute anything.

For security reasons, each process has its own separate address space. Therefore it is impossible for one process to manipulate another one’s memory, be it consciously or unconsciously. However, a thread does not have its own address space. One or multiple threads share the memory of the process they are part of.

Creating a process is done by a call of the \texttt{CreateProcess} API function that is available through \texttt{kernel32.dll}. At first, the defining data structure \texttt{EPROCESS} is set up. It holds general information that mainly describe the address space layout. It also contains a pointer to the \texttt{KPROCESS} block (holds a list of kernel threads) and to the process environment block (PEB). Then the initial thread is created via \texttt{CreateThread}. An \texttt{ETHREAD} data structure is established (holds e.g., the thread’s start address). It also points to a \texttt{KTHREAD} structure (saves scheduling information) and a pointer to the kernel stack. Also a thread environment block (TEB)\textsuperscript{11} is created. After a notification to the Windows subsystem, that a new process is created, the process begins to execute.

Notice, that a process allocates kernel- as well as user-address space. In contrast to the PEB and the TEB, all other mentioned structures lie in kernel-address space. Besides the program image, also \texttt{ntdll.dll} is always loaded into the user-address space of the process.

\textsuperscript{10}AMD developed a similar mechanism independently. There the instruction is named SYSCALL.

\textsuperscript{11}Some literature use the names PIB/TIB for PEB/TEB, e.g. [9], where the I stands for information.
2.3 Setting up a laboratory

This section is about how to set up a controlled environment to perform manual malware analysis with the help of a virtual machine. Be aware that there is malware that is able to detect virtual machines [10] and that it is possible to escape from it into the host system [19]! This is the reason why this is called a controlled environment and it is abstained from the term secure. In this paper, there are plenty of tools and settings recommended, besides there might be (and are) others for same use out there. The following is required:

- A machine with a x86 CPU\textsuperscript{12}
- Windows running on it (any version)
- Windows XP installation disc\textsuperscript{13}
- Virtualbox
- OllyDbg
- Windows SDK with WinDbg included and symbol files
- ProcessExplorer
- Strings
- PEiD
- OSR Driver Loader

**Virtualbox**

Virtualbox is a open source virtualization tool that can be downloaded for free\textsuperscript{14}. There are versions for Windows, Linux, OS X and Solaris. Create a new virtual machine by clicking **New** and following the instructions. Choose Microsoft Windows as type and Windows XP (32-bit) as version. Create a new virtual hard drive. After that, the virtual machine is listed on the left. Go to the settings, choose **Network**. Remove the tick at “Enable Network Adapter”. Got to **System→Accelaration** and activate Vt-x. Save the

\textsuperscript{12}It will be advantageous if it supports the virtualization feature called VT-x (for Intel) or AMD-V. For instance, this will allow to set hardware breakpoints later. For Intel CPUs, one can check whether VT-x is supported at http://ark.intel.com/Products/VirtualizationTechnology (last accessed 29.12.14).

\textsuperscript{13}If not available, one can download a virtual image Microsoft provides for free at https://www.modern.ie (last accessed 11.02.2015). These are trial version that can be used 30–90 days. Throughout this paper, Windows XP with service pack 2 is used.

\textsuperscript{14}https://www.virtualbox.org (last accessed 11.02.2015)
settings and run the virtual machine. Now select your CD drive and put your Windows installation disc in. Follow the Windows installation instructions. When Windows is installed and booted inside virtualbox, click **Devices → Insert Guest Additions CD image**. These will amongst others allow better screen resolution that uses the whole screen. We will refer to the Windows XP inside virtualbox as the *guest*. The system where virtualbox is installed, is called the *host*.

Virtual machines do not only offer the opportunity to easily run an operating system inside another one. They also allow to save machines states, reload them and thereby go back to a setting in the past. When doing malware analysis, this will help to save a clean lab environment and jump back to it, in case a malicious program infected the system. Saving the current machine state is called “taking a *snapshot*”. There is also another helpful feature. A snapshot, respectively a whole virtual machine, can be duplicated by *cloning* it. This enables the user to run several guests for different purposes without going through the whole installation process more than once.

**OllyDbg**

OllyDbg is a user-mode debugger which is freely available and open source\(^\text{15}\). It allows dynamic analysis of malware and because it is a debugger, the malware can executed in a more controlled manner (e.g., single-stepped) than simply letting it run. This paper uses version 1.10 as recommended in [14] although there is a version 2.0. There are also several plug-ins developed by the community. They can be downloaded from the OpenRCE website\(^\text{16}\). OllyDbg does not require any installation and no further configurations. Plug-ins have to be copied into the OllyDbg folder and are loaded automatically.

**WinDbg**

WinDbg (spoken as “windbag” sometimes) is Microsoft’s Debugger and interesting here because it not only allows user-mode debugging but also kernel-mode debugging. The MSDN\(^\text{17}\) describes different ways to get it. This paper recommends to download it as part of the Windows SDK (Software Development Kit) which also contains other useful tools. We will use it in GUI (graphic user interface) mode.

It is also recommended to download the *symbol files* for Windows XP SP2. They are

\(^{15}\)http://ollydbg.de/ (last accessed 17.01.2015)

\(^{16}\)http://www.openrce.org/downloads/browse/OllyDbg_Plugins (last accessed 17.01.2015)

also offered at the MSDN website\textsuperscript{18}. After installing them, one has to go to \texttt{File→Symbol File Path} in WinDbg and specify the right path.

Kernel-mode debugging of a machine can be done by a second machine which is connected via some wire. Here however, WinDbg will be running in the host Windows to debug the guest. The connection between them is a virtualized serial port. Before running the guest, one has to go to \texttt{Settings}, choose \texttt{Serial Ports} and:

- Activate \texttt{Enable Serial Port},
- set \texttt{Port Number} to \texttt{COM1},
- set \texttt{Port Mode} to \texttt{Host Pipe},
- activate \texttt{Create Pipe},
- set \texttt{Port/\text{\textbackslash}File Path} to \\
\textbackslash.\pipe\com\textsubscript{1} and
- leave \texttt{IRQ} and \texttt{I/P Port} as is.

To enable kernel-debugging of the guest, one has to boot it in \textit{debug mode}. This needs one configuration to be done. Run the guest in virtualbox and then open \texttt{C:\boot.ini}. Below the last entry start a new line and type in one(!) line:

```plaintext
multi(0)disk(0)partition(1)\WINDOWS="Microsoft Windows XP Debugged"
/noexecute=optin /fastdetect /debug /debugport=com1 /baudrate=115200
```

This enables a new boot option at system startup which will be labeled as “Microsoft Windows XP Debugged”. Be aware that making a mistake here may crash the system \textcite{1}!

To begin with kernel debugging one has to select \texttt{File→Kernel Debug} in WinDbg. The following options has to be set:

- Set \texttt{Baud Rate} to 115200,
- \texttt{Port} to \\
\textbackslash.\pipe\com\textsubscript{1} and
- activate \texttt{Pipe}.

When clicking **OK**, not much will happen, until rebooting the guest OS in debug mode (the boot process may take longer with WinDbg attached). If everything is configured well, WinDbg will now start to print some output. When Windows XP is loaded completely, the execution can be broken by clicking **Debug → Break** or by hitting **Ctrl+Break**. The guest should now be freezed and WinDbg should display a breakpoint information. To continue running again, one has to type **g** for **go** into the console line (which starts with **kd>**).

**Other useful tools**

Mark Russinovich (co-author of [20]) and Bryce Cogswell developed a couple of tools that are known as **Sysinternals**. They provide a lot of real-time information of Windows internals, like processes, threads, handles, registry changes and network traffic. After taken over by Microsoft, they are now offered on their websites[^19], still for free. This paper recommends at least the use of **ProcessExplorer** which is a must-have alternative to the Windows task manager for security related work. It shows order of magnitude more information about running processes. Even memory-related details like stack frames can be obtained. While doing dynamic analysis of user-mode malware, it is helpful to track the current level of infection.

A sysinternals tool used for static analysis in this paper is **Strings**. It searches for byte sequences that have the format of a string inside an executable. There exists a similar tool on Linux with the same name for the same purpose.

In order to detect packing algorithms, a very common tool that is also used in this paper is **PEiD**. It is able to detect more than 600 such algorithms and can be downloaded for free from woodman.com’s RCE collaborative RCE tool library[^20].

Some kernel-mode malware comes in shape of a driver and without the user-mode part that would load it. To do so nevertheless, one can take advantage of **OSR Driver Loader**[^21]. Downloading requires a registration.

**Malware samples**

Often malware analysis has to deal with new threats and unknown binaries. But of course there are scenarious where one has to handle malware that was already named and classified, e.g., for research reasons as in this paper. In order to analyze such a

malware, there are plenty of places where one can get known malware samples. Lenny Zeltser, who is the head of the private SANS Institute\textsuperscript{22}, recommends several free sources on his website\textsuperscript{23}.

The malware samples used in this paper were taken from Open Malware\textsuperscript{24} (formerly known as Offensice Security) and Malware.lu\textsuperscript{25}. To download a sample from the former, one has to authenticate with a Google account. For the later, a registration is needed which has to be approved. The samples are compressed and the password usually is infected. Appendix A lists the malware analyzed throughout this paper as well as their sources and hash values.

\textsuperscript{22}https://www.sans.org/ (last accessed 26.01.2015)
\textsuperscript{23}http://zeltser.com/combating-malicious-software/malware-sample-sources.html (last accessed 26.01.2015)
\textsuperscript{24}http://openmalware.org/ (last accessed 26.01.2015)
\textsuperscript{25}http://malware.lu (last accessed 01.02.2015)
Chapter 3

Malware Analysis

3.1 Static Analysis

The first step one can take in order to analyze a specimen, is to examine it by looking at it in a hex viewer. Under Linux, this could be generated with `xxd -u test.exe` to produce the output shown in figure 2.1.

```
0000000: 4D5A 9000 0300 0000 0400 0000 FFFF 0000 MZ..............
0000010: B800 0000 0000 0000 4000 0000 0000 0000 ........@.......
0000020: 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000030: 0000 0000 0000 0000 0000 0000 8000 0000 ................
0000040: 0E1F BA0E 00B4 09CD 21B8 014C CD21 5468 ........!..L.!Th
0000050: 6973 2070 726F 6772 616D 2063 616E 6E6F is program canno
0000060: 726F 6772 616D 2063 616E 6E6F 2062 65 2072 756E 2069 6E20 646F 6368 2E0D 0D0A 2400 0000 0000 0000 is program canno t be run in DOS
0000070: 6D6F 6465 2E0D 0D0A 5045 0000 4C01 0700 A5EA 684E 0000 0000 mode....$........
0000080: 5045 0000 4C01 0700 4C01 0700 A5EA 684E 0000 0000 PE...L......hN....
```

Figure 3.1: First lines of a hex view of an arbitrary executable

Examining the PE format

Executables in Windows NT come in a file format which is called PE (*portable executable*). From just viewing a suspicious file while understanding its format, one can get useful information and further hints, whether it could contain malicious code. The PE format is used for files of type `.dll` and `.exe`.

The PE format is relocatable [9]. This means it does not matter to what memory address it is loaded to work properly. Because of the many cross-references inside an executable (references to global variables or calls to imported library functions) a mechanism
is needed to substitute absolute addresses in the code with them really used when executing in memory. This work is done by the loader. Every module has a base address. If this address is already taken while loading, the executable gets relocated. The addresses in the PE header are always relative offsets, called *relative virtual addresses (RVA)*.

The overall structure of a PE file consists of the *header* followed by *sections* as shown in Figure 3.2\(^1\). The `MZ` value indicates the beginning of the old MS-DOS header with which every PE file starts. It is followed by the PE header (starting with `PE`) that signals that this is not a MS-DOS executable. One can discover both in the example hex dump above because they are readable ASCII signs. Thereafter, the *optional header*, *data directories* and a *sections table* follow, which complete the header part.

The bottom part of an PE file contains the sections [14]. The number, names, and order of sections differ from file to file. They depend on the used language, compiler settings and the programmer. At least there is a `.text` section (sometimes called `.code`) that lists the actual machine code instruction by instruction. Also there is a `.data` section that contains all global data the program needs (like global variables). Often a `.rdata` section contains information about the imported and exported functions and a `.rsr` section stores resources like images and menus.

![Figure 3.2: Structure of the PE file format](http://pics.corkami.com)

The Windows SDK comes with a tool called *dumpbin* which dumps an executable like a hex viewer would do but with regard to the PE file format. It can give a first

\(^1\)There is a great visualization of the PE file format in poster style under [http://pics.corkami.com](http://pics.corkami.com) (last accessed 07.01.2015).
overview about the suspicious file. If used with the /headers option, it shows for which architectures, operating system versions and when it is compiled. Keep in mind, that the malware author could also have changed the date afterwards to mislead analysis [14]. It also lists the sections of the PE file and their sizes. Usually, the virtual size and size of raw data are very similar. If the raw data is much smaller than the virtual size, this is an indication that it is program might be packed. See section 3.3 for how packed files can be handled.

Another useful option is /imports which lists the library functions that are imported by the executable. The use of some functions may be a good hint that the examined file could be malware. Some combinations of functions can lead a trained eye to a good guess about what the program might be doing. [14] provides a long and auxiliary list of functions that could be suspicious. Some examples are shown below. Although such functions can be a hint, of course they can also be used in benign programs. For instance, calc.exe also imports GetProcAddress from kernel32.dll.

CreateRemoteThread Tries to start a thread in a remote process.
GetProcAddress Returns the address of a function in a loaded module (e.g. dll) even if it is not listed in the list of imported functions.
NetScheduleJobAdd Starts another program deferred at a chosen date and time.
RegOpenKey Will read and edit a registry key.
VirtualAllocEx Tries to allocate memory in a remote process.

Table 3.1: Some API functions that might be suspicious

Strings

Another possibility to start examining an unknown binary is the use of strings under Linux or Windows. This tool searches for null-terminated byte sequences that have at least a specified number of bytes (≥ 4 by default). If the output contains one or more URLs that do not look confidential or an IP address, it could be malware that tries to connect to something like a command-and-control server. One could also observe the names of system or other critical files, or the names of suspicious registry keys. A program that contains such a registry key probably wants to set itself to run at start-up without being shown in the start menu:
Virustotal

Virustotal\(^2\) is an online anti-virus software that is actually a collection of many common anti-virus products. Users can upload files or submit hashes of files they own. The web service checks the uploaded file and provides an analysis or it shows an older analysis if the file or the submitted hash is already known from a former analysis. If the hash is not known, the file has to be uploaded. The hashes of most of the Windows default system files will probably be known.

Since Virustotal obviously runs many different anti-virus products, itself it is not restricted to static analysis. But for a user who submits files or hashes, it is a static analysis technique because she is not running the suspicious file inside her own system.

Example: Static analysis of Brontok

When applying these static analysis techniques to the Brontok malware, one can gain several important insights. The strings analysis reveals a long list of unreadable strings. But three strings are readable and helpful. Obviously two API functions from kernel32.d11 are used: LoadLibraryA and GetProcAddress. The /imports analysis verifies that these two are imported – but no other API functions. This suggests that LoadLibraryA\(^3\) is used to load other modules into the address space at runtime. Then GetProcAddress is used to get the actual addresses of needed API functions that are exported by the loaded module.

The /headers analysis reveals a suspicious ratio between virtual size and raw data in both sections. This is a hint that Brontok is packed. This could also be a reason why just a few strings are readable. In order to go on with the analysis of Brontok, one had to find a way to unpack it and do further static analysis – or one would start dynamic analysis, e.g. with running the unpacked version of Brontok in an user-mode debugger.

The analysis with Virustotal\(^4\) reveals several more information. 51 out of 56 anti-virus products recognize this file as malware. It confirms the conjecture that the file is packed

\(^2\)https://virustotal.com (last accessed 07.01.2015)
\(^3\)https://msdn.microsoft.com/en-us/library/windows/desktop/ms684175(v=vs.85).aspx (last accessed 05.02.2015)
\(^4\)https://www.virustotal.com/de/file/cac5bc25e94989ee18f48903f4675151b802f013d1365f174a84f0468918f168/analysis/ (last accessed 05.02.2015)
(i.e. MEW 11 SE v1.2 Northfox[HCC] packing algorithm). A behavioural analysis yields that three other .dlls are loaded at runtime. Also opened files, hooks (see section 3.4) and UDP communications are detected and listed.

### 3.2 User-mode debugging

A user-mode debugger allows the analyst to run a malicious executable in a controlled manner and, meanwhile, to examine the changes of the program’s state in detail. The analyst can control and adjust the program flow, stop and continue it wherever and whenever wanted. When stopped, one can examine the current contents of the CPU registers, the stack, the memory and the code, of course. Needless to say, although
the execution is controlled and maybe just done step-by-step, the program is indeed *executed*. Therefore user-mode debugging of malware should only be done in a controlled environment which is inside a virtual machine in our case!

When a malware is executing only under certain conditions in a malicious manner (e.g. on a special date), one needs to analyze the machine code in detail to detect such behaviour. Here, static analysis methods can come to an end just because of the complexity of the code. In the example in Figure 3.4, the program flow relies heavily on the contents of registers. When analyzing this with a disassembler only (which is a static method), one can’t say whether the program will take the *JA* (jump if above) at 0x0041001A or whether it executes the *CALL* at 0x0041001C instead. If the latter is the case, it is also impossible to say *where* the call would go to, based on this code snippet. One would have to retrace the execution starting from the entry point by her own, and calculate the changes of registers step-by-step until this location. This becomes quickly an impractical task for a human analyzer.

<table>
<thead>
<tr>
<th>Address</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0041000D</td>
<td>8B542408</td>
<td>mov edx, [esp+8]</td>
</tr>
<tr>
<td>00410011</td>
<td>8B03</td>
<td>mov eax, [ebx]</td>
</tr>
<tr>
<td>00410013</td>
<td>05FF000000</td>
<td>add eax, 0xFF</td>
</tr>
<tr>
<td>00410018</td>
<td>39D0</td>
<td>cmp eax, edx</td>
</tr>
<tr>
<td>0041001A</td>
<td>7702</td>
<td>ja 0041001E</td>
</tr>
<tr>
<td>0041001C</td>
<td>FFD0</td>
<td>call eax</td>
</tr>
<tr>
<td>0041001E</td>
<td>B804000000</td>
<td>mov eax, 4</td>
</tr>
</tbody>
</table>

Figure 3.4: Control flow heavily relying on registers

**General orientation**

To debug a program with OllyDbg, one can attach it to a running process (**File** → **Attach**) or start a new one (**File** → **Open**). After loading the image of the executable as well as imported libraries to memory, OllyDbg shows up the *CPU window*. It displays the disassembled machine code, the registers, the memory dump and the stack.

The machine code pane on the top-left corner contains all the information that also other disassemblers show: The memory address in the first column, the hexadecimal view of the machine code at this address and the mnemonics of this command. Furthermore, OllyDbg provides additional information that it gains from code analysis. In the first column the actual position of the EIP is highlighted by a black background (also *breakpoints* are highlighted, see next subsection). The second column start with an arrow
icon, in case this line is involved in jump operations. A down or top arrow indicates
the direction of the jump that will be executed in this line. A right arrow signals that
this address is a jump target in at least one other instruction. If one clicks on a line, a
frame below the disassembly will show further information. For instance, it will list all
the addresses that jump to the highlighted one. In the third column, OllyDbg replaces
addresses in JMP and CALL instructions with a name of the form module.functionname,
e.g. KERNEL32.GetModuleHandleA. If no name is known, the address will be displayed
instead. The fourth column leaves space for user comments and also provides comments
generated by OllyDbg for better code understanding. For example, API functions are
indicated and also their parameters, if any, are named here when they are pushed onto
the stack.

The register pane always lists all general purpose registers, the flags, the segment
registers and their current values. Values that have been changed by the last executed
instruction are highlighted red. The flags that are interesting in user mode are listed
individually with a 1 or 0 indicating their state, whereas the flags register is shown as a
whole in a hexadecimal form, which is not really human readable. However, in brackets
OllyDbg shows the semantic interpretation of the current flag setting. For example, NO
indicates no overflow, BE stands for below equal or PE for parity even\(^5\). The remaining
registers below are not necessary in this paper.

The pane in the bottom-left corner displays a hex dump of the memory like it was
already explained in section 2.1. Notice, that by clicking the labels (Hex dump and ASCII)
in the first row, one can change the representation to 16 bytes in one line and to show
Unicode instead of ASCII signs.

In the bottom-right corner the current stack is shown. The last column contains useful
hints in case the values on the stack are pointers to return addresses, function calls or
readable strings. Notice, that by default the top of the table shows indeed the top of
the stack and its address is highlighted. Moreover, memory addresses grow downwards like
in all the other panes of OllyDbg.

\(^5\)For reverse engineering it is necessary to understand that some condition mnemonics in assembly
language are just synonyms. For example, it should be clear that above is logically the same as neither
below nor equal. This is the reason why JA and JNBE are two different mnemonics for the same instruction
code: 77. But the x86 processor also refers the conditions equal and zero to the same status flag, which is
the set zero flag (ZF=1 or just Z 1 in OllyDbg’s representation). Appendix B in [15] provides a complete
list of all condition mnemonics and the status flag bits they refer to.
Navigation through the code

When a program is loaded, the EIP points to the module entry point and the program is paused (which is indicated at the very bottom-right corner). OllyDbg offers different modes to navigate through a program. An easy way is single stepping which is done by hitting F7\(^6\). The debugger will execute exactly one instruction and then pause the program again. The EIP will point to the next instruction and also all affected registers and the stack, if involved, will be changed. OllyDbg provides two alternatives for handling a function call. F7 will then step into a function, thus the next instruction will be the first in the called function. Else, F8 will step over a function call. In this case, the EIP will point to the instruction right after the CALL instruction. These and other modes can be found in the main menu under Debug. If stepped into a function, one could also use execute till return to execute the rest of the function and stop at the RET instruction (after one more single step, one would arrive at the same location where stepping over would have led to). One alternative to single stepping is to simply run the program (F9). This will end up in the termination – after the complete execution – of the debugged process. Needless to say, in case of malware analysis this is usually not what an analyst wants.

An important feature to control the program’s execution is the use of breakpoints. They can be set at any instruction. If the EIP reaches it, the execution will be paused. One can also set conditional breakpoints, that pause the program if additionally some conditions are fulfilled, e.g. a register containing a special value. Both, they are software breakpoints that are realized through inserting INT 3 commands (opcode CC) by the debugger. This results in a software interrupt with the number three, which is the reserved interrupt for breakpoints provided by the x86 architecture. Notice, that in OllyDbg’s view, the INT 3 commands are hidden. The debugger shows the machine code like it would show up when getting executed in an undebugged environment.

Besides software breakpoints, there are also hardware breakpoints. They are realized directly by the processor. The x86 architecture provides four special registers for this purpose. All kinds of breakpoints can be set by right-clicking on the appropriate address and choosing Breakpoint in the context menu.

\(^6\)A complete overview of the hotkeys is provided at http://ollydbg.de/quickst.htm (last accessed 12.01.2015).
Example: Analyzing W32.Koobface malware

In this subsection user-mode debugging is applied to analyze the Koobface malware. It is shown how OllyDbg can be used to determine what the specimen is doing when executed. This paper limits itself to some basic features during the setting up of the malware. In return, these parts are explained in detail and it is introduced what feature of OllyDbg can be deployed how to get the information that is needed for the analysis.

In malware analysis, hashes are of avail to determine what file exactly is analyzed [14]. The file examined here has the SHA1 value 06b798cf26ce07007cb5d1f2ad8b6be8c916fed9 (for all malware samples see Appendix A). Virustotal reveals whether and by which name this file is recognized by different anti-virus scanners7. It shows that for example Bitdefender knows this malware as Win32.Worm.Koobface.AM and Symantec as W32.Koobface8. Symantec also provides technical details about what the malware does and how it infects user9. There is also an elaborate paper from Trend Micro about Koobface with a lot of references10.

Koobface is infamous for his capability to spread through social networks and build a botnet out of the infected computers. According to the Trend Micro paper, it is the first malware that makes such heavily use of social networks. However, in this section the installation phase is analyzed only and network capabilities are omitted. Above all, the network adapter of the virtual machine has to be disconnected (as shown in section 2.3). When trying this analysis on one’s own, one has to take the following instructions to heart11.

1. Start the virtual machine with the network adapter turned off!
2. Run OllyDbg inside the virtual machine!
3. Let ProcessExplorer opened all the time to recognize which processes are running!

The following subsections reveal how the malware copies itself into the Windows directory, hides the new file and starts it in a new process. The presentation is subdivided

---

7https://www.virustotal.com/de/file/feba3417dc4b22146e1b428bc03904904866da785003bc43b86f4e3d41e78b3c/analysis/ (last accessed 17.01.2015)
8For the sake of convenience, we stick to Koobface here.
11One also has to take the unpacking step, first, which is explained as an example in section 3.3.
into initialization, string obfuscation I, memory setup, string obfuscation II, opening the malware’s image, copying the image, process creation, file hiding and termination.

**Initialization**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x004089D5</td>
<td>MOV EAX, DWORD PTR DS:[&lt;&amp;MSVCRT._acmdln&gt;]</td>
</tr>
<tr>
<td>0x004089DA</td>
<td>MOV ESI, DWORD PTR DS:[EAX]</td>
</tr>
<tr>
<td>0x004089DC</td>
<td>MOV DWORD PTR SS:[EBP-74], ESI</td>
</tr>
<tr>
<td>0x004089DF</td>
<td>CMP BYTE PTR DS:[ESI], 22</td>
</tr>
<tr>
<td>0x004089E2</td>
<td>JNZ SHORT koobface.00408A1E</td>
</tr>
<tr>
<td>0x004089E4</td>
<td>INC ESI</td>
</tr>
<tr>
<td>0x004089E5</td>
<td>MOV DWORD PTR SS:[EBP-74], ESI</td>
</tr>
<tr>
<td>0x004089E8</td>
<td>MOV AL, BYTE PTR DS:[ESI]</td>
</tr>
<tr>
<td>0x004089EA</td>
<td>CMP AL, BL</td>
</tr>
<tr>
<td>0x004089EC</td>
<td>JE SHORT koobface.00408F2</td>
</tr>
<tr>
<td>0x004089EE</td>
<td>CMP AL, 22</td>
</tr>
<tr>
<td>0x004089F0</td>
<td>JNZ SHORT koobface.00408E4</td>
</tr>
</tbody>
</table>

| Figure 3.5: Koobface initialization (1) |

The Figure 3.5 shows a short sequence of Koobface, debugged with OllyDbg. Since it contains conditional jumps that depend on the content of registers, we use user-mode debugging to examine what it does.

The first three lines essentially get a global variable called _acmdln. Looking it up in MSDN reveals that it stores the command line\(^\text{12}\). After executing these lines, ESI contains a pointer to it and EAX contains a pointer to the value in ESI. Notice how EAX is accessed in the second line. Not the value in EAX is copied to ESI but the value that the value in EAX points to. This is called indirect addressing and indicated by writing EAX in the brackets [ and ]. In the register pane, OllyDbg shows (behind the entry of ESI) the ASCII value of the pointer target, which is "C:\koobface.exe".

One could start the program with arguments passed to maybe get things clearer. To do so, toogle a breakpoint at 0x00489D5 with the hotkey F2. Now, go to Debug → Arguments and enter a b c as arguments. To take effect, we have to restart the program with Crtl+F2. Now, we check the breakpoint which should still be there. Run the program and it gets paused where we wanted to. After the first three lines, ESI now indeed points to "C:\koobface.exe" a b c.

---

Figure 3.6: Koobface initialization (1a)

These two lines execute a conditional jump. The first byte of the value that ESI points to is compared to 22 which is the hexadecimal value for the " symbol in ASCII. The mnemonic JNZ is a synonym for JNE (which would probably be more comprehensible here). If the string would not start with a " symbol, the program jumps away. To see what would happen then, one could highlight the line and press Enter. That will follow an address without executing anything. To easily get back, one can double-click on the EIP value in the register pane.

Figure 3.7: Koobface initialization (1b)

The rest of the code is a loop which OllyDbg indicates with a big bracket (just adumbrated with ASCII art in Figure 3.7). Incrementing ESI, which was the pointer to the command line string, lets ESI point to the next sign in the string (that is C). Then, the new ESI is put somewhere on the stack. C is moved to the AL register. Before executing the CMP AL, BL, a view to the pane below the machine code is useful to easily understand what will happen. BL is zero and AL contains 43 (which is still our C, what OllyDbg kindly shows). Therefore, the jump in the next line will not be taken what OllyDbg also indicates. Because AL does not contain a " symbol, the jump back to 0x00489E4 is taken (see last two lines). The loop goes into the next round.

String obfuscation I

Koobface does not contain many human-readable strings that e.g., could be stored in the .data section as in benign applications. Therefore, a static analysis that looks for strings will come to an end here. The author of Koobface implemented an own, small routine to build up strings not until runtime. The reason for such an approach is presumably to
fool analysis. Techniques that are not malicious features but try to hamper, or even to mislead the analysis are called *obfuscation* techniques [21].

![Image of Koobface string obfuscation](http://msdn.microsoft.com/en-us/library/ybk95axf.aspx)

**Figure 3.8: Koobface string obfuscation**

In case of Koobface, the programmer used the standard C function `sprintf` to build up a *format string*. Figure 3.8 shows a code extract that uses this approach and how OllyDbg is representing it. Like one can see, OllyDbg recognizes the `sprintf` function and names it in the CALL instruction. It even provides comments where the parameters are selected and named. If possible, OllyDbg will even fill in the values of the parameters, like the "07" in line 00406FFF.

While reverse engineering, one often has to look up library functions to understand the behaviour of the specimen. Again, MSDN gives an explanation of what the function does and how parameters are used\(^\text{13}\).

**Memory setup**

![Image of Koobface memory setup](http://msdn.microsoft.com/en-us/library/ybk95axf.aspx)

**Figure 3.9: Koobface memory setup**

\(^{13}\)http://msdn.microsoft.com/en-us/library/ybk95axf.aspx (last accessed 15.01.2015)
The `memset` function is used to overwrite \( n \) bytes with the character \( c \) starting at address \( s \). The first line sets ESI to 0x3FF. The XOR ESI, ESI in the second line is a short way to clear ESI (which becomes zero). This analysis is already done by OllyDbg. As one can see, at the place where these registers are used as parameters (\( n \) and \( c \) here), OllyDbg prints out their values in the comment section. Only the value of parameter \( s \), which is delivered by PUSH EAX here, is not shown by the debugger. Probably because it is an address in memory. It is precalculated by the LEA instruction in line 00403A6F. After the call of `memset`, the caller cleans up the stack by adding 0xC. 0xC is 12 in decimal, thus it is removing three parameters of length four bytes.

While single-stepping through the code above, OllyDbg will insert the missing value of parameter \( s \) into the comment section as soon as it reaches line 00403A76. In our case, the value of \( s \) is 0012C38A. Now, one can use another feature of the tool to verify what is done by the malware. OllyDbg has a Follow in dump command that can be chosen after right-clicking on a line that contains a memory address. In our case, one can use the value of EAX in the register pane. Now, the memory dump pane begins with address 0012C384. This section contains several bytes that soon will be overwritten. After executing the CALL, this section should consist of zeros only.

### String obfuscation II

![Code snippet](image)

In next part of the code (Figure 3.10), the malware uses the `sprintf` trick again to build up a string. This string is `c:\windows\nl07.exe` which can easily be obtained.

---

14The original declaration is `void *memset(void *dest, int c, size_t count)`. OllyDbg sometimes provides slightly different parameter names. For convenience, this paper sticks to the latter. Also notice, that the parameters are passed in reverse order (CDECL). [http://msdn.microsoft.com/en-us/library/aa246471(v=vs.60).aspx](http://msdn.microsoft.com/en-us/library/aa246471(v=vs.60).aspx) (last accessed 15.01.2015)

15This and the fact of passing the parameters in reverse order indicate the CDECL calling convention as explained in section 2.1.

16Notice, that the function name is overwritten in the representation here to fit into the page layout. At the end the instruction column is overlapped by the comments – where the name is still readable.
from the disassembly thanks to OllyDbg’s comments. Obviously, this should be or become a path to some file. EAX is used to deliver the pointer, and because it was reloaded by LEA with 0012C384 as in the part before, one can track the effect of the function call at the same position in the memory dump pane. Right after the CALL, the ASCII string containing the file path will show up there.

Opening the malware’s image

```
00403A9E \t. 8D85 F4F7FFFF LEA EAX,DWORD PTR SS:[EBP-80C]
00403AA4 \t. 57 \tPUSH EDI ;/BufSize
00403AA5 \t. 50 \tPUSH EAX ;\PathBuffer
00403AA6 \t. 56 \tPUSH ESI ;\hModule
00403AA7 \t. FF15 40A04000 CALL DWORD PTR DS:[<&KERNEL32.Get\GetModuleFileNameA
00403AAD \t. 8B3D 40A14000 MOV EDI,EDWORD PTR DS:[<&MSVCRT.fopen msvcrt.fopen
00403AB3 \t. 8D85 F4F7FFFF LEA EAX,EDWORD PTR SS:[EBP-80C]
00403AB9 \t. 68 40C04000 PUSH koobface.0040C040 ; /mode = "rb"
00403ABE \t. 50 \tPUSH EAX ; |path
00403ABF \t. FFD7 CALL EDI ; \fopen
00403AC1 \t. 8BD8 \tMOV EAX,EAX
00403AC3 \t. 59 \tPOP ECX
00403AC4 \t. 3BDE \tCMP EAX,ESI
00403AC6 \t. 59 \tPOP ECX
00403AC7 \t. 0F84 9A000000 JE koobface.00403B67
```

In the next part, the examined file is calling two functions: The `GetModuleFileNameA` function from the Windows API and the C standard function `fopen`. The API function is imported from kernel32.dll and needs three parameters. BufSize will contain the size of the buffer used. The parameter in the middle, PathBuffer, will contain the output of the function. The third parameter, hModule, has to contain the handle to the executable that the calling process wants to have the file name of. If it is zero – like in our case – the functions “returns the path for the file used to create the calling process”\(^\text{17}\). PathBuffer points to 0012BF84. Following that address in memory reveals that it holds C:\malware\koopface.exe after the CALL (this value depends on from where one executed the malware).

Then the address of `fopen` is loaded to EDI (line 00403AAD). EAX is reloaded with 0012BF84 (which now points to C:\malware\koopface.exe). Now, EAX and rb are put

\(^{17}\text{http://msdn.microsoft.com/en-us/library/aa909227.aspx (last accessed 15.01.2015). Here the ASCII version of the function is used, indicated by the A in the name.}
onto the stack as parameters. The rb mode opens a file for read access and in binary mode. Right after CALL EDI, the new handle is gained by the malware and shows up in View → Handles. If one has ProcessExplorer openend with the lower pane visible, one will obtain the new handle also there.

The remaining five lines are register operations and prepare a conditional jump. Maybe, the order of the instruction is confusing, especially for an untrained eye. First, the return value of fopen is moved the EBX. Then, a value is popped to ECX (and implicitly the ESP is reduced by four). After that, a CMP is executed between the return value and ESI, which still holds the zero (since it was never changed). Now, there is not a jump, but another POP ECX (and ESP gets reduced by another four bytes). Only now, the JE occurs. What one can see here, is with high probability caused by compiler optimization. It is called interleaved code [9]. It is no problem to execute a POP between a CMP and a conditional jump because POP does not affect the setting of the EFLAGS register [15]. Therefore, the JE will act still based on the result of the comparism. In our case, the jump is not taken because fopen did not return a zero, in other words, the file opening succeeded.

Copying the image

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>00403AEA</td>
<td>FF75</td>
<td>PUSH DWORD PTR SS:[EBP-4]</td>
<td>/size</td>
</tr>
<tr>
<td>00403AED</td>
<td>FF15</td>
<td>CALL DWORD PTR DS:[&lt;&amp;MSVCRT.malloc&gt;]</td>
<td>\malloc</td>
</tr>
<tr>
<td>00403AF3</td>
<td>53</td>
<td>PUSH EBX</td>
<td>/stream</td>
</tr>
<tr>
<td>00403AF4</td>
<td>8945</td>
<td>MOV DWORD PTR SS:[EBP-8],EAX</td>
<td>l</td>
</tr>
<tr>
<td>00403AF7</td>
<td>FF75</td>
<td>PUSH DWORD PTR SS:[EBP-4]</td>
<td></td>
</tr>
<tr>
<td>00403AFA</td>
<td>6A 01</td>
<td>PUSH 1</td>
<td></td>
</tr>
<tr>
<td>00403AFC</td>
<td>50</td>
<td>PUSH EAX</td>
<td></td>
</tr>
<tr>
<td>00403AFD</td>
<td>FF15</td>
<td>CALL DWORD PTR DS:[&lt;&amp;MSVCRT.fread&gt;]</td>
<td>\fread</td>
</tr>
</tbody>
</table>

Figure 3.12: Koobface copying the image (1a)

In the part shown in Figure 3.12, the malware is allocating memory in the heap with malloc. The size parameter here is set to 0xD000. The space is allocated at 009F0048 which we know because it is the return value of malloc. Again, it is possible to see the upcoming changes through following that address in the dump. fread is set up with the handle to the image of the actual executing malware (held in EBX, used as stream), the number of items to read (n = 0xD000) and the pointer to the allocated memory (ptr

---

Here, `size` is the parameter that defines the size of an atomic item. It is set to 1 here which means the file is read byte-wise. After the call, one can see the beginning of the PE header in the dump pane starting with the magical value `MZ`.

![Figure 3.13: Koobface copying the image (1b)](image)

What Koobface is doing in Figure 3.13, is to copy the content that was saved to heap into the newly created file `c:\windows\n107.exe`. For this purpose, the earlier created string is used. The path is copied from `0012C384`. Notice, that until the `CALL EDI` in line 00403B26 (which holds the address of `fopen`) there is neither a handle to the file, nor the new file created in the Windows directory. After this call, when checking `n107.exe` in the Windows directory, one can observe that it has size zero. After the call to `fwrite` is executed, it contains 52 KB which are `0xD000` bytes.

### Process creation

Now, that the Koobface malware has copied itself, it wants to create a new process that executes the file image at `C:\Windows\n107.exe` (not shown as a code example). For this purpose, it uses the API function `CreateProcessA`[^21]. Again, OllyDbg identifies values that are pushed onto the stack as parameters and names them right before the function call. At the moment, when the EIP points to the CALL, one can see all parameters and their values in the stack pane. The Figure 3.14 shows the current setting build up by the malware.

[^20]: OllyDbg is apparently not able to recognize calls to registers in the same manner as calls to library functions. The missing information is added here in the comments starting with `;#`.

Since we know that the file `nl07.exe` is an exact copy of the executable that we are examining with the debugger, we do not want to get it started in an uncontrolled manner. There are a couple of possibilities to deny the process creation. This paper presents two of them that both make use of OllyDbg’s capability to modify the currently active memory.

The first option is to take advantage of a Windows feature that allows to start a process (and its initial thread) in a wait state called suspended state [20]. This can be enabled in the `CreationFlags` parameter. In order to do so, one has to set the `0x4` (CREATE_SUSPENDED) flag. In OllyDbg this can be done with a right-click on the corresponding line in the stack pane and choosing Modify then. After changing the hexadecimal value from `00000000` to `00000004`, OllyDbg even changes the line in the stack pane to the following:

```
0012C718 00000004 |CreationFlags = CREATE_SUSPENDED
```

The second option is to run the new process in a debugged environment. It is done by modifying the address that contains the `CommandLine` parameter. In our case, it is the address `0012E634`. One can follow it in the dump and then right-click on the first byte (63 here) and choose Binary → Edit (or Ctrl+E). After deactivating Keep size, one can enter `c:\ollydbg\ollydbg.exe c:\windows\nl07.exe`.

After the process creation, one can observe the new process in ProcessExplorer. In case of option one, there will be a process `nl07.exe` in waiting state. Following option

---


23The path to `ollydbg.exe` has to be adapted to the reader’s installation path.
two will result in a new OllyDbg process with n107.exe as a child process. The malware copy is also not executed because, by default, OllyDbg pauses the process at the entry point.

File hiding

As shown in Figure 3.15, ESI is loaded with the ASCII value Hidden. Two instructions later there is a call to kernel32.Sleep and the PUSH right before is the corresponding parameter. That line starts with an right-arrow (>) which is Olly’s sign to indicate that this address is a jump target. The info pane reveals that these jumps will happen a few lines later. To make things less confusing, a label timeout is added. The next instruction performs a logical AND on [EBP-4] and 0, which is a short way to clear the four bytes below EBP. This can also be tracked in OllyDbg. One has to right-click into the stack pane, select Go to EBP and scroll up one line. It is necessary to right-click and select Lock stack. Otherwise, the pane would jump back to ESP right after the next instruction. Now, it can be seen that the stack at 009EFF7C is filled up with four zero-bytes. To switch back to the default behaviour, one has to Unlock stack. The following instructions prepare another version of the sprintf trick. This time, the resulting string is:

Figure 3.15: Koobface file hiding (1a)
As a result of the code shown in Figure 3.16, `RegOpenKeyA`\(^{24}\) writes the handle of the opened registry key into the address that is delivered as the parameter `pHandle`. Subkey is assigned to the result of the aforementioned function `sprintf` (which wrote the format string to the address `[EBP-3F8]`). Notice, that OllyDbg recognizes `80000001` as the handle of the root key `HKEY_CURRENT_USER`. After executing the call to `RegOpenKeyA`, the malware has a handle to the registry key whose handle number was written to `[EBP-4]`. Otherwise, a zero would indicate an error and the malware would jump back to the `timeout` address (last line). Thus, it would try again to get a handle after waiting 200 ms.

Next, \texttt{RegQueryValueEx}\textsuperscript{25} is used to retrieve type and data of the key \texttt{Hidden} in \texttt{SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced}. \texttt{RegQueryValueEx} comes with six parameters and seems a bit confusing when looking it up in MSDN. Two are input parameters, two are output parameters, one is in \textit{and} out, and one is reserved. Here, all parameters are held by addresses below the EBP. Furthermore, they all are contiguous. This gives us the opportunity to see them and their changings at a glance: If the stack pane focuses on EBP, one can right-click and choose \texttt{Address \rightarrow Relative to EBP}. This leads to such a view (which is supplement by comments):

\begin{center}
\begin{tabular}{ccc}
\textbf{BEFORE} & \textbf{AFTER CALL} \\
EBP-10 & 009EFF70 & 00000004  \\
EBP-C & 009EFF74 & 00000008  \\
EBP-8 & 009EFF78 & 77BE2070  \\
EBP-4 & 009EFF7C & 0000004C  \\
EBP==> & 009EFF80 & 009EFFB4 \\
\end{tabular}
\end{center}

\textbf{Figure 3.18: Koobface file hiding (1d)}

After a call of \texttt{RegSetValueExA} (not shown here), EAX contains zero which means the function worked successful\textsuperscript{26}. As one can see in Figure 3.18, \texttt{Buffer}, which is the most interesting value, contains 2 now. This results in hidden files not being shown anymore in Windows folder views. The malware tries to hide its image from the user. This can be manually undone in the usual folder options (but if the malware is executed or debugged the next time, this will change again).

\textbf{Termination}

After creating a new instance of itself, the malware deletes the originally executed file image and terminates the running process. The deleting is done via a batch file named 355674543.bat located in directory C:\. The batch file has the content shown in Figure 3.19.

Essentially, it tries to delete the originally image (the path can differ depending on the location of the file). If the file cannot be deleted although it exists, the program will


\textsuperscript{26}Strictly speaking, the MSDN says: “If the function succeeds, the return value is ERROR\_SUCCESS.” But ERROR\_SUCCESS is Windows way to indicate that an operation succeeded. It is represented by 0. For error codes see: http://msdn.microsoft.com/en-us/library/windows/desktop/ms681382(v=vs.85).aspx (last accessed 17.01.2015).
jump back to to the label VZ8 indicated with the : sign. If the removal was successful, the batch file itself is also deleted. A line starting with REM is a comment line\textsuperscript{27}.

3.3 Decryption and unpacking

It is not uncommon that malware is encrypted or packed (or even both)\textsuperscript{28}. Thereby, the malware author tries to hamper the detection of the program and to hide malicious features from analysis. Sometimes only little parts are encrypted, sometimes the whole malware. Information that one could try to hide in particular, because they are very sensible, include file names of infection targets (e.g., in case of viruses), identifications of command and control servers (e.g., in case of backdoors and botnets) or used system functions. Methods used to encrypt malware reach from simple substitution ciphers to the use of cryptographic standard ciphers. Even multiple layers of encryption are not uncommon. Unless explicitly stated, all information in this section are taken from \cite{21}.

Since the encrypted or packed code (or data) is used somewhere in the malware, it has to be decrypted or unpacked while running. Therefore, both techniques can fool some (not all) static analysis methods but they can be detected and inverted during dynamic analysis. This is because the encrypted/packed code has to be decrypted/unpacked and is at least temporarily written into memory during the execution. If one halts the execution at such a point, the analyst will be able to read the plain text. This is the idea that also automated approaches like \cite{6} are based on. Code that is decrypted and executed later is referred to as \textit{self-modifying code}.

\textsuperscript{27}http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/rem.mspx?mfr=true (last accessed 26.01.2015)

\textsuperscript{28}This paper treats packing in terms of compression (like e.g., in \cite{21}) which is usually done by external tools (i.e., packers) after compilation. Keep in mind, that there are other conventions that understand packing as the umbrella term of encryption and compression, as in \cite{6} or \cite{11}.
Simple encryption

The use of encryption in malware is not a new development. According to [21], the first occurrence was with the Cascade virus for DOS. It implemented a simple substitution cipher making use of the XOR instruction. This approach was in a very similar manner used in the 32-bit virus Mad. Its implementation is documented in the figure 3.20, taken from [21]. XOR is very common for simple encryption schemes because it is really easy to implement and it allows to proceed encryption and decryption with the same routine\(^{29}\).

```assembly
MOV EDI, 00403045
ADD EDI, EBP
MOV ECX, 0A6B ; length of the encrypted virus body
MOV AL, [key]

Decrypt:
XOR [EDI], AL ; decrypt body
INC EDI ; adjust EDI
LOOP Decrypt ; jump back if ECX is not zero
JMP Start

DB key 86
Start: ; start of the virus body
```

Figure 3.20: Decryptor of the W95/Mad.2736 virus [21]

\(^{29}\)This follows from \((A \ XOR \ Key) \ XOR \ Key = A\).
INC EDI  
EDI is incremented to let it point to the next byte.

LOOP Decrypt  
LOOP automatically decrements the ECX register and compares it to zero. If it is not zero, LOOP will jump back to the address given as the operand. Otherwise, the loop is terminated.

JMP Start  
The decryption is terminated and the execution will be continued at the beginning of the decrypted part.

**Evolutionary encryptions**

Nevertheless, an encrypted malware can easily be detected with anti-virus scanners because of its static construction of the decryptor. One could easily determine a *signature* that would match variants of the virus, regardless of how the body looks like after encryption. For this reason, malware authors started to also change the implementation of the decryptor. The goal was to elude signature based anti-virus products while retaining the same functionality of the decryptor.

In a first step, this led to what the malware research calls *oligomorphic viruses*. This generation of viruses have the capability to modify their decryptor slightly (while keeping the functionality). For this purpose, e.g., instruction substitutions or instruction reorderings are employed.

```
MOV EAX, 0
XOR EAX, EAX
SUB EAX, EAX
```

Figure 3.21: Examples of instructions for zeroing EAX

The next step were viruses whose decryptor can take millions of shapes. These are called *polymorphic viruses*. Additionally to oligomorphic viruses, they for example applied register displacements and the insertion of *junk instructions* to mutate their code. The former is achieved by changing a register that holds one value over several lines of code. It is just replaced by another register in all instructions where this register appears. It must be ensured that this is done in closed context and the register is used in later instructions for a different purpose. Otherwise, the functionality of the code would change with high probability. Such a closed context could be e.g., a subroutine.

---

30 Changing the binary code for new variants poses primarily a challenge for viruses because above all they are *self-replicating* malware. This is why here the term *virus* is used instead of malware.
Junk instructions are such that do not change the control flow a program. Thus, their existence is needless for the correct execution. The Figure 3.22 shows different variants of such instructions inserted into a subroutine that simply adds 5 to the parameter passed onto the stack (lines 0, 3 and 9 are necessary for this functionality). Inserting junk can be done by a single instruction with no effect (line 2, 4, 5) or by multiple instructions that cancel each other out. They can be contiguous (lines 6 and 7) or even distant (lines 1 and 8). The later works as junk here because the context between those lines does not rely on the direction flag \( DF \). Of course, it is imaginable to insert bigger parts of junk code that do not affect the malicious functions. But new instructions also have the drawback to increase the file size, which is usually not wanted by malware authors. Moreover, if inserted parts are too long and constant, they will be recognizable for signature based detections again.

```
0  MOV EAX, [ESP + 8]  ; get parameter from stack
1  STD  ; set direction flag
2  XCHG EAX, EAX
3  ADD EAX, 5  ; add five to parameter
4  NOP  ; no operation
5  MOV EDX, EDX
6  PUSH ECX
7  POP ECX
8  CLD  ; clear direction flag
9  RET  ; return with result in EAX
```

Figure 3.22: Examples of inserted junk instructions

The latest development in the direction of evolutionary viruses are metamorphic viruses. Polymorphics have changing decryptors but a constant virus body (when looking at it as plaintext). Metamorphic viruses do not have a decryptor part and a body part. Instead, they use code mutations to create new variants of itself (while keeping its functionality). Therefore, metamorphic are actually not encrypted viruses. The methods polymorphic viruses use to modify their decryptor – and others – are taken to modify their complete code base. This is the reason why metamorphic viruses are sometimes called body-polymorphics. The Figures 3.23 and 3.24 show an example of the metamorphic virus W32/Metaphor’s capability to generate the same function in different ways. Although both Figures look very different, it implements the same function (which aims to get the address of \( \text{kernel32.dll} \)). The example is taken from [18].
Unpacking

Packing is originally developed to decrease the file size of executables. This involves the reorganizing of code and has the side-effect of scrambling the binary code. The unpacking is done at run time in memory. Therefore it is not surprising that malware authors make heavily use of packers. Thus, the image of the binary gets smaller and packing comes across as an obfuscation technique. Static analysis methods like searching for strings and API functions can be bypassed. In [21], Szor states that about 90% of the viruses were packed at the time of the writing of the book. Malware analysis has to handle packing to get binaries unpacked again. Unfortunately, not all packers provide the option to unpack packed executables again.
If common packers are applied, one will have a good chance to determine it with the help of static analysis tools. A rampant tool for this purpose is PEiD. The analyst can load the suspicious file into it and PEiD tells whether it found a known packing algorithm. In case the malware author implemented an own packing strategy or modifies an existing one, PEiD will probably fail. One has to employ other techniques then, e.g. user-mode debugging.

**Example: Unpacking W32.Koobface with UPX**

A widely spread packer is UPX. Fortunately, UPX also allows to unpack executables again. This can be done with the following command: `upx -d example.exe`.

```plaintext
C:\> dumpbin /headers koopface.exe
PE signature found

SECTION HEADER #1
  UPX0 name
  A000 virtual size
  1000 virtual address
   0 size of raw data
Summary
  A000 UPX0
  5000 UPX1
  1000 UPX2

C:\> upx -d koopface.exe

File size  Ratio   Format  Name
53248 <- 22528 42.31%  win32/pe  koopface.exe

C:\> dumpbin /headers koopface.exe
PE signature found

SECTION HEADER #1
  .text name
  80F6 virtual size
  1000 virtual address
  9000 size of raw data
Summary
  2000 .data
  2000 .rdata
  9000 .text
```

Figure 3.25: Shortend output of PE header before and after unpacking
Example: Decrypting Max++

First Layer

The Max++ malware uses a similar approach to the W95/Mad virus for its first layer of encryption.

```
00413A2B  AD    LDS DWORD PTR DS:[ESI]
00413A2C  33D0  XOR EDX, EAX
00413A2E  2BC2  SUB EAX, EDX
00413A30  AB    STOS DWORD PTR ES:[ESI]
00413A31  3BFD  CMP EDI, EBP
00413A33  7D F6 JCE SHORT 00413A2B
00413A35  C3    RETN

Figure 3.26: Decryptor of the W95/Mad.2736 virus
```

LODS DWORD PTR DS:[ESI] LDS is the load string instruction. It loads a value from an address held in ESI into EAX (or AX or AL depending on the size of the value). Therefore, important to notice here is not the register but the keyword DWORD which indicates that four bytes are transferred. LDS is also automatically adjusting the address in ESI depending on the number of bytes transferred. If the direction flag (DF or just D in OllyDbg) equals 1, ESI is decremented. Otherwise it is incremented.

XOR EDX, EAX Here, EDX is the key which is applied on the new value in EAX.

SUB EAX, EDX Now, EDX is subtracted from EAX and the new value is written to EAX.

STOS DWORD PTR ES:[ESI] STOS, store string, is the inverse instruction to LDS. It stores the value held in EAX (AX, AL, respectively) to the destination EDI. Analogously to LDS, here EDI will be decremented (incremented) if DF is 1 (0).

CMP EDI, EBP This comparism between EDI and EBP prepares a conditional jump. Because EDI is changed during this code snippet and EBP is not, one can reason that EBP must hold the address of the end of the encrypted part.
This jump is executed under the condition that EDI is greater than or equal to EBP. The jump target is the first line of our code snippet.

Otherwise the routine will return.

The first time one breaks the execution at the first line of this decryption routine, one can observe the following register setting: EBP = 00413A40, ESI = EDI = 00413BAC. Based on the analysis above, one can reason that the encrypted area is located from 00413A40 to 00413BAC and the decryption is done backwards, from higher to the lower addresses. One can watch how the code is changing via following in dump.

Halting the execution at the RETN instruction will allow to view the decrypted code in memory. Because the corresponding area starts almost right below the snippet shown, one can observed the changed byte represented as DB 53 for example. OllyDbg interprets this area as data. Since we assume that it will be executed as code, it is necessary to change its representation. This can be done by marking the whole area and than right-click Analysis → During next analysis, treat selection as → Commands. This will lead to a disassembly representation. Now, one can additionally choose Analysis → Analyse code which will also recognize and mark loops even if the context outreaches the highlighted selection (which is the case in our example).

Second Layer

In contrary to W95/Mad, the decryption explained above is just the first step in Max++’s decryption process. It reveals an additional decryptor. This second decryptor is way more sophisticated than the first one. This second layer, which is analyzed in the following, is only mentioned but not further described in tutorial 6 of [11]. The overall functionality is to decrypt and overwrite a big part of its own code base. The processed code is laid from 00401018 to 00413A18 which are 76288 bytes (74,5 KB). This represents 96% of the complete executable.

The routine consists of several loops. The Figure 3.26 shows an abstract overview about the overall structure of the routine. The complete disassembly is shown in Appendix A. The routine uses the stack region to operate on data. Values in this region are accessed by calculating offsets based on one or two registers and hard-coded values. This construction makes it harder to gain an overall understanding about what this part of code does. For a better analytical understanding, this region on the stack can be devided into three different areas. Each area is of size 256 bytes. These areas are located and roughly explained as follows:
Area 1 from 12FCB0 to 12FDAF  First lookup table for substitution
Area 2 from 12FDB0 to 12FEAF  Second lookup table for substitution
Area 3 from 12FEBO to 12FFAF  Temporarily buffer for the actually processed code block

```plaintext
before(); // initializes area 1
outer {
    inner1 { } // substitutes area 1 values with values from area 2
    rep1 { } // copies next code block to area 3
    inner2 { } // substitutes area 3 values by area 3 XOR area 2
    inner3 { } // substitutes area 3 values with values from area 1
    rep2 { } // writes area 3 (processed code block) back to memory
}
```

Figure 3.27: Structure of the 2nd decryptor of Max++

To get an impression how looped structures work, one can use a feature of OllyDbg that is called tracing. When tracing a part of code, OllyDbg will debug the executable like it would when running it. But additionally, the debugger will store all executed instructions and register changings. Furthermore, it counts how often an instruction is executed. Just as with running, the tracing will be paused if a breakpoint or the end of the executable is reached\(^31\). In our example we will set a breakpoint at the first instruction of outer and start it with Debug → Trace into. The program’s execution gets paused the first time after executing the lines before the beginning of outer which includes the function call to before. We will choose Trace into again to also execute one round of outer. Now, one can view the run trace profile. To do so, one has to choose View → Executables modules and than right-click Max++ and click View run trace profile. By default the output is sorted descending by counts. The output shown below was sorted by addresses (by clicking on the top of that column). This will lead to a more chronological order of the lines, in our case.

\(^{31}\)Notice, that because of the overhead, tracing takes much longer than running. Tracing the complete outer loop will take several minutes.
The debugger divides the executed code into parts by itself. In run trace profile view, it will only show the first line of such a part. This allows a better overview for the analyst. If one compares the output with the abstract structure shown above, one will determine each loop inside `outer` clearly. Here it is also indicated by added comments. It is recognizable that these loops are executed multiple times. The parts between the loops are executed only once. Because they just prepare the loops and do not provide essential decrypting functionality, they were left out in the abstract structure above. One can also observe that the loops called `inner` are executed 256 times whereas the `rep`’s are executed only 64 times.

<table>
<thead>
<tr>
<th>COUNT</th>
<th>ADDRESS</th>
<th>FIRST COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>00413A44</td>
<td>MOV EBP,ESP</td>
</tr>
<tr>
<td>256.</td>
<td>00413A6F</td>
<td>MOVZX ECX,BYTE PTR DS:[ESI] ; begin of inner1</td>
</tr>
<tr>
<td>1.</td>
<td>00413A82</td>
<td>PUSH 40</td>
</tr>
<tr>
<td>64.</td>
<td>00413A8D</td>
<td>REP MOVSD DWORD PTR ES:[EDI],DWORD PTR DS:[ESI] ; rep1</td>
</tr>
<tr>
<td>1.</td>
<td>00413A8F</td>
<td>XOR DL,DL</td>
</tr>
<tr>
<td>256.</td>
<td>00413A97</td>
<td>INC BYTE PTR SS:[EBP-1] ; begin of inner2</td>
</tr>
<tr>
<td>1.</td>
<td>00413AE5</td>
<td>MOV EBX,DWORD PTR SS:[EBP-8]</td>
</tr>
<tr>
<td>256.</td>
<td>00413AEC</td>
<td>MOVZX EDX,BYTE PTR SS:[EBP+ECX-20D] ; begin of inner3</td>
</tr>
<tr>
<td>1.</td>
<td>00413B01</td>
<td>MOV EDI,EBX</td>
</tr>
<tr>
<td>64.</td>
<td>00413B0E</td>
<td>REP MOVSD DWORD PTR ES:[EDI],DWORD PTR DS:[ESI] ; rep2</td>
</tr>
<tr>
<td>1.</td>
<td>00413B10</td>
<td>MOV DWORD PTR SS:[EBP-8],EBX</td>
</tr>
</tbody>
</table>

----------------------------------------------------------------------------

| 1.    | 00413B21 | PUSH EBP ; begin of before |
| 256.  | 00413B62 | MOV DL,BYTE PTR SS:[EBP-1] |
| 1.    | 00413B70 | PUSH EBX             |
| 256.  | 00413B71 | MOVZX EAX,BYTE PTR SS:[EBP-1] |
| 1.    | 00413BA2 | POP EBX             |

Figure 3.28: Run trace profile after the first round of `outer`

| 00413A82 | . 6A 40 | PUSH 40 |
| 00413A84 | . 8DBD F4FEFFFF | LEA EDI,DWORD PTR SS:[EBP-10C] |
| 00413A8A | . 8BF3 | MOV ESI,EBX |
| 00413A8C | . 59 | POP ECX |
| 00413A8D | . F3:A5 | REP MOVSD DWORD PTR ES:[EDI],DWORD PTR DS:[ESI] |

Figure 3.29: Disassembly of `rep1`

The `rep` loops are worth mentioning whereas the other functions are not explained.
here in detail. The figure above shows the complete disassembly from the end of *inner1* until the end of *rep1*. When looking at the disassembly, one can recognize that there is neither a LOOP nor a JMP instruction. This is because a REP instruction is applied. REP (*repeat*) implies a loop and can be used with serveral string operations, **MOVS** in our case. It will be repeated as many times as specified in the ECX register. This register is decremented after each round. In each round **MOVS** will copy a DWORD from one memory location to another memory location. These location are held in ESI and EDI.

- **PUSH 40** Pushes 0x40 onto the stack which is 64 decimal.
- **LEA EDI, DWORD PTR SS:[EBP-10C]** EDI is loaded with a value from area 3.
- **MOV ESI, EBX** ESI points to the actual block of encrypted code in memory.
- **POP ECX** The 0x40 is used as the counter for REP.
- **REP MOVS DWORD PTR ES:[EDI], DWORD PTR DS:[ESI]** 64 times a 4 byte value is copied from memory to area 3.

### 3.4 Kernel-mode debugging

According to Peter Szor [21], the majority of viruses can be traced using a user-mode debugger. Nevertheless, there is malware out there that operates in the kernel address space. This can be way more dangerous because thereby the malware is capable of hiding from user-mode detection tools (e.g., ProcessExplorer) and it is able to manipulate internals of the operating system. It is also possible to start user-mode activities (e.g., creating a process) and hide these from the attacked user due to such manipulations. In this case, a user-mode debugger like OllyDbg will not help and a malware analyst has to take advantage of kernel-mode debuggers such as WinDbg.

WinDbg has a GUI which includes the **command window**. On the bottom there is a command line starting with `kd>` that is used to control the debugger. It is recommend to activate the verbose output via **View → Verbose Output** to recognize when a new module gets loaded. The command line knows a great variety of commands for memory dumping or examining processes, OS internal data types and functions. It even allows
scripting in a C-like scripting language\textsuperscript{32}. A set of scripts that is very helpful for malware analysis is provided by Lionel d’Hauenens and downloadable for free at Laboskopia\textsuperscript{33}. To install the scripts after extracting the archive, copy the folder script into the WinDbg installation folder\textsuperscript{34}. To load the scripts into WinDbg one has to type the following into the command line: `ad /q *; $$><script@@init\cmd.wdbg;` and press \textbf{Enter}.

\textbf{Navigation}

In the following, the most common commands to examine the memory are explained. They are shown in exemplarily form. Also some commands for process analysis are presented\textsuperscript{35}.

\begin{verbatim}
  da 0x77665544 Display memory (d) at address 0x77665544 as ASCII text (a).
  du 0x77665544 Display memory (d) at address 0x77665544 as Unicode text (u).
  dd 0x77665544 Display memory (d) at address 0x77665544 as DWORDS (d).
  dd 0x77665544 L3 Display memory (d) at address 0x77665544 as DWORDS (d) and only show three 32-bit blocks. The number behind L has to be a hexadecimal number.
  db 0x77665544 Display memory (d) at address 0x77665544 byte-wise (b) and with ASCII interpretation in the last column. This output looks like a “classical” hex view.
  dd MmSystemRangeStart L1 Display address and value of kernel variable MmSystemRangeStart (which is 80000000 if the kernel space is of size 2GB).
  dt _eprocess Display (d) type (t) _eprocess which is the data type of the process environment block.
\end{verbatim}

\textsuperscript{32}A well and brief tutorial about the scripting language is contained in the book \textit{Windows Crash Dump Analysis} by Dmitry Vostokov. The relevant chapter is published for free and accessible at \url{http://www.dumpanalysis.org/WCDA/WCDA-Sample-Chapter.pdf} (last accessed 31.01.2015).

\textsuperscript{33}\url{http://laboskopia.com/download/SysecLabs-Windbg-Script.zip} (last accessed 31.01.2015)

\textsuperscript{34}Notice, that it has to be one folder level above the level where windbg.exe lies. The path usually is `\..\Debuggers\x86\windbg.exe`. In this case, copy script into `\..\Debuggers\`. In this case, copy script into `\..\Debuggers\`.

\textsuperscript{35}Notice, that here the most basic instructions are listed only. To get to know the power of kernel-mode debugging with WinDbg, one has to take a look into further literature (e.g., \cite{1}).
dt _eprocess 0x77665544  Interpret memory beginning at 0x77665544 as an _eprocess structure (independent on whether it is used as such a structure or not).

? f7a  Evaluate hexadecimal value f7a as decimal value.

r edi  Display content of register edi.

u 0x77665544 L10  Disassemble memory beginning at address 0x77665544 and stopping after 16 instructions.

!process 0 0  List all currently running processes of the debugged machine. Output includes amongst others the process id (Cid), the address of the process environment block (Peb) and the name of the image.

!process 728  List detailed information about the running processes with ID 728. Output includes amongst others address space details, thread and scheduling information.

**Detecting hooks**

One important type of malware that applies kernel-space manipulations are *rootkits* (although, there are user-mode rootkits as well). Rootkits are considered as especially nefarious because of their ability to process underneath a lower layer of the operating system than other kinds of malware. According to Arnold [2], there are five main techniques that are used by rootkits that act on the level of the operating system (hardware-based rootkits exist as well): File masquerading, routine patching, insertion of filter drivers, direct kernel object manipulation and hooking. In this paper, we just focus on the last one.

Hooking is a technique that intercepts calls to certain functionalities (e.g., functions or processes) by redirecting the pointer to malicious code [2]. After executing it, the execution is continued as in the default case. Targets of hooking attacks are all files or memory addresses that contain such pointers. As briefly introduced in section 2.2, a function call of a simple API function will trigger a chain of several more calls to other functions. Every link of that chain is a potential target of a hooking attack. For example, a call to `WriteFile` (exported from `kernel32.dll`) will in detail lead to the chain shown below which is taken from malwaretech\(^\text{36}\).

\(^\text{36}\)\url{http://www.malwaretech.com/2013/09/ring3-ring0-rootkit-hook-detection-12.html} (last accessed 31.01.2015)
Hooking one of these functions will intercept all calls from the functions above but, needless to say, not the calls below. For instance, hooking KiFastSystemCall (3) will attack of course all calls of WriteFile (1). But also direct calls of NtWriteFile (2) in ntdll.dll are intercepted as well as simply all other system calls that are invoked via KiFastSystemCall. But this hook would not affect function calls that are made inside the kernel space because it is obviously a user-mode hook (also called userland hook).

**Example: Kernel-mode debugging of Necurs**

Necurs is a sophisticated rootkit that exists as a x86 (32-bit) version as well as a x64 (64-bit) version. Besides code obfuscation techniques, it also implements self-protection against anti-virus products. The information about Necurs in this subsection is taken from [3], the techniques for hook detection are taken from [5].

Since many rootkits, like Necurs, run in kernel-mode, they often come in shape of a driver. If the analyst does not have access to the user-mode part of the malware that loads the driver, one will have to use a tool like OSR Driver Loader to get it done [14]. In order to load the rootkit, one has to run the tool inside the guest machine, select the path to Necurs via Browse, click Register Service first and then Start Service.
After starting Necurs and switching to WinDbg in the host machine, one can examine the SSDT (system service descriptor table) in different ways. Using WinDbg default commands, one can run `dps KiServiceTable l11c`. This will show the addresses of the table entries, where they point to and which system service call is associated with this target address. Figure 3.31 shows the snippet of the output that is of interest in Necurs’ case.

```
804e2f04 8059ac32 nt!NtOpenObjectAuditAlarm
804e2f08 81f09e2b
804e2f0c 8056c8fc nt!NtOpenProcessToken
804e2f10 8056caf5 nt!NtOpenProcessTokenEx
804e2f14 805766cc nt!NtOpenSection
804e2f18 805a3c97 nt!NtOpenSemaphore
804e2f1c 8058770c nt!NtOpenSymbolicLinkObject
804e2f20 81f09f54
804e2f24 8056c383 nt!NtOpenThreadToken
804e2f28 8056c2f1 nt!NtOpenThreadTokenEx
```

Figure 3.31: Detect SSDT hooks

The output reveals that two addresses cannot be associated to a known system service call. Also they point to addresses that are obviously outside the default range. Both indicates that these functions are hooked. The hooked functions are `NtOpenProcess` and `NtOpenThread`. One could also use the `!!display_system_call` command from the Laboskopia scripts. In addition, it will show a label whether each function is `OK` or a `HOOK`.

As shown in Figure 3.30, there are several more places that a rootkit might hook. A prominent place would be a SYSENTER hook, in order to intercept all system service calls from user mode (position (4) in the example above). As mentioned in section 2.2, this instruction relies on the MSR special purpose registers. The MSR registers with the numbers 0x174, 0x175 and 0x176 define the target (which is `KiFastCallEntry` by default) [13]. To check their contents, the `rdmsr` command – with the specific number as the parameter – can be used. One can also use `!!display_current_msrs` from Laboskopia scripts to get further hints again. Notice that even though the SYSENTER is listed in user mode in Figure 3.30 (because it is callable from user-mode applications), the corresponding MSR registers cannot be changed from user mode. However, a kernel-mode rootkit is able to overwrite them.
Another table that might be hooked, is the IDT (explained in section 2.2). It can be checked with `!idt` standard command or `!!display_current_idt` from the Laboskopia scripts. Notice that according to [3], Necurs does not hook MSR registers or the IDT.
Chapter 4

Conclusion and further reading

This paper provided a hands-on introduction for newcomers to the field of manual malware analysis. Static and dynamic techniques were presented step by step. Enough background on assembly programming is provided for this paper to serve as a practical introduction to (dis-)assembly on x86 CPUs. In addition, Windows NT fundamentals are discussed. In conclusion, this paper might facilitate lectures on operating system security and offer an applicable starting point to examine this fields in practice. In case the reader wants to continue the studies, in the following further literature and web sites are recommended.

Practical Malware Analysis & Malware Analyst’s Cookbook

Concerning malware analysis, these two books represent the most comprehensive guides. Both books explain malware analysis techniques from scratch and in great detail, provide tons of practical hands-on explanations as well as tools, links and tricks. They describe static and dynamic techniques, introduce user- and kernel-mode debugging, deobfuscation, anti-debugging tricks and much more.

Both books are written to be highly understandable especially for people who are new in this research area. To name some differences, it is noticable that “the Cookbook” [1] mainly consists of its recipes which show practical commands, a lot of script code and tools to solve special issues (e.g. “Identifying Packers with YARA and PEiD” or “Automated Malware Analysis with VirtualBox”). In contrary, “the Practical book” [14] has fulltext chapters which explain their subject. At the end of each chapter, there are labs and crackme files with challenges to be solved by the reader. At the end of the book, there are detailed solutions for each lab. Moreover, both books cover some topics, the other leaves unresolved. For example the Cookbook has a chapter on anonymizing web activities whereas the Practical provides an x86 assembly introduction. Bottom line,
both books are highly recommended for the deepening of malware analysis knowledge. Which one to choose is a matter of taste.

**Dr. Fu’s Malware Tutorials**

As far as I know, [11] is the most comprehensive and detailed analysis of a single malware sample that is publicly available. Dr. Xiang Fu (Associate Professor at Hofstra University) analyzes the Max++ malware from scratch and chronologically in 35 parts. He also introduces how to set up an environment. This paper’s environment was geared to it, that is why the reader will have no problem to start with Dr. Fu’s tutorial. If one wants to go on with malware analysis it is highly recommended to read and work through these explanations. Because Max++ is a really sophisticated piece of malware, besides the mechanism explained in this paper, one has to handle thread injection, return-oriented programming and more kernel-mode debugging to understand it. There are also several anti-analysis tricks implemented and the first parts cover a savvy anti-debugging trick (which should not disencourage the reader).

**A Survey on Automated Dynamic Malware-Analysis Techniques and Tools**

Since this paper is limited to manual techniques, one possible way to continue studying could be automated approaches. Very recently, M. Egele et al. wrote a survey paper [8] that presents state-of-the-art analysis tools and compares their capabilities. It is worthwhile to read because of its comprehensiveness and its clarity. After a survey of malware types and dynamic analysis techniques, common tools are presented. At the end, a table displays which analysis features are provided by which tool. The paper also contains an extensive reference list.

**The Art of Computer Virus Research and Defense**

[21] is presumably the standard work in malware research and the most complete overview about attack and detection strategies of computer viruses. It was written by Peter Szor (*1970, †2013) who is a world renowned virus researcher and worked for different antivirus companies (for Symantec’s Security Response Team when the book was published in 2005).

The book focuses on viruses and leaves for example trojans and backdoors out. It provides a great classification of infections strategies, basic and advanced self-protection

---

1http://people.hofstra.edu/Xiang_Fu/XiangFu/index.php (last accessed 03.02.2015)
mechanisms (e.g., encryption) of viruses and a lot on anti-virus detection techniques. The later include signature-based, so-called “algorithmic” techniques and heuristics. It also covers network related topics like worms and intrusion detection. It reduces code examples to an absolute minimum and in return focuses on well understandable descriptions. The book lists tons of real-world examples for each technique. There is also a relatively short chapter (60 pages) about exploits starting with buffer overflows and reaching until format string attacks.

**Windows Internals**

[20] is one of the standard works about the Windows NT operating system. It was written by Mark E. Russinovich and David A. Solomon. It provides a comprehensive and detailed description of the system architecture, the memory management, the process and thread internals, file systems and security mechanisms, to name the most important parts. It comes with a lot of examples on how to dig into the described Windows internals. Often this is done with WinDbg, therefore it is a good starting point to learn more about debugging the Windows kernel. All in all, it is a reliable reference book for Windows NT fundamentals.

**Reversing: Secrets of Reverse Engineering**

[9] is a helpful beginner’s guide about reverse engineering in general in which malware analysis is just one topic amongst others and thereby not the focus. It starts with an overview of Windows fundamentals, an assembly introduction and low-level x86 architecture basics in greater detail than this paper provides. The book also covers anti-reversing mechanisms which include anti-debugging as well as anti-disassembler techniques and ends with decompilation. It provides one really detailed example on how to reverse engineer an undocumented Windows API function family with a lot of assembly code to descramble on one’s own (and with help).

**Blogs, websites and projects**

<table>
<thead>
<tr>
<th>URL</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.malwaretech.com/">http://www.malwaretech.com/</a></td>
<td>A great blog about malware analysis, system internals and security topics in general with the aim to write comprehensible.</td>
</tr>
</tbody>
</table>
http://www.reconstructer.org/ Nice website of Frank Boldewin. It contains a lot of interesting papers, scripts and links regarding rootkits and kernel debugging in particular.

http://pferrie.host22.com/ Homepage of Peter Ferrie who received the Virus Bulletin 2010 award for greatest contribution to anti-malware in the last 10 years. The site provides his articles, the vast majority related to malware. Especially, there are many analyses of single pieces of malware and a series about anti-unpacking methods.

http://www.openrce.org/ A lot of articles and hints about reverse engineering, debugging, anti-debugging, packing and so on. It has a frequently used forum and provides many downloads of scripts and plugins (e.g., for OllyDbg).

http://www.osronline.com/ A website about everything related to kernel-mode programming and WinDbg.

https://remnux.org/ A good and easy way to apply some of the techniques discussed here (and even more) on a Linux platform. It is a free toolkit that aims to assist malware analysis with reverse-engineering malicious software. It comes as a Linux distribution that is based on Ubuntu. It can even be downloaded as an .ova file that can directly be loaded in a virtualization tool like Virtualbox. In case there are problems during the installation, it is worth to check to following notes: http://zeltser.com/remnux4-installation-notes/.
Appendix A

List of analyzed malware

<table>
<thead>
<tr>
<th>Name</th>
<th>Source</th>
<th>SHA1 hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brontok</td>
<td>Openmalware.org</td>
<td>69f8ba6e92f08a1bbdd64a07041ff349f42f06df</td>
</tr>
<tr>
<td>Koobface</td>
<td>Openmalware.org</td>
<td>06b798cf26ce07007cb5d1f2ad8b6be8c916fed9</td>
</tr>
<tr>
<td>Max++</td>
<td>Openmalware.org</td>
<td>d0b7cd496387883b265d649e811641f743502c41</td>
</tr>
<tr>
<td>Necurs</td>
<td>Malware.lu</td>
<td>30f63b8cae41a97456a82131c4577a2020697b89</td>
</tr>
</tbody>
</table>
Appendix B

Disassembly of second decryptor of Max++

00413A40 | CD 2D | INT 2D
00413A42 | C3 | RETN
00413A43 | 55 | PUSH EBP
00413A44 | 8BEC | MOV EBP,ESP
00413A46 | 81EC 0C030000 | SUB ESP,30C
00413A4C | 53 | PUSH EBX
00413A4D | 56 | PUSH ESI
00413A4E | 8BD9 | MOV EBX,ECX
00413A50 | 57 | PUSH EDI
00413A51 | 8DB5 F4FDFFFF | LEA ESI,DWORD PTR SS:[EBP-20C]
00413A57 | 8BC3 | MOV EAX,EBX
00413A59 | 8955 F4 | MOV DWORD PTR SS:[EBP-C],EDX
00413A5C | EB C0000000 | CALL Max++.00413B21
00413A61 | 83C3 10 | ADD EBX,10
00413A64 | 89DF F8 | MOV DWORD PTR SS:[EBP-8],EBX
00413A67 | > 0C FF | OR AL,OFF
00413A69 | 8DB5 3FFEFFFF | LEA ESI,DWORD PTR SS:[EBP-10D]
00413A6F | > 0FB60E | MOVZX ECX,BYTE PTR DS:[ESI]
00413A72 | 88840D F4FCFFFF | MOV BYTE PTR SS:[EBP+ECX-30C],AL
00413A79 | 8AC8 | MOV CL,AL
00413A7B | FEC8 | DEC AL
00413A7D | 4E | DEC ESI
00413A7E | 84C9 | TEST CL,CL
00413A80 | 75 ED | JNZ SHORT Max++.00413A6F
00413A82 | 6A 40 | PUSH 40
00413A84 | 8DBD F4FEFFFF | LEA EDI,DWORD PTR SS:[EBP-10C]
00413A8A | 8BF3 | MOV ESI,EBX
Bibliography


