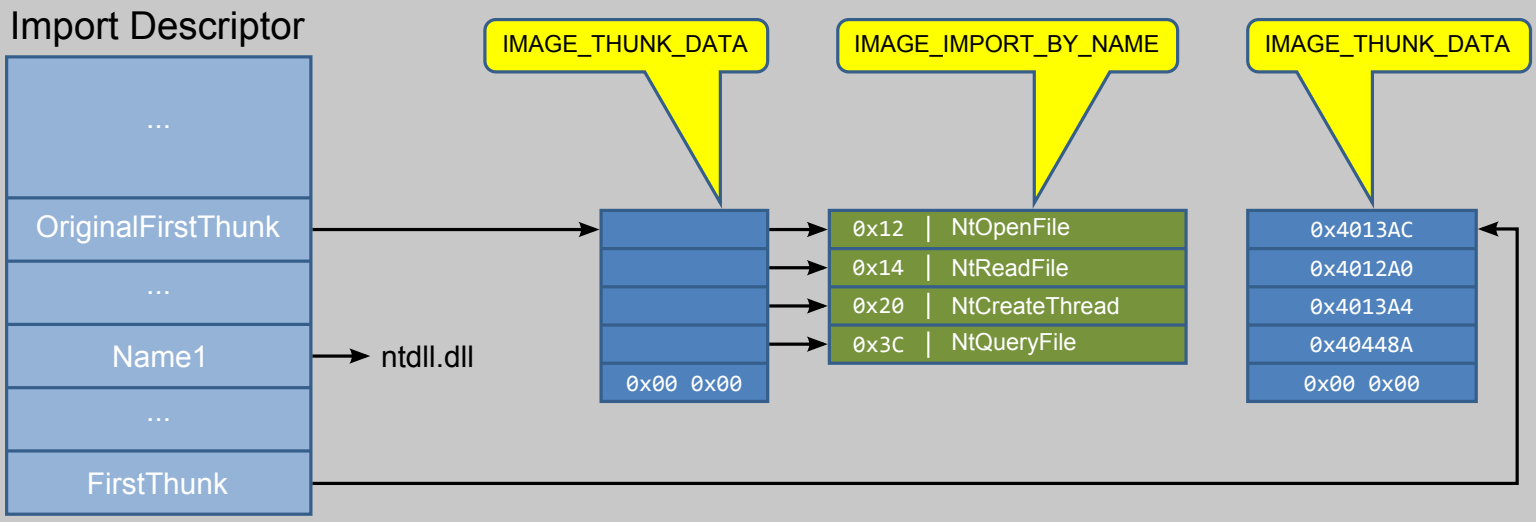
FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Open C3S
Open Competence Center for Cyber Security



Zertifikatsprogramm

Windows für Programmanalysten [MM-105]

Autoren:

Dr. rer. nat. Werner Massonne

Prof. Dr.-Ing. Felix C. Freiling

Friedrich-Alexander-Universität Erlangen-Nürnberg

Windows für Programmanalysten [MM-105]

Autoren:

Dr. rer. nat. Werner Massonne

Prof. Dr.-Ing. Felix C. Freiling

---

1. Auflage

Friedrich-Alexander-Universität Erlangen-Nürnberg

GEFÖRDERT VOM
Bundesministerium
für Bildung
und Forschung

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Martensstr. 3
91058 Erlangen

1. Auflage (9. Dezember 2016)

Um die Lesbarkeit zu vereinfachen, wird auf die zusätzliche Formulierung der weiblichen Form bei Personenbezeichnungen verzichtet. Wir weisen deshalb darauf hin, dass die Verwendung der männlichen Form explizit als geschlechtsunabhängig verstanden werden soll.

Das diesem Bericht zugrundeliegende Vorhaben wurde mit Mitteln des Bundesministeriums für Bildung, und Forschung unter dem Förderkennzeichen 16OH12022 gefördert. Die Verantwortung für den Inhalt dieser Veröffentlichung liegt beim Autor.

Inhaltsverzeichnis

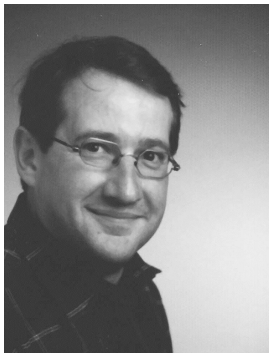
Einleitung

I. Abkürzungen der Randsymbole und Farbkodierungen

Exkurs	E
Quelltext	Q
Übung	Ü

II. Zu den Autoren

Felix Freiling ist seit Dezember 2010 Inhaber des Lehrstuhls für IT-Sicherheitsinfrastrukturen an der Friedrich-Alexander-Universität Erlangen-Nürnberg. Zuvor war er bereits als Professor für Informatik an der RWTH Aachen (2003-2005) und der Universität Mannheim (2005-2010) tätig. Schwerpunkte seiner Arbeitsgruppe in Forschung und Lehre sind offensive Methoden der IT-Sicherheit, technische Aspekte der Cyberkriminalität sowie digitale Forensik. In den Verfahren zur Online-Durchsuchung und zur Vorratsdatenspeicherung vor dem Bundesverfassungsgericht diente Felix Freiling als sachverständige Auskunftsperson.



Werner Massonne erwarb sein Diplom in Informatik an der Universität des Saarlandes in Saarbrücken. Er promovierte anschließend im Bereich Rechnerarchitektur mit dem Thema „Leistung und Güte von Datenflussrechnern“. Nach einem längeren Abstecher in die freie Wirtschaft arbeitet er inzwischen als Postdoktorand bei Professor Freiling an der Friedrich-Alexander-Universität.

III. Lehrziele

Unter Programmanalyse verstehen wir in diesem Studienbrief die Untersuchung unbekannter Software mit dem Ziel, ihre Funktionsweise zu rekonstruieren und zu dokumentieren. Im Gegensatz zum *Software Engineering*, wo es um die Übersetzung von Anforderungen in Code geht, muss bei der Programmanalyse der umgekehrte Weg gegangen werden, das Herauslesen von Anforderungen und Intentionen aus Software ist hier erforderlich. Man spricht deshalb von *Software Reverse Engineering*, oder einfach nur kurz *Reverse Engineering*. Die Hauptanwendungsgebiete von Reverse Engineering sind die folgenden:

- Software-Wartung
- Software-Analyse
- Malware-Analyse

Insbesondere die Malware-Analyse ist ein regelmäßiger Untersuchungsgegenstand in der digitalen Forensik. Im Laufe der forensischen Untersuchung eines Programms werden häufig Fragen der folgenden Art aufgeworfen, die mit Hilfe von Reverse Engineering beantwortet werden können:

1. Welche Veränderungen hat ein Programm auf einem Rechner möglicherweise vorgenommen?
2. Mit welchen Rechnern kann ein Programm potenziell kommunizieren?
3. Welche Daten liest ein Programm, während es auf einem Rechner läuft?
4. Wer hat ein bestimmtes Programm geschrieben?

In der Praxis hat man es bei Malware standardmäßig mit Software zu tun, deren Quellcode nicht verfügbar ist, die also nur in Binärform (als Maschinenprogramm) vorliegt. Für die erfolgreiche Analyse eines Maschinenprogramms sind die folgenden Voraussetzungen unabdingbar:

1. Tiefgehende Kenntnisse über die Architektur und insbesondere die Maschinensprache des Zielrechners.
2. Kenntnisse über das Betriebssystem des Zielrechners. Das Betriebssystem bildet das Bindeglied zwischen Hardware und Programm, indem es einem Programm seine Ausführungsumgebung bereitstellt. Wenn wir davon ausgehen, dass es sich bei einer Malware letztendlich auch nur um ein „normales“ Programm[1] handelt, so benutzt sie auch die normalen Gegebenheiten und Schnittstellen eines Betriebssystems. Darunter fallen insbesondere Programmaufbau, Prozessstrukturen und die Kommunikation mit den Betriebssystembibliotheken.

Microsoft Windows ist eines der am häufigsten in der Praxis eingesetzten Betriebssysteme weltweit und steht damit auch im Fokus der Malware-Autoren. Der ganz überwiegende Teil der bis heute in Umlauf gebrachten Malware wurde für das Betriebssystem Microsoft Windows geschrieben. Als Malware-Analyst müssen Sie sich daher mit Windows auseinandersetzen.

In diesem Studienbrief werden die für eine Programmanalyse wesentlichen Funktionen von Microsoft Windows erklärt. Im Mittelpunkt der Betrachtungen stehen die verschiedenen Datenstrukturen, die von Windows erzeugt und benutzt werden, um Programme auszuführen. Die Analyse dieser Strukturen liefert wesentliche Erkenntnisse über Funktion und Verhalten eines Programms.

Dieser Studienbrief soll Ihnen die Fähigkeit vermitteln, Ablauf und Aufbau von Programmen und Verwaltungsstrukturen unter Windows zu verstehen und zu analysieren. Kenntnisse zu den allgemeinen Grundlagen von Betriebssystemen werden hierbei vorausgesetzt.

[1] Bei sogenannter Kernel-Malware verhält sich das etwas anders, aber Kernel-Malware steht außerhalb der Betrachtungen dieses Studienbriefs.

Windows für Programmanalysten

1 Lernergebnisse

Sie können den grundsätzlichen Aufbau von Windows beschreiben. Sie können erklären, wie einige wichtige Betriebssystemfunktionalitäten in Windows realisiert sind. Speziell das Windows-PE-Format können Sie detailliert erklären und damit im Hinblick auf einige, für eine Programmanalyse wichtige, Aufbaudetails analysieren.

Sie können die Windows-Programmierschnittstelle (API) und den Mechanismus der Systemaufrufe erklären. Ebenso können Sie den Aufbau von Windows-Prozessen und der zugehörigen Datenstrukturen beschreiben und analysieren. Die Verfahren der Exception-Behandlung können Sie benennen und erklären.

2 Einführung

Windows-Betriebssystemfamilie

Wir betrachten hier innerhalb der Windows-Betriebssystemfamilie die 32-Bit-NT-Linie, die mit Windows NT ihren Anfang nahm und über Windows 2000, XP, Vista und Windows 7 bis zum aktuellen Windows 10 bis heute weitergeführt wurde. Der überwiegende Teil der heute im Einsatz befindlichen Arbeitsplatzrechner arbeitet mit diesen Betriebssystemen. Auch die neueren 64-Bit-Varianten unterscheiden sich im prinzipiellen internen Aufbau nicht wesentlich von den 32-Bit-Systemen. Wegen ihres bedeutenden Marktanteils sind Windows-Systeme für eine Programmanalyse besonders interessant. Im Hinblick auf eine Programmanalyse interessieren uns hierbei insbesondere die Programmierschnittstellen sowie der Aufbau von Windows-Prozessen und der Prozessdatenstrukturen.

Abb. 1 zeigt den logischen Aufbau des Windows-Betriebssystems. Prozesse, die unter dem Betriebssystem Windows laufen, gliedern sich in zwei Kategorien, nämlich in Kernel- und User-Prozesse. Die Bestandteile von Windows, die im Kernelmode laufen, sind unterhalb des schwarzen Querbalkens dargestellt.

Schichtenmodell von Windows

Im Kernelmode werden die Prozesse des eigentlichen Betriebssystemkerns, die Gerätetreiber, HAL und die elementaren internen Verwaltungsdienste ausgeführt. Zu den internen Verwaltungsdiensten zählen bspw. die Prozess- und Speicherverwaltung, das I/O-Management und das *Graphical Device Interface* (GDI). HAL (*Hardware Abstraction Layer*) ist die unterste Schicht von Windows zur Hardware hin, die direkt auf den verwendeten Prozessor und bestimmte Besonderheiten der Hardware Bezug nimmt. Diese Zwischenschicht vereinfacht die Portierung des Betriebssystems und schirmt die hardwarespezifischen Eigenschaften der Zielplattform vom Rest des Betriebssystems ab.

Systemprozesse

Im Usermode laufen zunächst einmal verschiedene Systemprozesse wie z. B. LSASS (*Local Security Authority Subsystem*). LSASS ist der lokale Sicherheitsauthentifizierungsserver. Er überprüft die Gültigkeit der Benutzeranmeldung. LSASS ist eng mit dem Winlogon-Dienst und dem Session Manager verbunden, die für das An- und Abmelden der Benutzer und die Verwaltung der benutzerspezifischen Einstellungen zuständig sind.

Dienste

Weiterhin existieren Prozesse, die Steuerungsfunktionen im System übernehmen. Sie werden deshalb Dienste genannt. Diese Dienste arbeiten im Hintergrund und kommunizieren dabei nicht direkt mit dem Anwender. Häufig gibt es zur Konfiguration und Steuerung eines Dienstes separate Programme, in Windows sind die meisten dieser Programme in der Systemsteuerung zusammengefasst. Der Dienst *spoolsv.exe* ermöglicht es bspw., Druck- oder Faxaufträge durchzuführen,

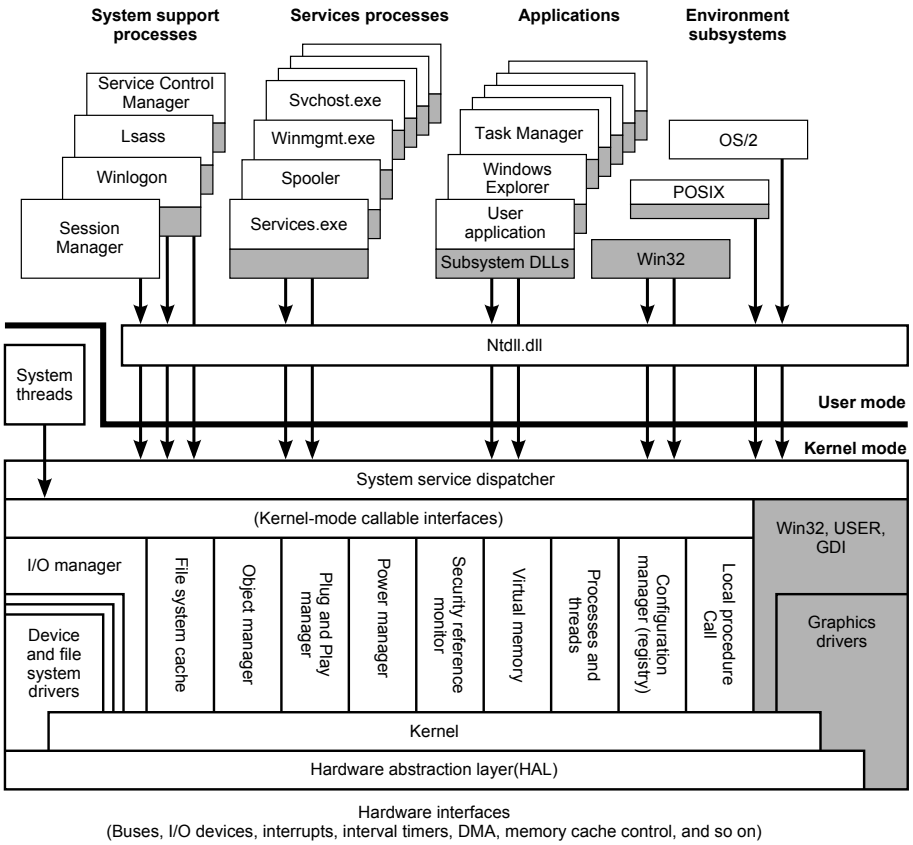


Abb. 1: Logischer Aufbau von Windows [Russinovich and Solomon, 2012]

ohne andere Arbeiten am Computer zu unterbrechen. Der übergeordnete Dienst *services.exe* verwaltet das Anhalten und Starten von Diensten.

Anwendungen

Weitere Prozesse, die im Benutzermodus laufen, sind die Anwendungen. Sie werden meist vom Benutzer selbst gestartet, sofern nicht eingestellt wurde, dass sie automatisch bei jedem Systemstart anlaufen. Explorer und Task Manager sind bekannte Beispiele für solche Anwendungen. Auch Programme, die nachträglich auf das System installiert werden, zählen dazu, z. B. Word oder PowerPoint.

Subsysteme

Schließlich existieren Subsysteme, die Funktionen zur Verwaltung eines bestimmten Programm- bzw. Prozesstyps zur Verfügung stellen. Beispiele sind das OS/2-, POSIX- und MS-DOS-Subsystem, die die Ausführung von Programmen dieser Betriebssysteme gestatten. Jedes Subsystem stellt eine eigene Programmierschnittstelle (API) zur Verfügung. Win32 ist mit seiner API aus dieser Sicht selbst ein Subsystem, auch innerhalb der eigenen 32-Bit-Windows-Umgebung.

3 Anwendungen und Bibliotheken

Portable-Executable-Format

Alle Dateitypen, die auf einem Windows-System zur Ausführung kommen, haben eine Eigenschaft gemein. Sie alle verwenden das PE(*Portable Executable*)-Format. Anwendungen haben die Endungen *.exe*, *.com* oder *.scr* (Screensaver). Dynamische Bibliotheken haben die Endungen *.dll* und *.ocx*. Kernel-Programme oder Treiber enden mit *.sys*, *.vxd*, *.exe* oder *.dll*. Darüber hinaus gibt es noch ausführbare Skriptdateien (z. B. mit den Endungen *.bat* oder *.cmd*), die uns hier allerdings nicht weiter interessieren.

Die Windows-Anwendungen lassen sich in drei Gruppen unterteilen:

1. Windows-Systemprogramme (z. B. *winlogon.exe* für die Benutzeranmeldung und der Programmmanager *explorer.exe*)
2. Windows-Hilfsprogramme (z. B. der Taschenrechner *calc.exe* und das Textbearbeitungsprogramm *notepad.exe*)
3. Programme, die nicht zum Betriebssystem selbst gehören.

Alle Windows-Anwendungen bestehen aus drei grundlegenden Teilen: Code, Daten und optionalen Debug-Informationen.

Libraries

Anwendungen können vorgefertigte Programmbibliotheken (*Libraries*) verwenden. Auch diese Bibliotheken bestehen aus Code und Daten, sind aber nicht alleine ausführbar. Windows stellt viele Standardbibliotheken zur Verfügung, diese bilden die API (*Application Programming Interface*). Beispiele hierfür sind die Bibliotheken *user32.dll* und *kernel32.dll*. Des Weiteren gibt es Compiler-Bibliotheken, aus denen sich der Programmierer bei der Erstellung eines Hochsprachenprogramms bedienen kann. Darüber hinaus gibt es unzählige Bibliotheken von irgendwelchen Herstellern für irgendwelche benutzerspezifischen Spezialanwendungen.

Import und Export

Bibliotheken stellen für Anwendungen Funktionen und Daten zur Verfügung. Man sagt auch, Bibliotheken **exportieren** Funktionen und Daten. Diese können von Anwendungen ihrerseits **importiert** werden. Bibliotheken exportieren Funktionen und Daten via Namen (z. B. *CreateFileA*) oder Ordinalzahl (eine Nummer, z. B. `0x30`), die von den Anwendungen ihrerseits via Namen oder Ordinalzahl importiert werden.

Der Import von Bibliotheken bzw. einzelner Bestandteile davon kann grundsätzlich statisch oder dynamisch erfolgen (s. Abb. 2). Bei der statischen Bindung werden die Bibliotheken der Anwendung einverleibt. Dies geschieht bereits bei der Kompilierung der Anwendung. Bei der dynamischen Bindung werden die Bibliotheken erst zur Laufzeit der Anwendung in den Speicher geladen. Beim Kompilieren werden lediglich Verweise auf die für die Ausführung notwendigen Bibliotheken sowie darin enthaltenen Funktionen und Daten in der Anwendung hinterlegt. Beim Start der Anwendung wird geprüft, ob die in den Verweisen stehenden Bibliotheken schon geladen wurden. Falls nicht, lädt der Loader fehlende Bibliotheken in den Speicher.

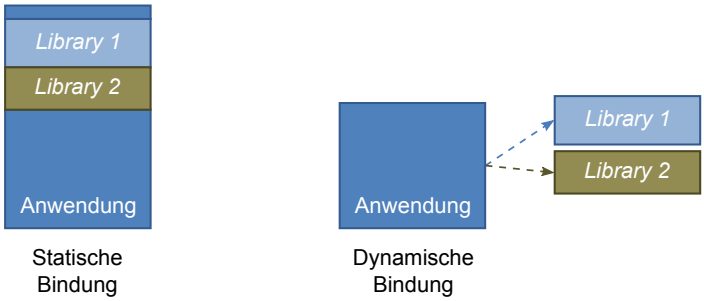


Abb. 2: Statische und dynamische Bindung

Dynamic Link Library

Windows-Bibliotheken sind für die dynamische Bindung vorgesehen. Das erklärt auch die Dateiendung *.dll* für *Dynamic Link Library* (DLL). Bibliotheken werden entweder zusammen mit der Anwendung auf einem Rechner installiert, oder ihr Vorhandensein wird auf dem Zielrechner der Anwendung vorausgesetzt.

Beide Bindungsarten haben offensichtliche Vor- und Nachteile. Die Vorteile einer statischen Bindung sind das schnelle Laden und Starten einer Anwendung und

die Unabhängigkeit vom Vorhandensein einzelner Bibliotheken. Die Nachteile bestehen neben der logischerweise ausgeschlossenen Mehrfachverwendung von Bibliotheken zum einen in einer größeren, aufgeblähten Anwendungsdatei und zum anderen in der Notwendigkeit, die Anwendungen bei einem Update der Bibliothek neu zu kompilieren.

K

Kontrollaufgabe 1

Formulieren Sie die Vor- und Nachteile der dynamischen Bindung!

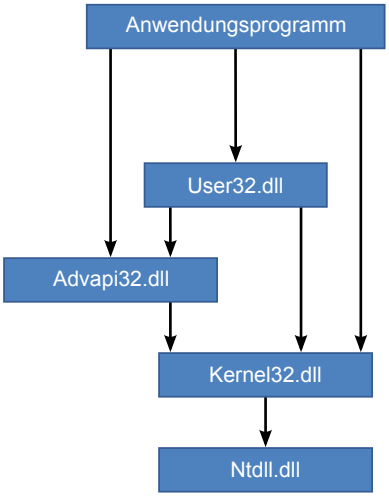
4 Windows API

Das Windows *Application Programming Interface* (API) bildet die Schnittstelle zwischen Betriebssystem und Anwenderprogrammen. Die API besteht aus vielen DLLs, die sich im *system32*-Ordner eines Windows-Systems befinden. Implementiert sind diese DLLs in C oder Assembler. Die Ordnung der DLLs ist nicht flach, sondern hierarchisch. Viele DLLs benutzen ihrerseits andere DLLs, wie dies in Abb. 3 dargestellt ist.

Einige wichtige DLLs sind die folgenden:

- Native API: *ntdll.dll*
- Kernelfunktionen: *kernel32.dll*
- Basisfunktionen: *advapi32.dll*
- Grafik/Fenster: *user32.dll*, *gdi32.dll*
- TCP/IP: *Winsock ws2_32.dll*

Abb. 3: DLL-Hierarchie



Systemaufruf

Die native API, die aus der Bibliothek *ntdll.dll* besteht, befindet sich am äußersten Ende der DLL-Hierarchie und ist nicht dazu vorgesehen, von Anwenderprogrammen direkt angesprochen zu werden. Einem Aufruf einer Funktion dieser Bibliothek folgt ein Systemaufruf, also die Kommunikation mit einem Kernel-Prozess. Alle übergeordneten Bibliotheksaufrufe, die letztendlich eine Kommunikation mit dem Kernel benötigen, landen bei der *ntdll.dll*. Die Bibliothek *ntdll.dll* kommuniziert als einzige Bibliothek mit dem Kernel. Ein Interrupt, der den Wechsel

vom Benutzermodus in den Kernel-Modus einleitet, wird in der *ntdll.dll* ausgelöst. Folgerichtig wird nach dem Bearbeiten der Anfrage im Kernel-Modus ein Ergebnis erst an *ntdll.dll* übergeben, bevor es an andere DLLs weitergereicht wird. Offiziell ist *ntdll.dll* undokumentiert, weil sie nicht zur Benutzung durch direkte Aufrufe vorgesehen ist. Microsoft ändert auch gelegentlich die Schnittstellen, um die Aktivitäten von Reverse Engineering an dieser DLL zu untergraben.

Eine Ebene oberhalb befindet sich *kernel32.dll*, die Funktionsaufrufe an *ntdll.dll* weiterleitet. Basisfunktionen wie die Registry-Verwaltung bietet *advapi32.dll* an. Sie sendet ihrerseits Informationen an die Bibliothek *kernel32.dll* weiter. Bibliotheken zur Steuerung von Grafiken und Fenstern sind die *user32.dll* und die *gdi32.dll*. Sie leiten den Informationsfluss an *advapi32.dll* oder *kernel32.dll* weiter.

Ein Aufruf einer Funktion einer DLL hat allgemein die Form *DLL!Funktion(A/W)*, bspw. *Kernel32!CreateFileA*. Dabei steht *Kernel32* für die aufzurufende DLL *kernel32.dll*. Ein Ausrufezeichen oder ein Punkt trennen den Namen der DLL von der Funktion, die innerhalb dieser DLL aufgerufen werden soll. *CreateFile* ist der Name einer Funktion zum Erstellen von Dateien. Viele API-Funktionen haben hinter dem eigentlichen Funktionsnamen noch einen Buchstaben (*A* oder *W*) angehängt. Diese Unterscheidung wird dann gemacht, wenn die Übergabeparameter einen oder mehrere Strings enthalten. Hierbei steht *A* für ASCII-Format und *W* für Unicode-Format.

B

Beispiel 1

Die Deklaration der API-Funktion *CreateFile* sieht in C-Notation wie folgt aus:

```
HANDLE CreateFile(
  LPCTSTR lpFileName,           // pointer to name of the file
  DWORD dwDesiredAccess,        // access (read-write) mode
  DWORD dwShareMode,            // share mode
  LPSECURITY_ATTRIBUTES         // pointer to
  lpSecurityAttributes,         // security attributes
  DWORD dwCreationDistribution, // how to create
  DWORD dwFlagsAndAttributes,   // file attributes
  HANDLE hTemplateFile          // handle to file with
                                // attributes to copy
  );
```

Auf Details soll hier nicht eingegangen werden. Der Funktion werden Namen und diverse Zugriffsvorgaben für die zu erstellende Datei übergeben. Rückgabewert ist ein sogenannter *Handle*; dies ist ein eindeutiger Referenzwert zu einer vom Betriebssystem verwalteten Systemressource, z. B. einem Bildschirmobjekt oder einer Datei auf einer Festplatte.

Aufruf von API-Funktionen

API-Funktionen werden von einer Hochsprache wie C vielfach indirekt über die Funktionen der Standardbibliothek aufgerufen, können aber auch direkt von der Hochsprache aus oder von einem Assemblerprogramm benutzt werden. Als Beispiel ist nachfolgend ein Windows-Programm in C aufgelistet, das die Funktion *MessageBox* der Windows-DLL *user32.dll* benutzt, die ein in Windows übliches Meldungsfeld erzeugt.

Q

Quelltext 1

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <windows.h>
4 const char szText[] = "Haben Sie d. Programm verstanden?";
5
6 int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE
      hPrevInstance, PSTR szCmdLine, int iCmdShow) {
7     int iAntwort = MessageBox(NULL, szText, "Eine kleine Box",
          MB_ICONINFORMATION | MB_OKCANCEL | MB_DEFBUTTON1);
8     if (IDOK == iAntwort)
9       MessageBox(NULL, "Das ist prima!","",
          MB_ICONINFORMATION | MB_OK | MB_DEFBUTTON1);
10    else if (IDCANCEL == iAntwort)
11      MessageBox(NULL, "Schade, dann schauen Sie es sich
          bitte nochmal an!","",MB_ICONINFORMATION | MB_OK |
          MB_DEFBUTTON1);
12    return 0;
13 }
```

Die Deklarationen der Windows API werden durch Einfügen der Datei `windows.h` eingebunden. Windows-Programme in C verwenden statt der üblichen Hauptfunktion `main` die Funktion `WinMain` mit diversen Standardparametern. Die Funktion *MessageBox* erhält einige Textparameter und Parameter für die zu generierenden interaktiven Bedienknöpfe.

Wer sich eingehend mit der Programmierung der Windows API, insbesondere zur Erzeugung grafischer Oberflächen, beschäftigen will, kann sich bspw. bei diesen (einführenden) Webseiten bedienen. Details würden den Rahmen dieses Studienbriefs sprengen:

- http://www.gis-management.de/pdf/winAPI.pdf
- http://www.zetcode.com/gui/winapi/main/
- http://www.pronix.de/pronix-1037.html
- http://www.win-api.de/tutorials.php

5 Systemaufrufe

Ob ein Programm im User- oder im Kernelmode arbeitet, wird durch den *Current Privilege Level* (CPL) angezeigt.[1] Hardware-Zugriffe und privilegierte CPU-Befehle, die direkt auf die Hardware wirken, sind nur im Kernelmode möglich, ebenso wie Zugriffe auf den Kernelspace. Bei Anwendungen sind daher unter Umständen oft Wechsel in den Kernelmode notwendig.

In Abb. 4 ist ein Aufruf der Funktion *CreateFileA* abgebildet, der durch mehrere DLLs schließlich den Sprung in den Kernelmode schafft, um dort die nötige Funktion auszuführen. Der Wechsel in den Kernelmode geschieht über einen Systemaufruf (Syscall). In der Regel wird eine Anwendung den Syscall durch den Umweg über die Windows API ausführen. Einen Syscall direkt aus einer Anwendung heraus aufzurufen ist durchaus auch möglich, allerdings wohl ausschließlich

[1] Bei der Intel IA-32-Architektur ist dieser für Windows-Programme in Bit 0 und 1 des Segmentregisters `cs` hinterlegt.

auf der Assemblerebene. Darüber hinaus sind die Schnittstellen zum Kernel nicht offiziell dokumentiert, was einem „normalen“ Programmierer den Gebrauch stark erschwert.

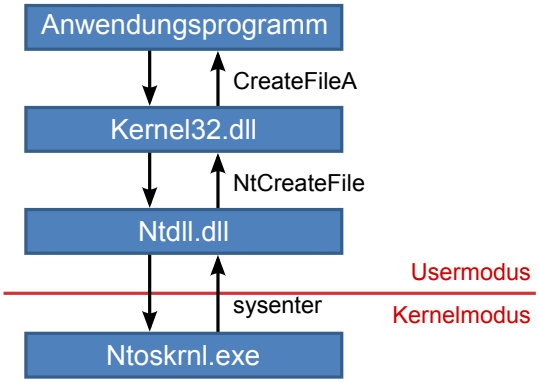


Abb. 4: Übergang in den Kernelmode

Übergang in den Kernelmode

Der Übergang in den Kernelmode erfolgt bei älteren Systemen durch Auslösen des Interrupt 3 mit dem `int`-Befehl. Ab dem Pentium II gibt es den Befehl `sysenter`, bei AMD Prozessoren heißt er `syscall`. Durch `systenter` werden die Register `cs, ss, eip` und `esp` aus CPU-modellspezifischen Systemregistern geladen, was den Befehl unter die Kontrolle des Kernelmode stellt, obwohl er im Usermode aufgerufen werden kann. Der Rücksprung aus dem Kernelmode in den Usermode erfolgt (bei Aufruf über `sysenter`) mit dem Befehl `sysexit`. Die vier genannten Register werden dabei wieder aus CPU-modellspezifischen Registern geladen.

6 Das PE-Format

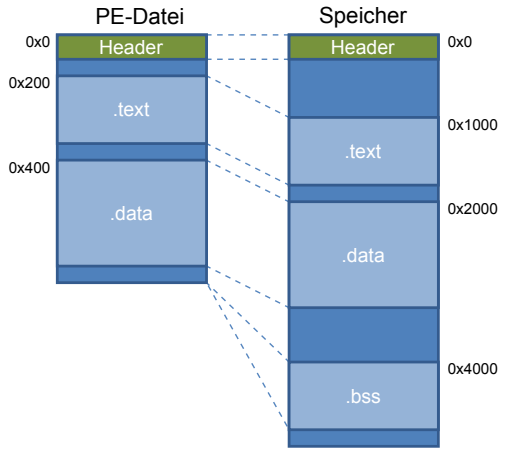
Das PE-Format (*Portable Executable*) ist das Binärformat für ausführbare Dateien in Windows-Systemen und basiert auf dem *Common Object File Format* (COFF)[2]. *Portable* suggeriert eine Austauschbarkeit der Dateien zwischen unterschiedlichen Betriebssystemen. In Wirklichkeit sind PE-Dateien allenfalls von anderen Betriebssystemen ladbar, aber noch lange nicht ausführbar. Dies gilt auch innerhalb der Windows-Familie.

Das PE-Format spezifiziert die Struktur unter Windows ausführbarer Dateien, also von Anwendungen, Bibliotheken und Gerätetreibern. Insbesondere legt die Spezifikation fest, wo und wie in einer PE-Datei die Informationen über benötigte Bibliotheken stehen, und wo die Code- und Datenblöcke zu finden sind. Der Windows Loader wird durch das PE-Format direkt bei der Erstellung eines Prozesses unterstützt.

Eine PE-Datei wird nahezu 1:1 in den Speicher übertragen. Allerdings ist eine PE-Datei kompakter als das Speicherabbild. Dies ist in Abb. 5 dargestellt. Die einzelnen Blöcke der PE-Datei (Header und Sections[3]) beginnen in der Datei häufig bei Mehrfachen von `0x200` Byte, also bei `0x0, 0x200, 0x400` usw. Dies wird als Datei-Alignment bezeichnet. Diese Blöcke werden in den Speicher ab den Mehrfachen von `0x1000` Byte geladen, also ab `0x0, 0x1000, 0x2000` usw. Dies ist das Speicher-Alignment der Blöcke. Zwangsläufig entstehen so „Lücken“ im Speicher, die größer sind als die Lücken in der Datei.

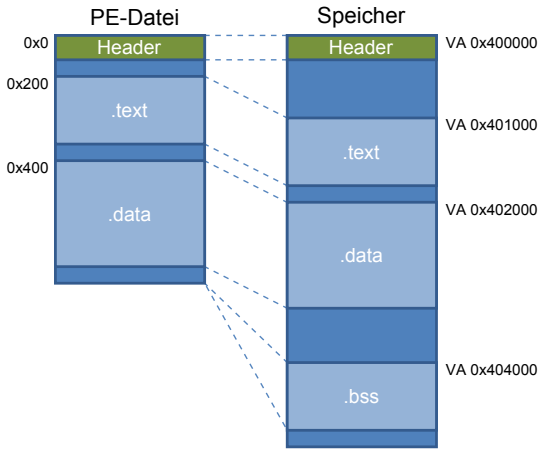
[2] Nähere Informationen zum COFF-Format sind bspw. unter en.wikipedia.org/wiki/COFF zu finden.
[3] Dies sind insbesondere Code- und Datenblöcke, die meist als .text- und .data Sections bezeichnet werden.

Abb. 5: Alignment in Datei und Speicher



Eine PE-Datei wird in den linearen oder virtuellen Adressraum geladen und belegt dort i. Allg. nicht den Platz ab Adresse `0x0`. Abbildung 6 zeigt dies nochmals anhand des vorher gezeigten Beispiels. Es wird angenommen, dass das Speicherabbild der PE-Datei ab der virtuellen Adresse (VA) `0x400000` im Speicher platziert wird. Die .text Section beginnt damit bspw. an der virtuellen Adresse `0x401000`.

Abb. 6: Offset, virtuelle Adresse und relative virtuelle Adresse



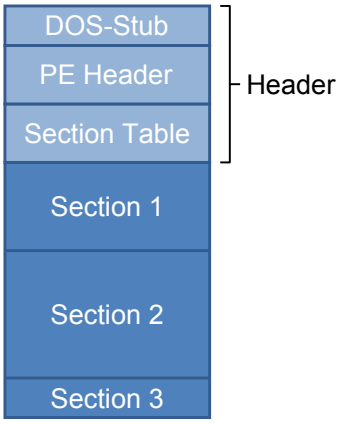
Der Offset zur relativen Startadresse des Speicherabbilds wird als relative virtuelle Adresse (RVA) bezeichnet. Die .text Section beginnt also ab der RVA `0x1000` zur VA `0x400000` des Speicherabbilds der PE-Datei.

Alle Dateien im PE-Format (s. Abb. 7) setzen sich aus zwei Bestandteilen zusammen: einem **Header** und ihm nachfolgende **Sections**. Der Header besteht aus drei Teilen:

1. DOS Header[4]
2. PE Header
3. Section Table

[4] Der DOS Header wird oft vereinfacht auch DOS Stub genannt, auch wenn der eigentliche DOS Stub nur ein Teil des DOS Header ist.

Abb. 7: PE-Format



Der DOS Header gewährleistet eine Art Aufwärtskompatibilität. Es gab diesen schon in der DOS-Zeit, in der er MZ-Header genannt wurde. Der DOS Header erlaubt es theoretisch, eine für Windows kompilierte Datei unter einer alten DOS-Version auszuführen. Allerdings wird dies in der Praxis kaum genutzt, und die meisten Windows-Programme brechen mit der Meldung „This program cannot be run in DOS Mode“ (oder ähnlich) ab, wenn sie unter einem inkompatiblen Betriebssystem ausgeführt werden.

Auf den DOS Header folgt der PE Header, der für eine Programmanalyse besonders wichtige Informationen enthält und daher nachfolgend detailliert vorgestellt wird.

Der letzte Bereich des Header enthält die Section Table. Ihre Größe ist variabel, weil die Anzahl der Sections beliebig sein kann. In der Section Table sind Verweise auf die dahinter liegenden Sections gespeichert.

Die Sections nach dem Header enthalten Code, Daten, Debug-Information, Ressourcen (z. B. Icons) oder Verwaltungsinformationen.

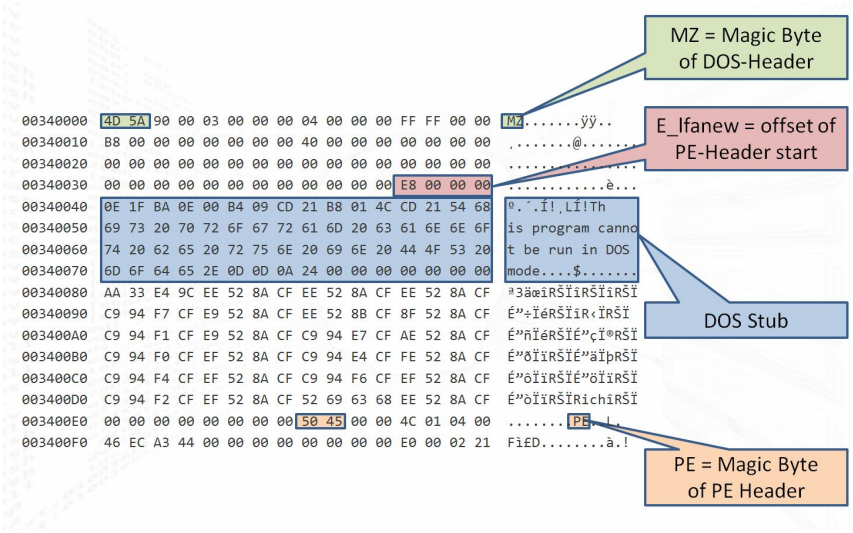
Die einzelnen Teile einer PE-Datei werden in den folgenden Abschnitten genauer beschrieben.

6.1 DOS Header

Abb. 8 zeigt den Anfang einer PE-Datei. Die Datei beginnt immer mit den *Magic Bytes* **MZ**. MZ sind die Initialen von Mark Zbikowski, dem Entwickler des DOS-Programmformats.

Der Anfang des DOS Header besteht aus folgenden Komponenten:

Abb. 8: DOS Header




```
+0x000 e_magic          : Uint2B ; MZ
+0x002 e_cblp           : Uint2B
+0x004 e_cp             : Uint2B
+0x006 e_crlc           : Uint2B
+0x008 e_cparhdr        : Uint2B
+0x00a e_minalloc       : Uint2B
+0x00c e_maxalloc       : Uint2B
+0x00e e_ss             : Uint2B
+0x010 e_sp             : Uint2B
+0x012 e_csum           : Uint2B
+0x014 e_ip             : Uint2B
+0x016 e_cs             : Uint2B
+0x018 e_lfarlc         : Uint2B
+0x01a e_ovno           : Uint2B
+0x01c e_res            : [4] Uint2B
+0x024 e_oemid          : Uint2B
+0x026 e_oeminfo        : Uint2B
+0x028 e_res2           : [10] Uint2B
+0x03c e_lfanew         : Int4B ; offset of PE-Header start
```

Dies sind insbesondere Verweise auf Codesegment, Stack-Segment, Programmeinstiegspunkt usw. eines DOS-Programms. Auf Details soll hier allerdings nicht eingegangen werden. Der letzte Eintrag `e_lfanew` verweist auf den Anfang des PE Header. Danach folgt der eigentliche DOS Stub.

6.2 PE Header

Der Aufbau des PE Header ist in Abb. 9 dargestellt und wird im Folgenden näher aufgeschlüsselt.

Die Magic Byte **PE** kennzeichnen den Anfang eines PE Header. Es folgen zwei Verweise auf den *File Header* und den *Optional Header*. Der PE Header ist aus diesen beiden *Sub Headern* zusammengesetzt, wobei der Optional Header trotz seines Namens keineswegs optional ist:

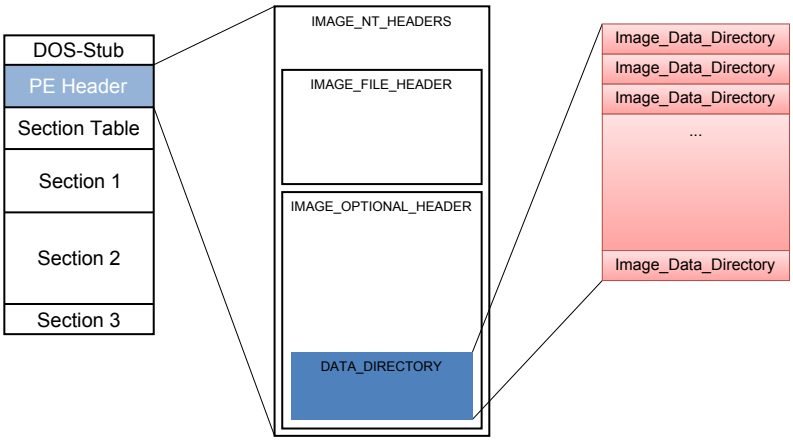


Abb. 9: Aufbau PE Header [Holz, 2012]

```
PE-Header = IMAGE_NT_HEADERS
+0x000 Signature        : Uint4B ; PE
+0x004 FileHeader       : IMAGE_FILE_HEADER
+0x018 OptionalHeader   : IMAGE_OPTIONAL_HEADER
```

File Header

Der File Header enthält unter anderem Informationen über die Anzahl der Sections (`NumberOfSections`), den erforderlichen CPU-Typ (`Machine`), die Größe des Optional Header (`SizeOfOptionalHeader`) sowie das Feld `Characteristics`, das unter anderem angibt, ob es sich bei der PE-Datei um eine Anwendung oder eine Bibliothek (DLL) handelt:

```
File-Header = IMAGE_FILE_HEADER
+0x000 Machine               : Uint2B
+0x002 NumberOfSections      : Uint2B
+0x004 TimeDateStamp         : Uint4B
+0x008 PointerToSymbolTable  : Uint4B
+0x00c NumberOfSymbols       : Uint4B
+0x010 SizeOfOptionalHeader  : Uint2B
+0x012 Characteristics       : Uint2B ; u.a. DLL Flag
```

Optional Header

Der Optional Header enthält Informationen zu Umfang von Code und Daten (`SizeOfCode` usw.), den Programmeinstiegspunkt (`AdressOfEntryPoint`) und die benötigten Daten, um die Expansion einer PE-Datei in den Speicher vorzunehmen, wie es am Anfang von Abs. 6 gezeigt wurde (`SectionAlignment` und `FileAlignment`). `ImageBase` gibt an, ab welcher virtuellen Startdresse die PE-Datei normalerweise in den Speicher abgebildet werden soll, sofern diese nicht belegt ist:[5]

```
Optional-Header =  IMAGE_OPTIONAL_HEADER
+0x000 Magic                   : Uint2B
+0x002 MajorLinkerVersion      : Uchar
+0x003 MinorLinkerVersion      : Uchar
+0x004 SizeOfCode              : Uint4B
+0x008 SizeOfInitializedData   : Uint4B
```

[5] Wünschen sich z. B. 2 DLLs dieselbe Startadresse, so ist eine Relocation erforderlich.

```
+0x00c SizeOfUninitializedData : Uint4B
+0x010 AddressOfEntryPoint     : Uint4B
+0x014 BaseOfCode              : Uint4B
+0x018 BaseOfData              : Uint4B
+0x01c ImageBase               : Uint4B
+0x020 SectionAlignment        : Uint4B
+0x024 FileAlignment           : Uint4B
-
+0x050 SizeOfHeapReserve       : Uint4B
+0x054 SizeOfHeapCommit        : Uint4B
+0x058 LoaderFlags             : Uint4B
+0x05c NumberOfRvaAndSizes     : Uint4B
+0x060 DataDirectory           : [16] IMAGE_DATA_DIRECTORY
```

Data Directory

Das *Data Directory*, ein Array mit 16 Einträgen, ist ebenfalls im Optional Header zu finden. Es verweist auf Tabellen, die für das Laden und Ausführen des in der PE-Datei enthaltenen Programms unabdingbar sind.

Jeder Eintrag im Data Directory hat die Form:

```
IMAGE_DATA_DIRECTORY
+0x000 VirtualAddress   : Uint4B
+0x004 Size             : Uint4B
```

Ein solcher Eintrag enthält Adresse und Größe der zugeordneten Tabelle. Die einzelnen Tabellen sind die folgenden:

```
Data Directory Eintraege
IMAGE_DIRECTORY_ENTRY_EXPORT       = 0;   // Export Directory
IMAGE_DIRECTORY_ENTRY_IMPORT       = 1;   // Import Directory
IMAGE_DIRECTORY_ENTRY_RESOURCE     = 2;   // Ressource Directory
IMAGE_DIRECTORY_ENTRY_EXCEPTION    = 3;   // Exception Directory
IMAGE_DIRECTORY_ENTRY_SECURITY     = 4;   // Security Directory
IMAGE_DIRECTORY_ENTRY_BASERELOC    = 5;   // Base Relocation Table
IMAGE_DIRECTORY_ENTRY_DEBUG        = 6;   // Debug Directory
IMAGE_DIRECTORY_ENTRY_COPYRIGHT    = 7;   // Description String
IMAGE_DIRECTORY_ENTRY_GLOBALPTR    = 8;   // Machine Value (MIPS GP)
IMAGE_DIRECTORY_ENTRY_TLS          = 9;   // TLS Directory
IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG  = 10;  // Load Configuration Dir.
IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT = 11;  // Bound Import Directory
IMAGE_DIRECTORY_ENTRY_IAT          = 12;  // Import Address Table
IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT       = 13;  // Delayed Imports
IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTORS    = 14;  //
```

Aus der Sicht der Programmanalyse sind insbesondere das *Export Directory*, das *Import Directory* und die *Import Address Table* wichtig. Import und Export Directory geben die zur Programmausführung zu importierenden Funktionen bzw. die vom Programm selbst exportierten Funktionen an. Die Import Adress Table enthält die Adressen der importierten Funktionen. Export Directory und Import Directory werden in den Abschnitten 6.4 und 6.5 näher untersucht.

6.3 Section Table

In den Sections sind die eigentlichen Inhalte einer PE-Datei enthalten. Die Section Table enthält die Metadaten der Sections, also alle Informationen über die auf den Header folgenden, aneinandergereihten Sections. Pro Section enthält die Tabelle einen Eintrag. Ein Eintrag in der Section Table ist wie folgt aufgebaut:

```
IMAGE_SECTION_HEADER
+0x000 Name                    : [8] Uchar
+0x008 VirtualSize             : Uint4B ; Groesse im Speicher
+0x00c VirtualAddress          : Uint4B ; Adresse im Speicher
+0x010 SizeOfRawData           : Uint4B
+0x014 PointerToRawData        : Uint4B ; Offset in Datei
+0x018 PointerToRelocations    : Uint4B
+0x01c PointerToLinenumbers    : Uint4B
+0x020 NumberOfRelocations     : Uint2B
+0x022 NumberOfLinenumbers     : Uint2B
+0x024 Characteristics         : Uint4B ; Code? Daten?
```

Neben dem Namen der Section sind hier seine Größe und Lage sowohl in der Datei als auch im Speicher und die Art der Section (`Characteristics`) angegeben. Der Name der Section (`Name`) ist frei wählbar, wobei .text für Code und .data für Daten sehr häufig vorkommen. Der Name .bss deutet auf uninitialisierte Daten hin; das bedeutet in den meisten Fällen, dass diese Section zunächst nur Nullen enthält. Der Name .rdata wird für Sections mit Read-Only-Daten verwendet. Solche Sections können besonders interessant sein, weil sie z. B. verschlüsselte Passwörter enthalten könnten. Des Weiteren ist noch der Name .rsrc für Sections mit Programmressourcen (z. B. Windows Icons) gebräuchlich, und .idata deutet auf die Import Address Table hin.

6.4 Export Directory

Bibliotheken exportieren Funktionen und/oder Daten, die zusammenfassend als Symbole bezeichnet werden. Die exportierten Symbole werden über das Export Directory veröffentlicht, das folgenden strukturellen Aufbau hat:

```
typedef struct _IMAGE_EXPORT_DIRECTORY
{
    DWORD   Characteristics;
    DWORD   TimeDateStamp;
    WORD    MajorVersion;
    WORD    MinorVersion;
    DWORD   Name;                   // interner Name der DLL
    DWORD   Base;                   // gewuenschte VA
    DWORD   NumberOfFunctions;      // Anzahl Export
    DWORD   NumberOfNames;          // Anzahl Exporte mit Namen
    DWORD   AddressOfFunctions;     // EAT  Export Address Table
    DWORD   AddressOfNames;         // ENT  Export Name Table
    DWORD   AddressOfNameOrdinals;  // EOT  Export Ordinal Table
}
IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

Neben mehreren Statusinformationen sind im Export Directory der Name der exportierenden Bibliothek, die (gewünschte) virtuelle Adresse im Speicher, die

Anzahl der Exporte, die Anzahl der Exporte mit Namen und Verweise auf die Tabellen mit den eigentlichen Exporten enthalten:

Export Address Table

Die *Export Address Table* (EAT) enthält einen Eintrag für jedes exportierbare Symbol. Dieser Eintrag ist der Offset der Funktion zur Basisadresse der Bibliothek im Speicher, also die relative virtuelle Adresse (RVA) der Funktion. Die EAT hat `NumberOfFunctions` Einträge.

Export Name Table

Die *Export Name Table* (ENT) hat einen Eintrag für jeden exportierten Symbolnamen. Die ENT hat `NumberOfNames` Einträge.

Export Ordinal Table

Die *Export Ordinal Table* (EOT) korrespondiert mit der ENT, hat also auch `NumberOfNames` Einträge, die den Symbolnamen eindeutige (16-Bit)-Ordinalzahlen zuordnen, nämlich den Index des zugehörigen Symbols in der EAT.

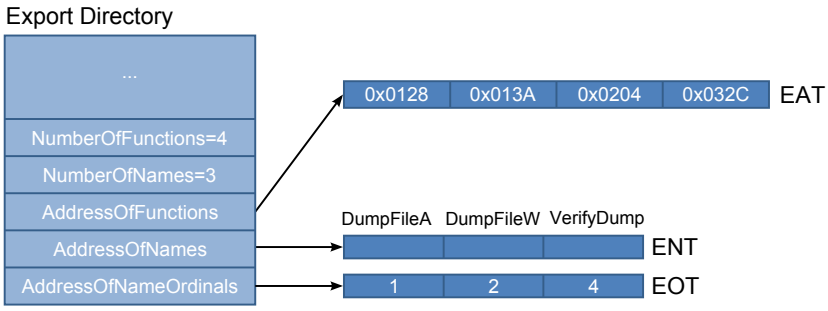
Die Länge von EAT und ENT bzw. EOT kann unterschiedlich sein, weil einerseits nicht jedes exportierte Symbol in ENT referenziert sein muss und andererseits auch mehrere Namen (*Aliase*) für ein und dasselbe Symbol existieren können.

B

Beispiel 2

Abb. 10 zeigt den relevanten Teil eines Export Directory, das vier Symbole exportiert. EOT referenziert drei dieser Symbole. Um die Startadresse des Symbols *VerifyDump* zu erhalten, muss zunächst in der ENT nach dem Namen gesucht werden. Es ist der dritte Eintrag in der ENT. Der entsprechende Eintrag in der EOT ist die 4, also ist das vierte Symbol gemeint, das in der EAT den Eintrag `0x032C` hat. Durch Addition von `0x032C` und DLL-Ladeadresse (Basisadresse) erhält man die gewünschte Startadresse.

Abb. 10: Export Directory mit EAT, ENT und EOT



B

Beispiel 3

Abb. 11 zeigt einen Ausschnitt der EAT der Windows-Bibliothek *kernel32.dll* mit den korrespondierenden Einträgen in der ENT und der EOT. Zu dem Symbol *CreateFileA* sind bspw. das Ordinal 80 und die relative virtuelle Adresse `0x1A28` eingetragen.

Prinzipiell ist es auch möglich, ohne den Umweg über ENT und EOT direkt über EAT auf die exportierten Symbole zuzugreifen. Das geht allerdings nur solange gut, wie sich dessen Aufbau, also die Reihenfolge der exportierten Symbole, nicht ändert.

E	Ordinal	Function ^	Entry Point
	60 (0x003C)	ContinueDebugEvent	0x0005B53D
	61 (0x003D)	ConvertDefaultLocale	0x000383FF
	62 (0x003E)	ConvertFiberToThread	0x0002FEDF
	63 (0x003F)	ConvertThreadToFiber	0x0002FF1E
	64 (0x0040)	CopyFileA	0x000286EE
	65 (0x0041)	CopyFileExA	0x0005F39C
	66 (0x0042)	CopyFileExW	0x00027B32
	67 (0x0043)	CopyFileW	0x0002F87B
	68 (0x0044)	CopyLZFile	0x0005989A
	69 (0x0045)	CreateActCtxA	0x0006C8E5
	70 (0x0046)	CreateActCtxW	0x000154FC
	71 (0x0047)	CreateConsoleScreenBuffer	0x000741A8
	72 (0x0048)	CreateDirectoryA	0x000217AC
	73 (0x0049)	CreateDirectoryExA	0x0005C213
	74 (0x004A)	CreateDirectoryExW	0x0005B5CA
	75 (0x004B)	CreateDirectoryW	0x00032402
	76 (0x004C)	CreateEventA	0x000308B5
	77 (0x004D)	CreateEventW	0x0000A749
	78 (0x004E)	CreateFiber	0x0002FFB7
	79 (0x004F)	CreateFiberEx	0x0002FFD7
	80 (0x0050)	CreateFileA	0x00001A28
	81 (0x0051)	CreateFileMappingA	0x0000950A
	82 (0x0052)	CreateFileMappingW	0x0000943C
	83 (0x0053)	CreateFileW	0x00010800
	84 (0x0054)	CreateHardLinkA	0x0006C769
	85 (0x0055)	CreateHardLinkW	0x0006C5AC
	86 (0x0056)	CreateIoCompletionPort	0x0003138D
	87 (0x0057)	CreateJobObjectA	0x0006C4CC
	88 (0x0058)	CreateJobObjectW	0x0002CB13

Abb. 11: EAT, ENT und EOT von *kernel32.dll*

E

Exkurs 1: **Manuelle Exportsuche**

Wie sucht man aber nach einem bestimmten Export?

Normalerweise erledigt der Windows Loader die Arbeit, indem er beim Laden einer PE-Datei die benötigten Importe erkennt, die DLLs in den Speicher lädt und den Offset der korrespondierenden Exporte ermittelt.

Man kann allerdings den Suchprozess auch „manuell“ veranlassen. Ein Grund, einen Export manuell zu suchen, kann die Verschleierung der Nutzung einer DLL oder eines Symbols einer DLL sein. Beim Laden der PE-Datei bzw. bei der statischen Analyse sieht es dann so aus, als ob keine (kritischen) Exports benötigt würden.

Dies kann zum Beispiel von Malware genutzt werden, die eine aktive Internetverbindung aufbauen will, dem Analysten jedoch suggerieren will, dass sie keine Verbindung nach außen aufbauen möchte. In einem solchem Fall wäre es denkbar, die zum Aufbau einer Internetverbindung bspw. erforderlichen und verräterischen Symbole der *ws2_32.dll* erst während des Programmlaufs manuell aufzusuchen.

Manuelle Exportsuche ist mit der API-Funktion *GetProcAddress* möglich. Übergabeparameter sind die zu durchsuchende Bibliothek und der Symbolname bzw. das Symbolordinal. *GetProcAddress* liefert die virtuelle Adresse des gesuchten Symbols.

6.5 Import Directory

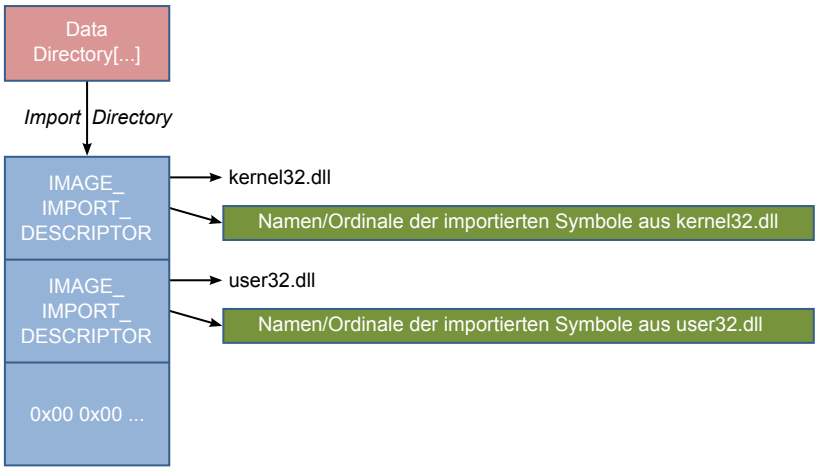
Neben dem Export Directory ist das Import Directory die zweite wesentliche Datenstruktur, auf die das Data Directory verweist. Das Import Directory enthält eine Liste der verwendeten Bibliotheken und der zu importierenden Symbole.

Einträge in dieser Liste sind letztendlich entweder Namen oder Zahlen des jeweils

korrespondierenden Exports. Abb 12 zeigt den Aufbau des Import Directory einer PE-Datei.

Es ist durchaus üblich, dass importierte Bibliotheken ihrerseits weitere Bibliotheken aufrufen. Beim Laden einer PE-Datei analysiert daher der Windows Loader *rekursiv* das Import Directory, lädt alle dabei referenzierten Bibliotheken und ermittelt die Adressen aller referenzierten Symbole. Diese Adressen landen in der *Import Address Table*(IAT).

Abb. 12: Import Directory



Das Import Directory ist als eine Kette von *Import Deskriptoren* aufgebaut. Import Deskriptoren haben den folgenden strukturellen Aufbau:

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR
{
  union
  {
      DWORD   Characteristics;      // alt; 0 fuer Listenende
      DWORD   OriginalFirstThunk;   // RVA zur INT/ILT
  };
  DWORD   TimeDateStamp;            // fuer bound imports
  DWORD   ForwarderChain;           //
  DWORD   Name;                     // RVA zum Namen der DLL
  DWORD   FirstThunk;               // RVA zur IAT
}
IMAGE_IMPORT_DESCRIPTOR, *PIMAGE_IMPORT_DESCRIPTOR;
```

OriginalFirstThunk und FirstThunk

Ein Null-Eintrag zeigt das Ende der Liste an. Ansonsten enthält der Import Deskriptor einen Verweis auf den Namen der importierten Bibliothek sowie Verweise auf *OriginalFirstThunk* und *FirstThunk*. OriginalFirstThunk und FirstThunk sind Tabellen, deren Einträge zunächst auf die *Import_Name/Import_Locator Table* (INT/ILT) verweisen. In der INT/ILT sind die Namen und Ordinale der zu importierenden Symbole enthalten. INT/ILT besteht aus Elementen des folgenden Typs:

```
typedef struct _IMAGE_IMPORT_BY_NAME
{
 WORD Hint;     // Ordinal
 BYTE Name1;    // null-terminierter ASCII-String
}
IMAGE_IMPORT_BY_NAME, *PIMAGE_IMPORT_BY_NAME;
```

Beim Laden der PE-Datei ermittelt der Windows Loader die Adressen der Importe und legt diese statt der Verweise auf INT/ILT in FirstThunk ab. Die Adressen erhält der Loader aus dem Export Directory der importierten DLL. Diese Adressen werden bei der Programmausführung zum Aufruf der importierten Bibliotheksfunktionen verwendet. FirstThunk wird damit zu einem Teil der Import Address Table (IAT). Die gesamte IAT entsteht durch die Zusammenfassung der FirstThunks aller Import Deskriptoren.

Import Address Table

OriginalFirstThunk und FirstThunk bzw. IAT sind Arrays aus Elementen des Verbunddatentyps `IMAGE_THUNK_DATA`, der entweder die Adressen der Importe oder die Einträge der INT/ILT aufnehmen kann.

B

Beispiel 4

Die Abbildungen 13 und 14 zeigen den „Mechanismus“.

OriginalFirstThunk und FirstThunk zeigen auf Arrays vom Typ `IMAGE_THUNK_DATA`. Beide Arrays haben zunächst identische Inhalte, die Array-Elemente zeigen auf dieselben Namen/Ordinale der INT/ILT.

Nach dem Laden der PE-Datei in den Speicher bleibt OriginalFirstThunk unverändert. Die Array-Elemente von FirstThunk zeigen jetzt aber nicht mehr auf die Namen/Ordinale der INT/ILT, sondern beinhalten die Adressen der jeweiligen Importe.

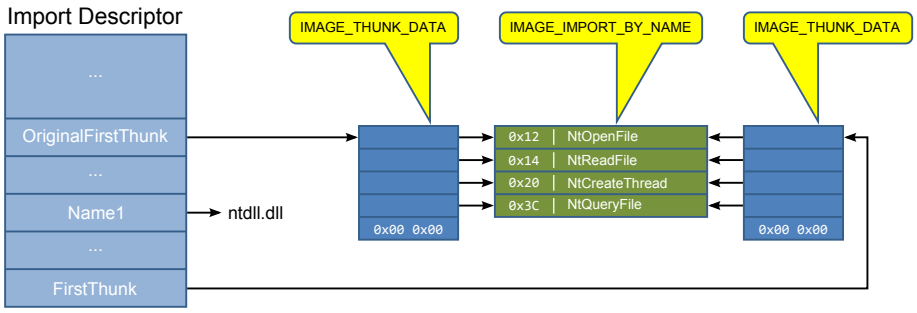


Abb. 13: Vor dem Aufbau der IAT

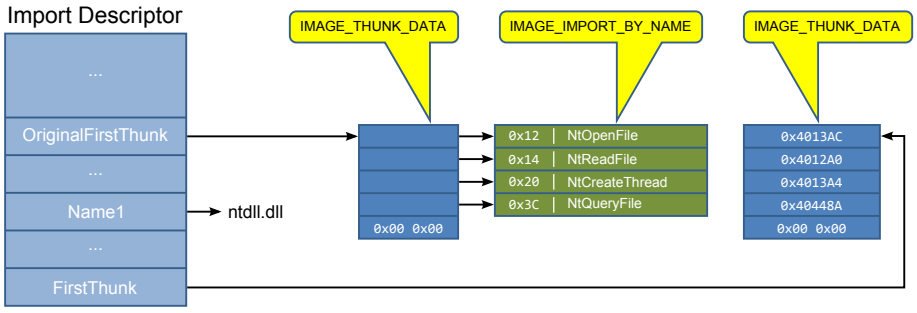


Abb. 14: Nach dem Aufbau der IAT

Der Aufruf importierter Funktionen erfolgt über die in der IAT hinterlegten Adressen. Ein solcher Aufruf erfolgt entweder direkt in der Form

```
call [IAT_Entry_CreateFileA],
```

wobei `IAT_Entry_CreateFileA` auf die in der IAT stehende Adresse von *CreateFileA* verweist, oder über eine *Jump Table* in der Form

```
call JmpTable_CreateFileA.
```

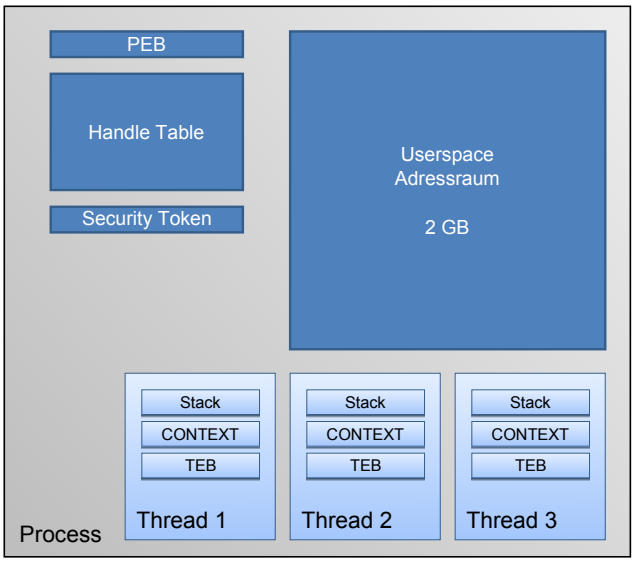
Trampolin

In der Jump Table – bildhaft auch als *Trampolin* bezeichnet – sind die eigentlichen Aufrufe der importierten Funktionen zusammengefasst. Für den gezeigten Aufruf existiert dort also folgender Eintrag:

```
JmpTable_CreateFileA:
   jmp [IAT_Entry_CreateFileA]
```

7 Windows-Prozesse

Abb. 15: Prozess-komponenten (Prozessrahmen)



Ein Windows-Prozess hat den in Abb. 15 gezeigten logischen Rahmenaufbau. Er beinhaltet einen eigenen Adressraum von 2 GB Größe im Userspace, die Verwaltungsdaten im PEB (*Process Environment Block*), die *Handle Table*, das *Security Token* und einen oder mehrere Threads. Die Threads wiederum bestehen aus Verwaltungsdaten im TEB (*Thread Environment Block*), einem eigenen Stack und dem Kontext (Instruction Pointer und Registerinhalte, die bei einem Thread(Prozess)-Wechsel aus bzw. in den physikalischen Prozessor entladen bzw. daraus geladen werden müssen). PEB und TEB werden in Abs. 7.1 detaillierter betrachtet.

Security Token

Das Security Token bzw. Access Token enthält Informationen über die Benutzerrechte, die sich aus der Art der Benutzeranmeldung bei Windows ergeben. Grob unterscheidet Windows bei der Programmausführung zwischen Administratoren mit vollen Rechten und Benutzern mit eingeschränkten Rechten. Das Security Token enthält einen Security Descriptor, der die Rechte eines Benutzers beschreibt. Damit kann bspw. die Ausführung bestimmter Teile einer Anwendung für normale User gesperrt werden.

Handle Table

Die Handle Table dient zur Kommunikation mit dem Kernel. Ein Handle ist ein eindeutiger Referenzwert zu einer vom Betriebssystem verwalteten System-Ressource, wie z. B. Bildschirmobjekte oder einzelnen Dateien auf Festplatten. Wenn ein Anwendungsprogramm eine solche Ressource verwenden will, erhält es durch den Aufruf einer geeigneten Systemfunktion (zum Beispiel zum Öffnen oder Erzeugen von Dateien) als Rückgabewert die Referenz, die zur weiteren Verwendung

der Ressource durch Systemfunktionen anzugeben ist (etwa zum Lesen aus einer Datei). Die Handles eines Prozesses werden in der Handle Table verwaltet.

Es existieren zahlreiche API-Funktionen zur Prozesserstellung, wie z. B. *ShellExecuteA/W, WinExec, CreateProcessA/W, CreateProcessUserA/W* und *CreateProcessWithTokenA/W*, die letztendlich alle bei der API-Funktion *CreateProcessInternalW* landen.

Prozesserstellung

Bei der Prozesserstellung wird der gezeigte Prozessrahmen aufgebaut: Insbesondere wird der neue Prozessadressraum erstellt, Programmcode und Daten werden geladen, die System-DLL *ntdll.dll* wird eingebunden, der PEB aufgebaut und der Haupt-Thread wird erstellt (Stack und Kontext werden initialisiert und der TEB wird erstellt). Anschließend werden der Prozess beim Betriebssystem registriert und der Haupt-Thread gestartet.

Bis zu diesem Zeitpunkt wurde lediglich *ntdll.dll* in den Speicher geladen. Innerhalb des Haupt-Thread werden die zusätzlich benötigten DLLs (ggf. rekursiv) importiert, die Symbole aufgelöst und die IAT gefüllt. Schließlich wird der eigentliche *Entry Point* (EP) des Programms aufgerufen. In einem C-Programm wäre das die *main*-Funktion.

Ein Prozess läuft so lange, bis alle seine Threads beendet sind, oder bis er vorzeitig über die API-Funktionen *TerminateProcess* oder *ExitProcess* abgebrochen wird.

Abb. 16 zeigt das Speicher-Mapping des Prozesses aus Abb. 15. Im Userspace befinden sich alle Datenstrukturen, Code, globale Daten usw. Im Kernelspace liegen die Handle Table und die Exception Handler des Betriebssystems (s. Abs. 8). Die Handle Table liegt zwar im Kernelspace, wird aber in den Adressraum des Prozesses als *Read Only* eingeblendet. Eine Modifikation der Handle Table kann somit nur der Kernel vornehmen.

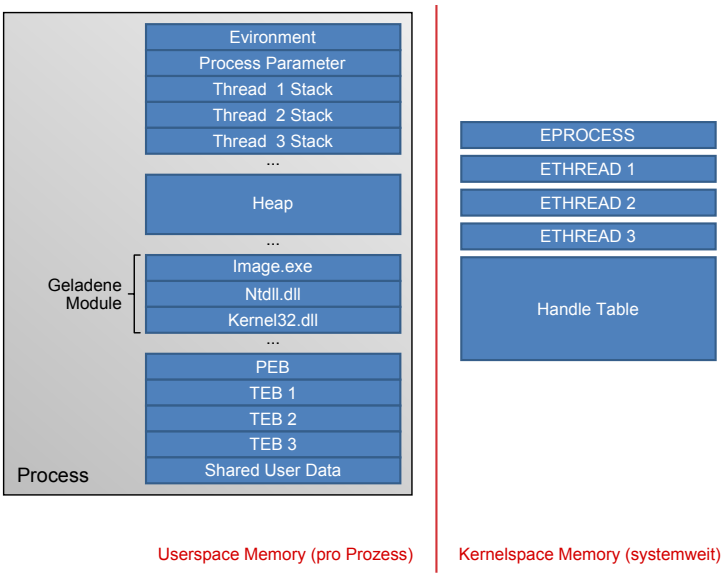


Abb. 16: Prozess im Speicher

7.1 PEB und TEB

Der Windows Loader ist in der *ntdll.dll* implementiert. Er wird immer dann aktiv, wenn ein neuer Prozess initialisiert wird, also insbesondere beim Starten eines Anwenderprogramms. Der Windows Loader initialisiert den Prozess, lädt (mittels der API-Funktion *LoadLibrary*) die benötigten DLLs in den Speicher und füllt die

IAT. Die Prozessinitialisierung beinhaltet den Aufbau der Verwaltungsstrukturen des Prozesses und des initialen Thread.

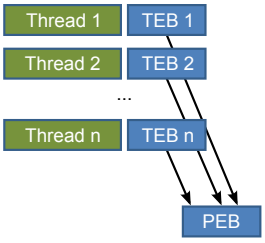
Process Environment Block (PEB)

Die Verwaltungsstrukturen eines Prozesses sind im PEB (*Process Environment Block*) zusammengefasst, der alle zur Ausführung benötigten Prozess- und Loader-Informationen enthält.

Thread Environment Block (TEB)

Jeder Windows-Prozess bestimmt **einen** initialen Thread. Während der Prozessausführung können weitere Threads dynamisch erzeugt werden. Die Verwaltungsinformationen der einzelnen Threads sind im jeweiligen TEB (*Thread Environment Block*) zusammengefasst. Die TEBs verweisen auf den übergeordneten PEB (Abb. 17). PEB und TEB werden in der Regel nicht von den Anwendungen selbst, sondern nur vom Betriebssystem benutzt.

Abb. 17: TEB/PEB



Im TEB sind bspw. Informationen über Adresse und Größe des Stack, den TLS[6] (*Thread Local Storage*) und die Exception Handler (s. Abs. 8) abgelegt.

Abb. 18 zeigt die ersten Einträge im TEB. Der TEB wird über das Segmentregister `fs` referenziert. `fs:[0]` zeigt auf die Liste der Exception Handler, `fs:[0x18]` beinhaltet die lineare Adresse des TEB und `fs:[0x30]` zeigt auf den übergeordneten PEB.

Abb. 18: Anfang des TEB

```
fs register  ►  FS:[0x00]      Structured Exception Handlers (SEH)
                FS:[0x04]      Top of Stack
                FS:[0x08]      Bottom of Stack
                FS:[0x0C]      Subsystem
                FS:[0x10]      Fiber Data
                FS:[0x14]      Arbitrary Data
                FS:[0x18]      Thread Environment Block (TEB)
                FS:[0x1C]      Environment Pointer
                FS:[0x20]      Process ID (PID)
                FS:[0x24]      Thread ID (TID)
                FS:[0x28]      Active RPC Handle
                FS:[0x2C]      Thread Local Storage (TLS) Array
                FS:[0x30]      Process Environment Block (PEB)
                FS:[0x34]      Last Error Number
                FS:[....]      ...
```

Abb. 19 zeigt den Aufbau des PEB. Beispielhaft wollen wir lediglich zwei Strukturen daraus näher betrachten, die bei der Programmanalyse von Bedeutung sein können: `PEB_LDR_DATA` und `RTL_USER_PROCESS_PARAMETERS`.

`PEB_LDR_DATA` enthält Loader-Informationen über geladene DLLs mit der jeweiligen Startadresse, ihrer Größe und der Adresse der Initialisierungsroutine. Außerdem ist die Reihenfolge angegeben, in der diese DLLs in den Speicher geladen und initialisiert wurden. `PEB_LDR_DATA` ist wie folgt aufgebaut und verweist auf drei verkettete Listen:

[6] TLS ist ein Speicherbereich für private Daten eines Thread. Speicher im TLS kann über API-Funktionen oder Konstrukte mancher Hochsprachen allokiert, benutzt und freigegeben werden.

```
+0x000 InheritedAddressSpace : UChar
+0x001 ReadImageFileExecOptions : UChar
+0x002 BeingDebugged    : UChar
+0x003 SpareBool        : UChar
+0x004 Mutant           : Ptr32 Void
+0x008 ImageBaseAddress : Ptr32 Void
+0x00c Ldr              : Ptr32 _PEB_LDR_DATA
+0x010 ProcessParameters : Ptr32 _RTL_USER_PROCESS_PARAMETERS
+0x014 SubSystemData    : Ptr32 Void
+0x018 ProcessHeap      : Ptr32 Void
+0x01c FastPebLock      : Ptr32 _RTL_CRITICAL_SECTION
+0x020 FastPebLockRoutine : Ptr32 Void
+0x024 FastPebUnlockRoutine : Ptr32 Void
+0x028 EnvironmentUpdateCount : Uint4B
+0x02c KernelCallbackTable : Ptr32 Void
+0x030 SystemReserved   : [1] Uint4B
+0x034 AtlThunkSListPtr32 : Uint4B
+0x038 FreeList         : Ptr32 _PEB_FREE_BLOCK
+0x03c TlsExpansionCounter : Uint4B
+0x040 TlsBitmap        : Ptr32 Void
+0x044 TlsBitmapBits    : [2] Uint4B
+0x04c ReadOnlySharedMemoryBase : Ptr32 Void
+0x050 ReadOnlySharedMemoryHeap : Ptr32 Void
+0x054 ReadOnlyStaticServerData : Ptr32 Ptr32 Void
+0x058 AnsiCodePageData : Ptr32 Void
+0x05c OemCodePageData  : Ptr32 Void
+0x060 UnicodeCaseTableData : Ptr32 Void
+0x064 NumberOfProcessors : Uint4B
+0x068 NtGlobalFlag     : Uint4B
+0x070 CriticalSectionTimeout : _LARGE_INTEGER
+0x078 HeapSegmentReserve : Uint4B
+0x07c HeapSegmentCommit : Uint4B
+0x080 HeapDeCommitTotalFreeThreshold : Uint4B
+0x084 HeapDeCommitFreeBlockThreshold : Uint4B
+0x088 NumberOfHeaps    : Uint4B
+0x08c MaximumNumberOfHeaps : Uint4B
+0x090 ProcessHeaps     : Ptr32 Ptr32 Void
+0x094 GdiSharedHandleTable : Ptr32 Void
+0x098 ProcessStarterHelper : Ptr32 Void
+0x09c GdiDCAttributeList : Uint4B
+0x0a0 LoaderLock       : Ptr32 Void
+0x0a4 OSMajorVersion   : Uint4B
+0x0a8 OSMinorVersion   : Uint4B
+0x0ac OSBuildNumber    : Uint2B
+0x0ae OSCSDVersion     : Uint2B
+0x0b0 OSPlatformId     : Uint4B
+0x0b4 ImageSubsystem   : Uint4B
+0x0b8 ImageSubsystemMajorVersion : Uint4B
+0x0bc ImageSubsystemMinorVersion : Uint4B
+0x0c0 ImageProcessAffinityMask : Uint4B
+0x0c4 GdiHandleBuffer  : [34] Uint4B
+0x14c PostProcessInitRoutine : Ptr32     void
+0x150 TlsExpansionBitmap : Ptr32 Void
+0x154 TlsExpansionBitmapBits : [32] Uint4B
+0x1d4 SessionId        : Uint4B
+0x1d8 AppCompatFlags   : _ULARGE_INTEGER
+0x1e0 AppCompatFlagsUser : _ULARGE_INTEGER
+0x1e8 pShimData        : Ptr32 Void
+0x1ec AppCompatInfo    : Ptr32 Void
+0x1f0 CSDVersion       : _UNICODE_STRING
+0x1f8 ActivationContextData : Ptr32 Void
+0x1fc ProcessAssemblyStorageMap : Ptr32 Void
+0x200 SystemDefaultActivationContextData : Ptr32 Void
+0x204 SystemAssemblyStorageMap : Ptr32 Void
+0x208 MinimumStackCommit : Uint4B
```

Abb. 19: PEB

```
PEB_LDR_DATA
 +0x000 Length                           : Uint4B
 +0x004 Initialized                      : UChar
 +0x008 SsHandle                         : Ptr32 Void
 +0x00c InLoadOrderModuleList            : _LIST_ENTRY
 +0x014 InMemoryOrderModuleList          : _LIST_ENTRY
 +0x01c InInitializationOrderModuleList : _LIST_ENTRY
 +0x024 EntryInProgress                  : Ptr32 Void
```

Die von `InLoadOrderModuleList` referenzierte Liste gibt die Reihenfolge an, in der die DLLs in den Speicher geladen wurden, `InMemoryOrderModuleList` verweist auf die Reihenfolge im Speicher und `InInitializationModuleList` auf die Reihenfolge der Initialisierung.

`RTL_USER_PROCESS_PARAMETERS` enthält wichtige Prozessdaten wie den Pfad und den Dateinamen der Anwendung, Kommandozeilenparameter, den DLL-Suchpfad und andere Umgebungsinformationen. Der Aufbau ist wie folgt:

```
RTL_USER_PROCESS_PARAMETERS
 +0x000 MaximumLength    : Uint4B
 +0x004 Length           : Uint4B
 +0x008 Flags            : Uint4B
 +0x00c DebugFlags       : Uint4B
 +0x010 ConsoleHandle    : Ptr32 Void
 +0x014 ConsoleFlags     : Uint4B
 +0x018 StandardInput    : Ptr32 Void
 +0x01c StandardOutput   : Ptr32 Void
 +0x020 StandardError    : Ptr32 Void
 +0x024 CurrentDirectory : _CURDIR
 +0x030 DllPath          : _UNICODE_STRING
 +0x038 ImagePathName    : _UNICODE_STRING
```

```
+0x040 CommandLine      : _UNICODE_STRING
+0x048 Environment      : Ptr32 Void
+0x04c StartingX        : Uint4B
+0x050 StartingY        : Uint4B
+0x054 CountX           : Uint4B
+0x058 CountY           : Uint4B
+0x05c CountCharsX      : Uint4B
+0x060 CountCharsY      : Uint4B
+0x064 FillAttribute    : Uint4B
+0x068 WindowFlags      : Uint4B
+0x06c ShowWindowFlags  : Uint4B
+0x070 WindowTitle      : _UNICODE_STRING
+0x078 DesktopInfo      : _UNICODE_STRING
+0x080 ShellInfo        : _UNICODE_STRING
+0x088 RuntimeData      : _UNICODE_STRING
+0x090 CurrentDirectores : [32] _RTL_DRIVE_LETTER_CURDIR
```

8 Exceptions

Eine Exception ist ein Ereignis während eines Programmablaufs, das die Ausführung von Code außerhalb des normalen Kontrollflusses erfordert bzw. erzwingt. Es existieren zwei Arten von Exceptions: Hardware Exceptions und Software Exceptions.

- Hardware Exceptions werden von der CPU selbst initiiert, bspw. bei einer Division durch 0 oder durch den Zugriff auf eine ungültige Speicheradresse.
- Software Exceptions werden explizit durch die Anwendung selbst oder das Betriebssystem ausgelöst. Beispiele hierfür sind Breakpoints beim Debugging eines Programms oder ungültige Parameter bei einem Funktionsaufruf, die nicht zur Kompilierungszeit entdeckt werden konnten.

Exceptions erfordern eine Exception-Behandlung (*Exception Handling*), also die Ausführung einer Ausnahme- bzw. Fehlerbehandlungsroutine. Diese unterscheidet sich prinzipiell nicht für Software und Hardware Exceptions.

Aus der Sicht des Exception Handling sind drei Arten von Exceptions zu unterscheiden:

1. **Faults**
   Faults sind Exceptions **bei** der Ausführung einer Operation. Der „Fehler“ kann durch den Exception Handler behoben werden, und die Operation wird danach wiederholt.
2. **Traps**
   Traps sind Exceptions **nach** der Ausführung einer Operation. Der „Fehler“ kann durch den Exception Handler behoben werden, und die nächste Operation wird danach ausgeführt.
3. **Aborts**
   Aborts sind kritische Fehler, die zu einem Prozessabbruch führen.

Exception Handler

Exceptions werden von einem *Exception Handler* abgewickelt. Dies sind allgemein Routinen, die auf eine Ausnahmesituation oder einen Fehlerfall in einem Prozess angemessen reagieren sollen.

Structured Exception Handling

Windows bietet das sogenannte *Structured Exception Handling* (SEH) an. Da der Ort (im Programmcode), an dem eine Exception ausgelöst wird, nicht voraussagbar

ist, sind Anwendungscode und Exception Handler voneinander getrennt. Ohne weitere Vorkehrungen sind Exception Handler dann Routinen im Kernelmode, die die Standardbehandlung von Exceptions beinhalten.[7] Die Exception selbst wird durch einen Exceptioncode (Art der Exception) und Kontextinformationen (Instruction Pointer und andere CPU-Register) spezifiziert.

SEH bietet die Möglichkeit, bestimmte Exceptions direkt im Programm abzufangen. Mit entsprechenden Hochsprachenkonstrukten können Code Blocks durch einen eigenen, zugeordneten Exception Handler „geschützt“ werden. Geschützt ist der Code Block dann in dem Sinne, dass der Standard-Exception-Handler zunächst außen vor bleibt.

B

Beispiel 5

Nehmen wir den Fall einer Division durch 0 an. Was soll bei einem solchen Fehler passieren? Der Standard-Exception-Handler von Windows würde vermutlich den Prozess in dem Fall beenden. Das Programm sollte die Möglichkeit haben, diese Ausnahme selbst zu bedienen und eventuell „milder“ zu reagieren. In C++ bspw. existiert das **try-catch Statement**, mit dessen Hilfe ein eigener Handler für diesen Fall programmiert werden kann.

Abb. 20 zeigt eine solche Situation mit zwei Code Blocks und den zugeordneten Exception Handlern.

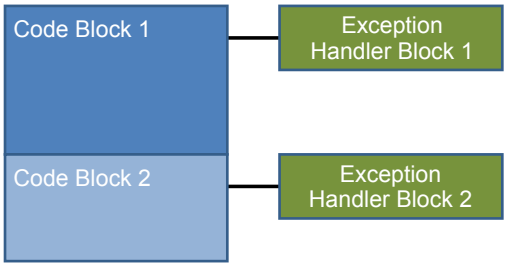


Abb. 20: Structured Exception Handling (SEH)

SEH ist in Form einer verketteten Liste implementiert. Ein Eintrag im TEB zeigt auf das erste Listenelement, wie dies in Abb. 21 dargestellt ist. Die Liste besteht aus *Exception Registration Records*. Diese Records verweisen auf den nächsten Record und den eigentlichen Exception Handler. `0xffffffff` zeigt das Ende der Liste an.

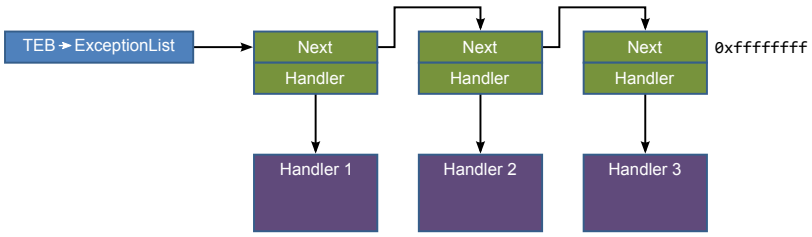


Abb. 21: Implemen- tierung SEH

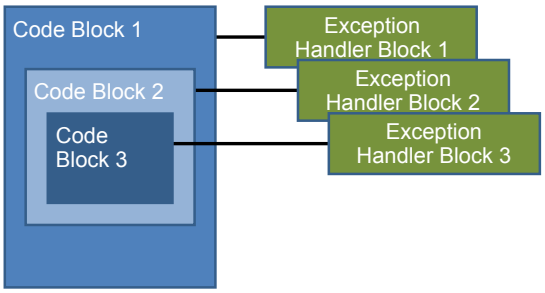
Eine Erweiterung von SEH ist das *Frame Based Exception Handling* (FBEH). Wird bspw. eine Exception innerhalb einer verschachtelt aufgerufenen Funktion ausgelöst, so kann der zugeordnete Exception Handler in diesem Kontext nicht immer die Exception angemessen bedienen. Er gibt die Kontrolle daher an den Exception

Frame Based Exception Handling

[7] und die meist nichtssagenden Windows-Fehlermeldungen erzeugen.

Handler der aufrufenden Funktion weiter. Dieses Verfahren wird ggf. rekursiv fortgesetzt. Die Realisierung erfolgt über einen *Exception Handler Stack*. Dies ist in Abb. 22 anhand eines Beispiels dargestellt. Wenn Code Block 3 eine Exception auslöst, dann versucht zunächst Handler 3, diese zu bedienen. Gelingt dies nicht, so wird die Exception-Behandlung an Handler 2 des aufrufenden Code Block 2 weitergereicht, dann eventuell an Handler 1. Erst wenn alle Handler des Stack die Exception nicht bedienen können, wird der Standard-Exception-Handler aktiviert, also der von Windows.

Abb. 22: Frame Based Exception Handling (FBEH)



Vectored Exception Handling

Eine andere Erweiterung von SEH ist das *Vectored Exception Handling* (VEH). Hierbei beziehen sich die Exception Handler nicht auf Code Blocks, sondern auf den gesamten Thread. Realisiert wird VEH durch eine beliebig lange Liste von Handlern, die bei einer Exception der Reihe nach aufgerufen werden. Werden VEH und FBEH/SEH gleichzeitig implementiert, so werden zunächst die VEH Handler aufgerufen, dann die von FBEH/SEH.

9 Zusammenfassung

Sie haben das Betriebssystem Windows insbesondere aus dem Blickwinkel der Programmanalyse kennengelernt. Für eine erfolgreiche Analyse sind einige Punkte besonders wichtig:

- Programme, die auf Windows-Rechnern funktionieren sollen, müssen mit dem Betriebssystem interagieren. Dies geschieht durch den Aufruf von Bibliotheksfunktionen über eine klar definierte Schnittstelle (API). Systemaufrufe (Syscalls) bilden dabei den Übergang in den eigentlichen Betriebssystemkern.
- Windows-Programme benutzen ein einheitliches Format (PE-Format). Darin sind in einer streng strukturierten Weise neben dem Programm selbst alle Informationen enthalten, um aus einem Programm einen Prozess zu erzeugen. Die Analyse des PE-Formats liefert bereits viele Anhaltspunkte für das mögliche Verhalten eines Programms zur Laufzeit.
- Der Windows Loader macht aus einer Datei im PE-Format einen Prozess. Die dabei entstehenden Speicher- und Datenstrukturen sind für die Analyse der Dynamik eines Programms, also seines Laufzeitverhaltens, wesentlich.
- Windows benutzt mit SEH eine besondere Form der Ausnahmebehandlung. Diese wird von Schadsoftware gerne benutzt.

10 Übungen

Ü

Übung 1

Das folgende Listing zeigt die (reale) Übersetzung des Beispielprogramms 1 aus Abs. 4 in Assembler. Beschreiben Sie den Ablauf des Programms. Analysieren Sie nun die Aufrufe der API-Funktion *MessageBoxA*. Recherchieren Sie mit Hilfe des Internet die Parameter und erklären Sie ihre Bedeutung.

```
.text:004013B0  push  ebp
.text:004013B1  mov   ebp, esp
.text:004013B3  sub   esp, 28h        ; hWnd
.text:004013B6  mov   [esp+28h+var_1C], 41h
.text:004013BE  mov   [esp+28h+var_20], offset aEineKleineBox ;
                      "Eine kleine Box"
.text:004013C6  mov   [esp+28h+var_24], offset aHabenSieDasPro;
                      "Haben Sie das Programm verstanden?"
.text:004013CE  mov   [esp+28h+var_28], 0
.text:004013D5  call  MessageBoxA
.text:004013DA  sub   esp, 10h        ; hWnd
.text:004013DD  mov   [ebp+var_C], eax
.text:004013E0  cmp   [ebp+var_C], 1
.text:004013E4  jnz   short loc_40140F
.text:004013E6  mov   [esp+28h+var_1C], 40h
.text:004013EE  mov   [esp+28h+var_20], offset unk_4030B3
.text:004013F6  mov   [esp+28h+var_24], offset aDasIstPrima ;
                      "Das ist prima!"
.text:004013FE  mov   [esp+28h+var_28], 0
.text:00401405  call  MessageBoxA
.text:0040140A  sub   esp, 10h        ; hWnd
.text:0040140D  jmp   short loc_40143C
.text:0040140F loc_40140F:
.text:0040140F  cmp   [ebp+var_C], 2
.text:00401413  jnz   short loc_40143C
.text:00401415  mov   [esp+28h+var_1C], 40h
.text:0040141D  mov   [esp+28h+var_20], offset unk_4030B3
.text:00401425  mov   [esp+28h+var_24], offset aSchadeDannScha;
                      "Schade, dann schauen Sie es sich ..."
.text:0040142D  mov   [esp+28h+var_28], 0
.text:00401434  call  MessageBoxA
.text:00401439  sub   esp, 10h
.text:0040143C loc_40143C:
.text:0040143C  mov   eax, 0
.text:00401441  leave
.text:00401442  retn  10h
```

Anmerkungen (Das Listing wurde mit dem Disassembler IDA erzeugt.):

1. Der Befehl `leave` beseitigt einen Stack Frame. Hinter ihm verbirgt sich nichts anderes als der bekannte Epilog: `mov esp, ebp ; pop ebp`. `leave` ist also eine abkürzende Schreibweise für eigentlich zwei Befehle, die in der CPU ablaufen.
2. `var_1C, var_20, var_24, var_28` sind Namen von Variablen, die der Di-

sassembler IDA generiert. Sie beinhalten hier die Werte `-1Ch, -20h, -24h, -28h`.

3. `unk_4030B3` verweist auf einen leeren String, also auf ein Null-Byte.

Ü

Übung 2

Diese Übung soll Sie sich mit der Windows-API-Schnittstelle besser vertraut machen.

Schreiben Sie ein Programm, das eine beliebige Datei einliest und deren Inhalt in eine andere Datei ausgibt. Verwenden sie dafür ausschließlich die Windows-API-Funktionen (*CreateFile, ReadFile* etc.).

Hinweis: Es ist keine spezielle Programmiersprache zur Lösung der Aufgaben vorgeschrieben, solange die entsprechenden Windows-Funktionen direkt aufgerufen werden. Es wird allerdings empfohlen, die Aufgabe in C zu programmieren: Wie im Beispielprogramm in Abs. 4 beschrieben muss dazu die Datei `windows.h` eingebunden werden. Deklarationen und Erklärungen zu den benötigten API-Funktionen sind leicht im Internet zu finden.

Ü

Übung 3

Schreiben Sie ein Programm in Assembler oder C, das die API-Funktion *IsDebuggerPresent* verwendet. Diese Funktion überprüft, ob ein Programm unter einem Debugger – möglicherweise zu Analysezwecken – ausgeführt wird. Wenn das Programm einen Debugger entdeckt, soll es irreführende Dinge tun, ansonsten soll es irgendwelche „vernünftigen“ Ergebnisse produzieren. Was das Programm jeweils produziert, bleibt Ihrer Fantasie überlassen.

Ü

Übung 4

Schreiben Sie ein Programm, das sein eigenes PE-Format analysiert. Es soll „interessante Teile“ des PE Header und der Section Table ausgeben, je mehr, desto besser. Allerdings ist keinerlei Vollständigkeit gefordert.

Anmerkung: Mit dem Programm *PEview* aus dem Kursmaterial können Sie Ihre Ergebnisse überprüfen.

Liste der Lösungen zu den Kontrollaufgaben

Lösung zu Kontrollaufgabe 1 auf Seite 10

Die Nachteile der statischen Bindung sind die Vorteile der dynamischen Bindung und umgekehrt:

Die Nachteile einer dynamischen Bindung sind das langsame Laden und Starten einer Anwendung und die Abhängigkeit vom Vorhandensein einzelner Bibliotheken. Die Vorteile bestehen neben der möglichen Mehrfachverwendung von Bibliotheken zum einen in einer kleineren, kompakteren Anwendungsdatei und zum anderen in der entfallenden Notwendigkeit, die Anwendungen bei einem Update der Bibliothek neu zu kompilieren.

Verzeichnisse

I. Abbildungen

II. Exkurse

III. Literatur

Thorsten Holz. *Program Analysis*. Vorlesung, Ruhr-Universität Buchum, 2012.

Intel. *Intel 80386, Programmers Reference Manual*.
http://css.csail.mit.edu/6.858/2011/readings/i386.pdf, 1987.

Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*.
http://www.intel.com/content/www/us/en/architecture-and-technology/
64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html, 2012.

Kip R. Irvine. *Assembly Language for Intel-Based Computers (5th Edition)*. Prentice Hall, 2006.

Microsoft. Microsoft pe and coff specification.
http://msdn.microsoft.com/en-us/windows/hardware/gg463119.aspx, 2010.

Gary Nebbett. *Windows NT/2000 Native API Reference*. Macmillan Technical Publishing, 2000.

Matt Pietrek. An in-depth look into the win32 portable executable file format. *February 2002 issue of MSDN Magazine*, 2002.
http://msdn.microsoft.com/en-us/magazine/bb985992.aspx.

Joachim Rohde. *Assembler GE-PACKT, 2. Auflage*. Redline GmbH, Heidelberg, 2007.

M. E. Russinovich and D. A. Solomon. *Windows Internals*. Microsoft Press Corp, 2012.