

Program Analysis

Lecture 07: Windows I
Winter term 2011/2012

Dipl.-Inform. Carsten Willems

Übersicht

- 5.1 Einführung
- 5.2 Anwendungen und Bibliotheken
- 5.3 Die Windows-API und Systemaufrufe
- 5.4 Das PE-Format
- 5.5 Der Windows-Loader
- 5.6 Prozesse
- 5.7 Ausnahme-Behandlung

Windows

5.1 Einführung

Microsoft Windows

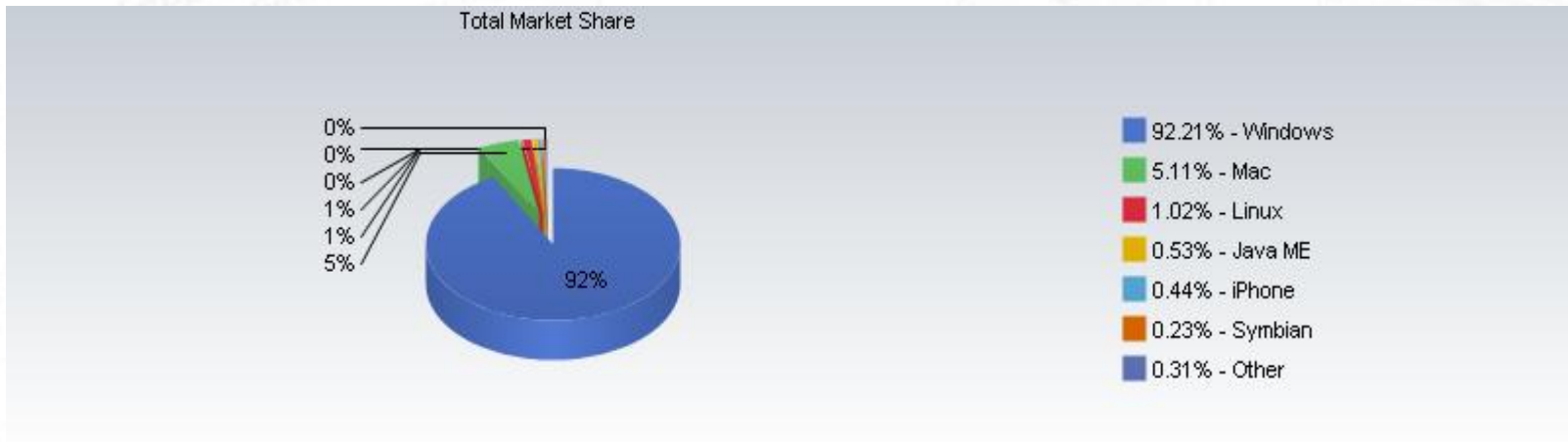
- *Closed-source* Betriebssystem
- 16Bit, 32Bit, 64Bit
- Verschiedene Architekturen
 - Workstation, Server, Mobile/PDA, meist x86

Zeitleiste der Windows -Versionen von 1985 bis heute																										
Typ	1980er					1990er										2000er										2010er
	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0
16-Bit	1.0		Windows 2.0			Windows 3.0	Windows 3.1		Windows 3.11																	
9x-Linie											Windows 95			Windows 98		Windows ME										
NT-Linie									NT 3.1		NT 3.5	NT 3.51	NT 4.0		2000	XP				Windows Vista		Windows 7				
Server-OS auf NT-Basis									NT 3.1 Srv.		NT 3.5 Srv.	NT 3.51 S.	NT 4.0 Server		2000 Server		Server 2003				Server 2008					
CE-Linie														CE 1.0	CE 2.0		CE 3.0		CE 4.0		Mobile 2003	Mobile 5.0	Mobile 6.0	Mobile 6.1	Mobile 6.5	Mobile 7.0

Quelle: Wikipedia

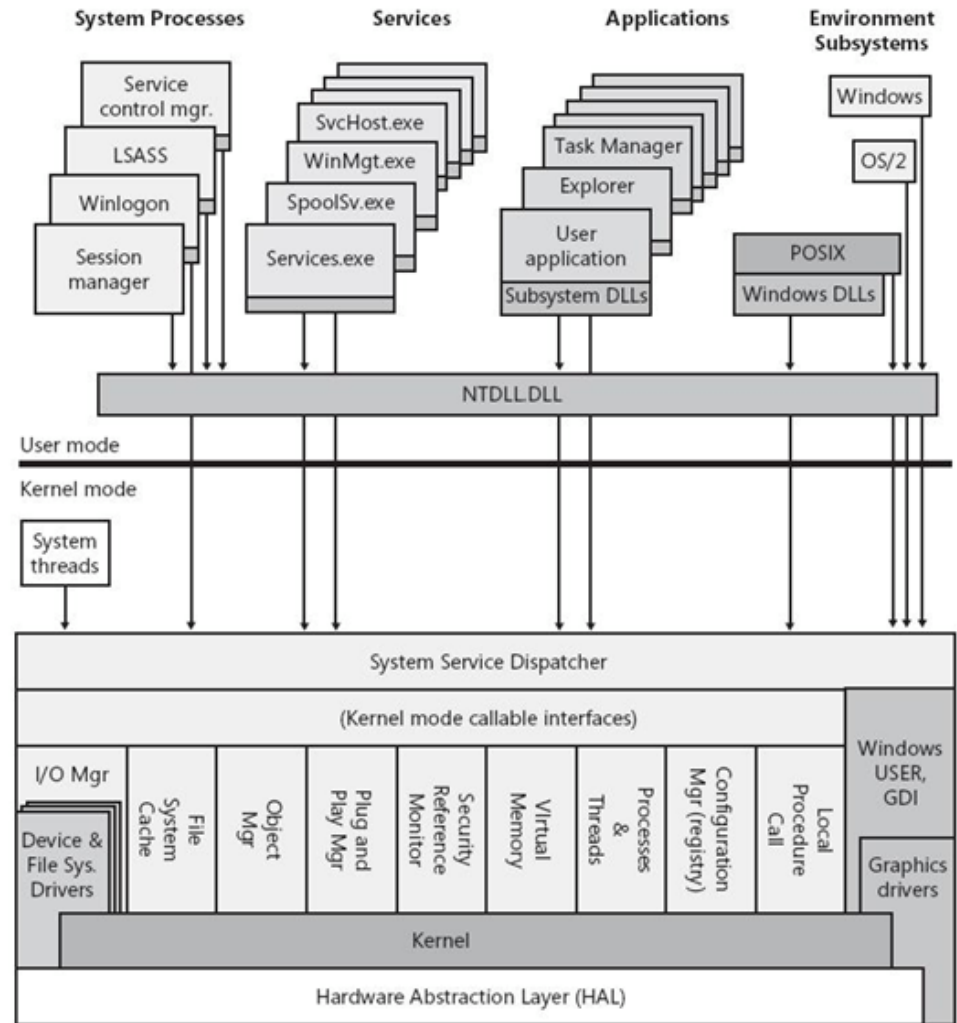
Marktanteil

- Marktanteil von 92,21%
 - „MarketShare – Net Applications“, Stand Dez. 2009
- Daher besonders interessant für RE
 - Malware
 - Security, Vulnerabilities
 - DRM, Kopierschutz



Bestandteile von Windows

- Usermode
 - Systemprozesse
 - Dienste
 - Subsysteme
 - Anwendungen
 - Windows API
- Kernelmode
 - Executive
 - Kernel
 - Gerätetreiber
 - HAL



Quelle: Windows Internals Book
Lehrstuhl Praktische Informatik 1

Windows

5.2 Anwendungen und Bibliotheken

Dateitypen

- Anwendungen (.exe, .scr)
- Dynamische Bibliotheken (.dll, .ocx)
- Kerneltreiber (.sys, .vxd, .exe, .dll)

➔ verwenden alle ***Portable Executable–Format (PE)***

- Skripte
 - Shell-Skripte (.cmd, .bat)
 - Skriptsprachen (.vbs, .js)

➔ hier nicht weiter betrachtet

Windows Anwendungen

- Windows-Systemprogramme
 - Ntoskrnl.exe (eigentlicher Kernel)
 - Winlogon.exe (Benutzeranmeldung, Aktivierung)
 - Csrss.exe (Windows-Subsystem)
 - Explorer.exe, Services.exe, ...
- Windows-Hilfsprogramme
 - Calc.exe, notepad.exe
- Dritthersteller Programme
 - Winword.exe, miranda.exe

Anwendungen

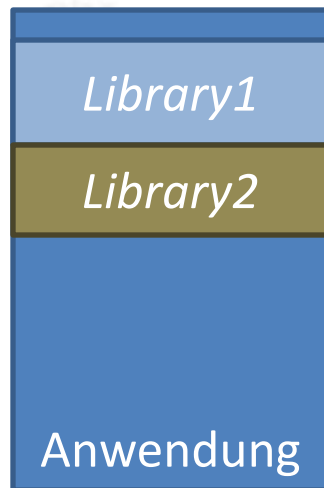
- Anwendungen beinhalten
 - Code = Funktionen, Callback-Routinen, ...
 - Daten = Strings, Keys, Fenster-Layout, Bilder, ..
 - Optional auch Debug-Informationen
- Anwendungen verwenden *Bibliotheken*
 - Windows Standardbibliotheken
 - *Kernel32.dll, user32.dll, ws2_32.dll*
 - Compiler Bibliotheken
 - *Msvcrt80.dll, msvcrt80d.dll*
 - Dritthersteller Bibliotheken
 - *Skype4.dll, flash10.ocx*

Bibliotheken

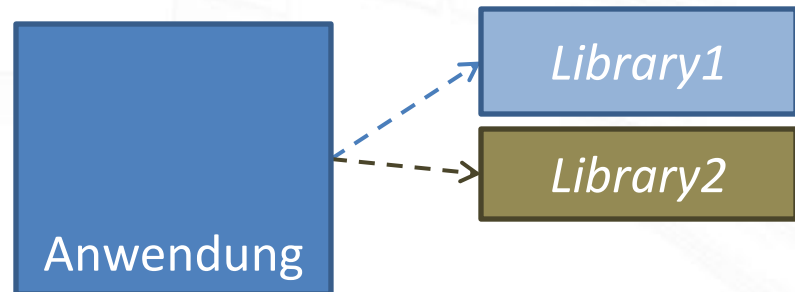
- Sind sehr *ähnlich* zu Anwendungen
 - beinhalten ebenfalls *Code* und *Daten*
- Sind nicht *alleine* ausführbar
 - haben *keine* main()-function, nur eine DLLMain()
- *Exportieren Symbole*
 - Funktionen oder Daten (Variablen)
- Anwendungen *importieren* diese
 - via Namen („CreateFileA“) oder Ordinalzahl („0x30“)

Bindung mit Bibliothekscode

- Bindung mit Bibliotheksdatei
 - *Statisch*: Bibliothek zur Compilezeit in Binary einbinden
 - *Dynamisch*: Bibliothek zur Laufzeit in den Speicher laden
 - **DLL = Dynamic Link Library**



Statische
Bindung



Dynamische
Bindung

Statische Bindung

- Bibliotheksfunktionen und –daten werden **mit in** der Anwendungsdatei gespeichert
- Vorteile:
 - schnelleres Starten der Anwendung
 - keine Abhängigkeitsprobleme („dll hell“)
- Nachteile:
 - Größere Anwendungsdatei
 - Keine Mehrfachverwendung von Libraries möglich
 - Bei Bibliotheksupdates müssen Anwendungen neu erstellt werden

Dynamische Bindung

- Anwendung enthält lediglich Referenz auf
 - verwendete Bibliotheken
 - Importierte Funktionen / Daten
- Bibliothek werden
 - mit Anwendung ausgeliefert
 - Oder befinden sich bereits auf dem Zielrechner
- Nachteile / Vorteile
 - Gegenteil der statischen Bindung 😊

Windows

5.3 Windows API

Windows API

- Aufgaben eines Betriebssystems
 - Verwaltung der Betriebsmittel
 - Speicher, Ein-/Ausgabegeräte, ...
 - Steuerung der Ausführung von Programmen
- Windows API (*Application Programmer Interface*)
 - Schnittstelle zwischen OS und Anwendungsprogrammen
 - Besteht aus vielen **DLLs** im *system32*-Verzeichnis
 - Implementiert in C und Assembler

Beispiel

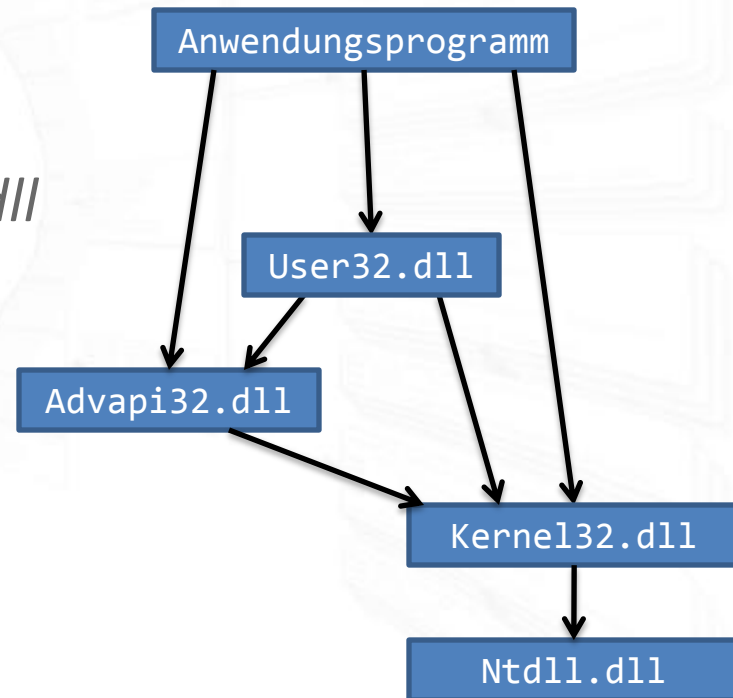
- Erstellen/Öffnen von Dateien: *Kernel32!CreateFileA*

```
HANDLE WINAPI CreateFile(  
    __in LPCTSTR lpFileName,  
    __in DWORD dwDesiredAccess,  
    __in DWORD dwShareMode,  
    __in_opt LPSECURITY_ATTRIBUTES lpSecurityAttributes,  
    __in DWORD dwCreationDisposition,  
    __in DWORD dwFlagsAndAttributes,  
    __in_opt HANDLE hTemplateFile );
```

- Viele API-Funktionen gibt es als A und W-Version
 - A: String-Parameter sind ASCII-Strings
 - W: String-Parameter sind Unicode-Strings
 - Bsp: CreateFileA vs. CreateFileW

DLL-Hierarchie

- Hierarchie von DLLs
 - Native API: *ntdll.dll*
 - Kernfunktionen: *kernel32.dll*
 - Basisfunktionen: *advapi32.dll*
 - Grafik/Fenster: *user32.dll*, *gdi32.dll*
 - TCP/IP: Winsock *ws2_32.dll*
 - viele, viele mehr ...



Native API

WINDOWS® NT/2000
NATIVE API REFERENCE



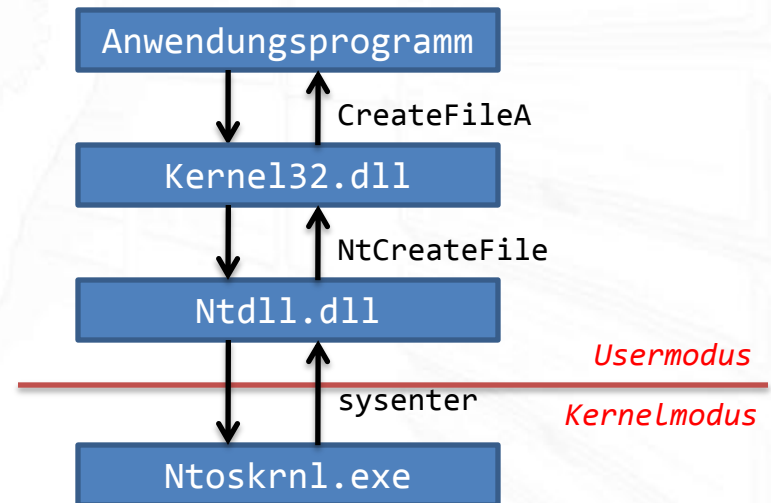
- *Interne Windows NT API*
 - Häufige Änderungen an Schnittstelle
 - Meistenteils undokumentiert
 - Sehr gute (RE-)Referenz: Gary Nebbett
- Aufruf
 - nicht direkt von Anwendungsprogrammen benutzt
 - sondern nur indirekt von Windows API, z.B. kernel32.dll
- Implementierung
 - Usermode Komponente: *ntdll.dll*
 - Meiste Funktionen nur Wrapper für Kernel-Aufruf
 - Kernelmode Komponente: *ntoskrnl.exe*

Kernel-/Usermodus

- Kernel- oder Usermodus?
 - Bestimmt durch *current privilege level* (CPL)
 - Bit 0 und 1 des cs-Segment-Descriptors
 - Hardware-Zugriffe nur im KM
 - Privilegierte Operationen nur im KM
 - in, out, write TLB
 - Zugriff auf *Kernelspace-Memory* nur vom KM
 - Zugriff auf *Userspace-Memory* vom UM und KM
- Oft Wechsel in Kernelmodus notwendig

Systemaufruf

- Wechsel in den Kernelmodus = *Systemaufruf*
 - Realisiert über
 - `int 3` (alt)
 - `syscall` (AMD)
 - `sysenter` (Intel)
- Kein direkter Systemaufruf aus Anwendungen, sondern Windows API führt Systemaufrufe durch



Systemaufruf

- Systemaufruf implementiert in ntdll.dll

NtOpenFile:

```
mov eax, 0x74                ; SSID(OpenFile) = 0x74
mov edx, KiFastSystemCall
call ds:[edx]
retn 18                      ; clean up stack (stdcall)
```

KiFastSystemCall:

```
mov edx, esp                ; current userstack
sysenter                    ; call into kernel
retn
```

- EAX = *SystemServiceID* (SSID)
- EDX = Zeiger auf Usermode Stack
 - Kernel verwendet eigenen Stack

Windows

5.4 PE-Format

Portable Executable Format

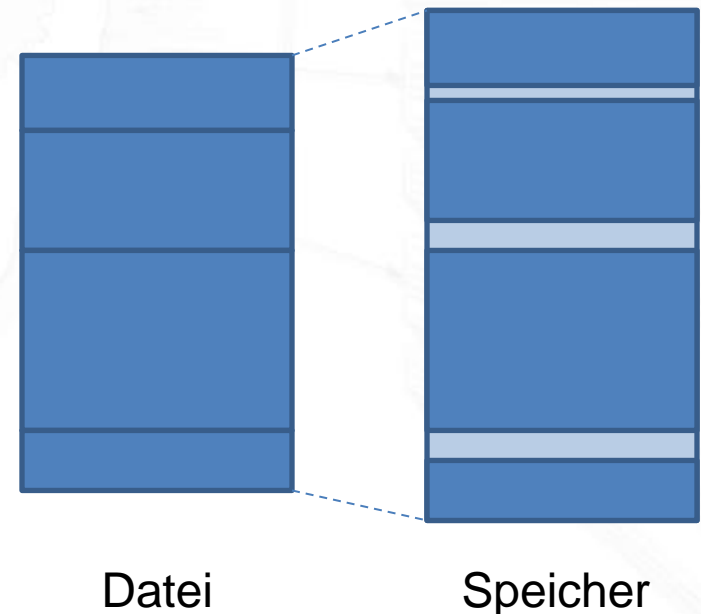
- Abgeleitet vom *Common Object File Format (COFF)*
- ***Portable***
 - ermöglicht das Laden (und Ausführen?) von Anwendungen unter verschiedenen OS
- ***Executable***
 - beschreibt die Struktur einer *ausführbaren Datei*
 - Anwendungen
 - Bibliotheken
 - (Geräte)-Treiber

Referenzen

- Informationen zum PE-Format
 - ARTeam Tutorial „PE File Format“
<http://arteam.accessroot.com/arteam/site/download.php>
 - Microsoft PE Executable and COFF Specification
<http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx>
 - An In-Depth Look into the Win32 PE File Format
<http://msdn.microsoft.com/en-us/magazine/bb985992.aspx>
- Informationen zum COFF-Format
 - <http://en.wikipedia.org/wiki/COFF>

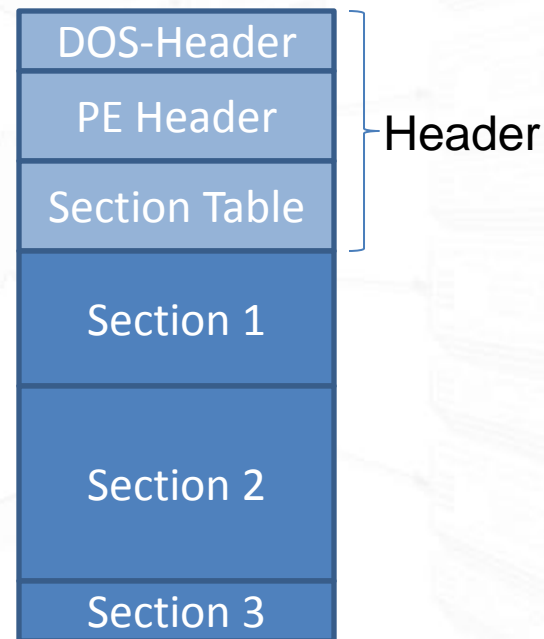
PE-Format und Loader

- PE-Format unterstützt Windows-Loader beim
 - Erstellung eines neuen Prozesses
 - Nachladen von DLLs
- Nahezu *1:1-Mapping* zwischen Datei und Speicher



Aufbau PE-Datei

- Header
 - DOS-Header
 - PE-Header
 - Section Table
- Beliebige viele Sections
 - Code
 - Daten
 - Debug-Informationen
 - Ressourcen (Icon, ...)



Windows

5.4.1 PE-Header

DOS-Header

```

00340000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ.....ÿÿ..
00340010  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  .....@.....
00340020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
00340030  00 00 00 00 00 00 00 00 00 00 00 00 E8 00 00 00  .....è...
00340040  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  e.!.Í!LÍ!Th
00340050  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
00340060  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS
00340070  6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.
00340080  AA 33 E4 9C EE 52 8A CF EE 52 8A CF EE 52 8A CF  3äæîRŠİîRŠİîRŠİ
00340090  C9 94 F7 CF E9 52 8A CF EE 52 8B CF 8F 52 8A CF  É”÷îéRŠİîR<îRŠİ
003400A0  C9 94 F1 CF E9 52 8A CF C9 94 E7 CF AE 52 8A CF  É”ñîéRŠİÉ”çî®RŠİ
003400B0  C9 94 F0 CF EF 52 8A CF C9 94 E4 CF FE 52 8A CF  É”ðîîRŠİÉ”äîpRŠİ
003400C0  C9 94 F4 CF EF 52 8A CF C9 94 F6 CF EF 52 8A CF  É”òîîRŠİÉ”öîîRŠİ
003400D0  C9 94 F2 CF EF 52 8A CF 52 69 63 68 EE 52 8A CF  É”òîîRŠİîRichîRŠİ
003400E0  00 00 00 00 00 00 00 00 50 45 00 00 4C 01 04 00  .....PE.
003400F0  46 EC A3 44 00 00 00 00 00 00 00 00 00 E0 00 02 21  FîfD.....à.!
  
```

MZ = Magic Byte
of DOS-Header

E_lfanew = offset of
PE-Header start

is program cannot
be run in DOS
mode....\$.

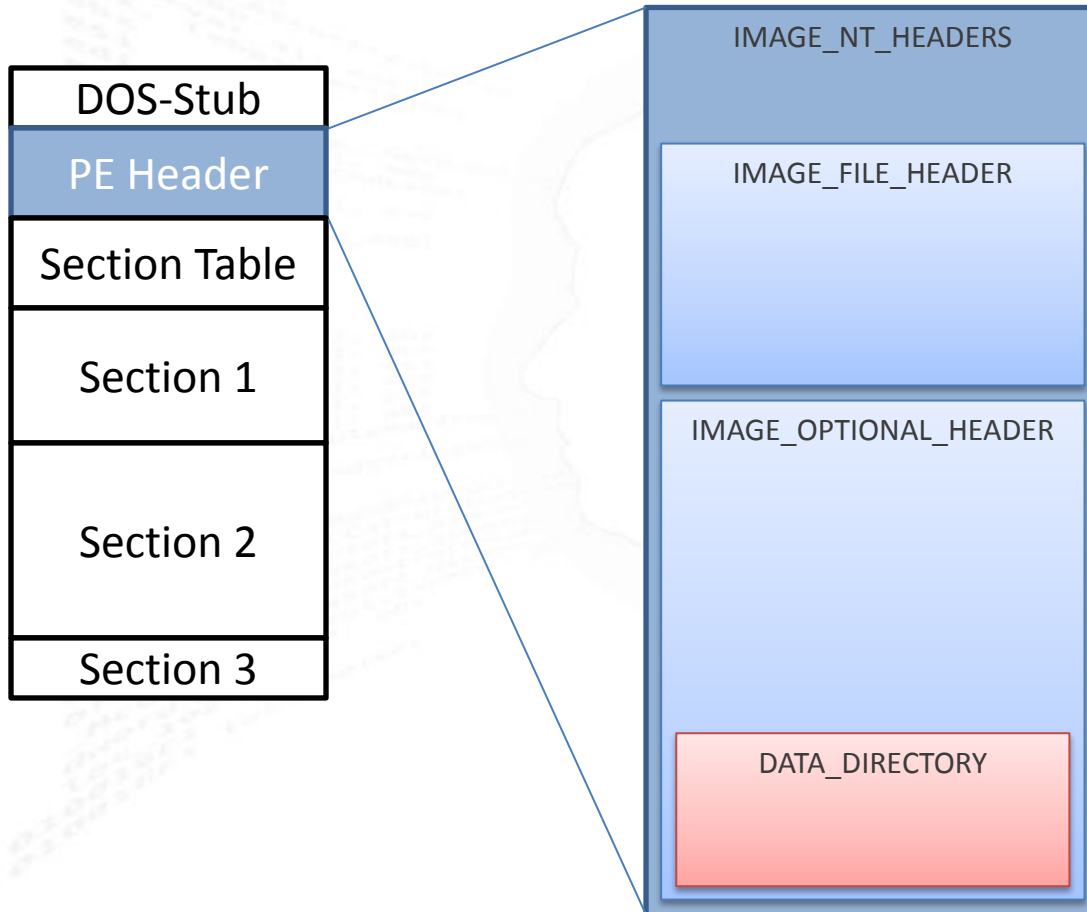
DOS Stub

PE = Magic Byte
of PE Header

IMAGE_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

PE-Header



IMAGE_NT_HEADERS

- PE-Header = IMAGE_NT_HEADERS

+0x000 Signature	: Uint4B	; PE
+0x004 FileHeader	: _IMAGE_FILE_HEADER	
+0x018 OptionalHeader	: _IMAGE_OPTIONAL_HEADER	

- 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

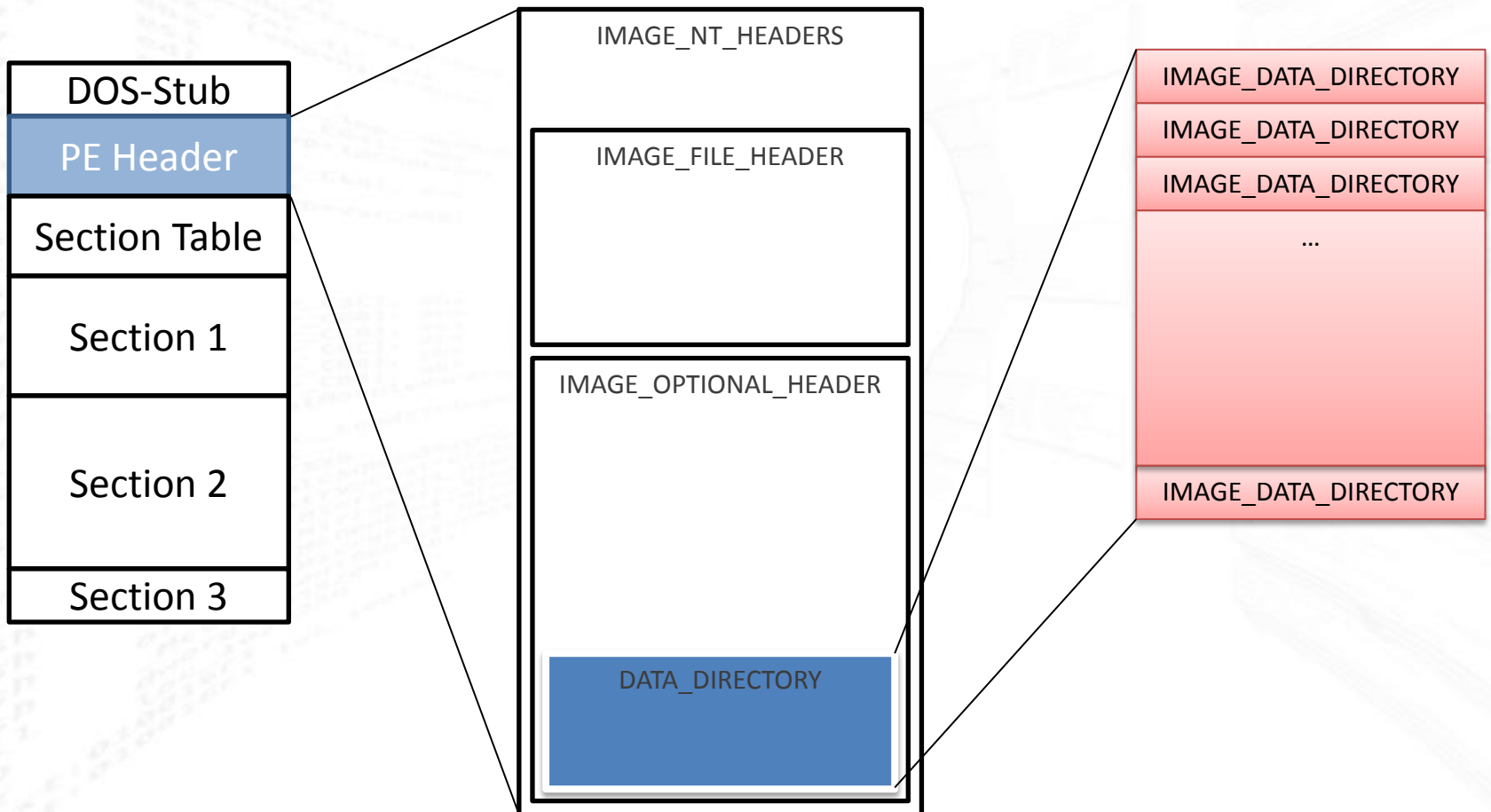
IMAGE_OPTIONAL_HEADER

+0x000	Magic	: Uint2B
+0x002	MajorLinkerVersion	: Uchar
+0x003	MinorLinkerVersion	: Uchar
+0x004	SizeOfCode	: Uint4B
+0x008	SizeOfInitializedData	: Uint4B
+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

Header-Felder

- AddressOfEntryPoint
 - Einstiegspunkt: erste Instruktion die ausgeführt wird
- ImageBase
 - Gewünschte virtuelle Startadresse
 - Wenn besetzt: *Relocation* notwendig
 - Address space layout randomization (ASLR)
- Granularität: SectionAlignment und FileAlignment
 - Auf Speicher-Ebene, normalerweise 4 KB (Page-Size)
 - Auf Datei-Ebene, normalerweise 512 Bytes
 - Beispiel folgt ...

Data-Directory



Data Directory

- Array mit 16 Einträgen:
 - IMAGE_DATA_DIRECTORY
 - +0x000 VirtualAddress : Uint4B
 - +0x004 Size : Uint4B
- Beispiel-Einträge:
 - Import-Directory
 - Liste der importierten Funktionen (aus DLLs)
 - Import Address Table (IAT)
 - Adressen der importierten Funktionen
 - Export-Directory
 - Liste der exportierten Funktionen

Data Directory Einträge

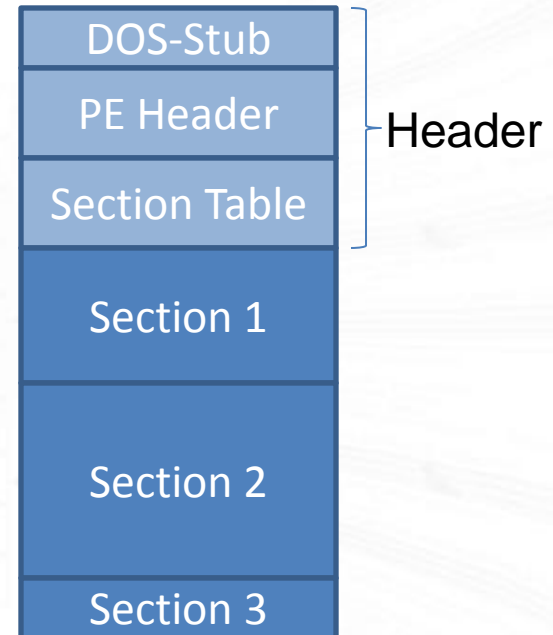
IMAGE_DIRECTORY_ENTRY_EXPORT	= 0;	// Export Directory
IMAGE_DIRECTORY_ENTRY_IMPORT	= 1;	// Import Directory
IMAGE_DIRECTORY_ENTRY_RESOURCE	= 2;	// Resource Directory
IMAGE_DIRECTORY_ENTRY_EXCEPTION	= 3;	// Exception Directory
IMAGE_DIRECTORY_ENTRY_SECURITY	= 4;	// Security Directory
IMAGE_DIRECTORY_ENTRY_BASERELOC	= 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_DELAY_IMPORT	= 13;	// Delayed Imports
IMAGE_DIRECTORY_COM_DESCRIPTOR	= 14;	// COM Runtime descriptor
IMAGE_DIRECTORY_RESERVED	= 15;	// ...

Windows

5.4.2 Sections

Sections

- *Eigentlicher Inhalt* der Datei
 - Code, Daten, Ressourcen
 - Debuginfos, ...
- Im Header:
Metadaten in *Section Table*
- Auf Header folgen
aneinandergereihten Sections



Section Table

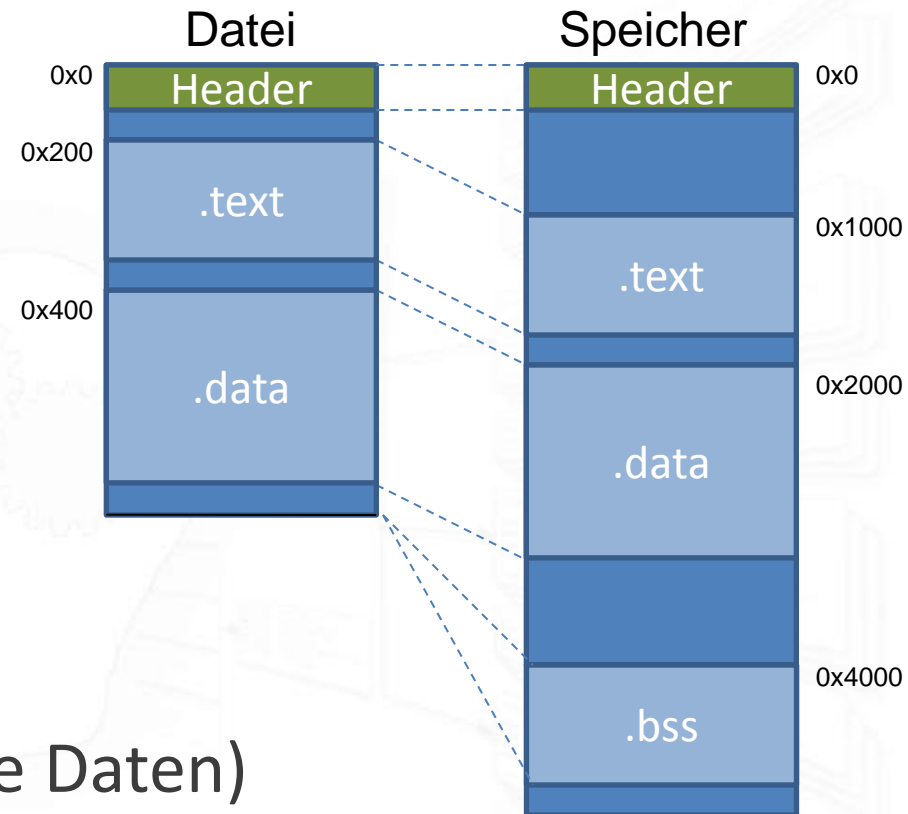
- 1 Eintrag pro vorhandener Section
 - Name, Größe und Lage (in Datei und im Speicher)
- Section-Name ist frei wählbar, jedoch häufig:
 - `.text` Code
 - `.data` Daten
 - `.bss` uninitialisierte Daten
 - `.rdata` readonly-Daten (Konstanten)
 - `.rsrc` Ressourcen
 - `.idata` Import Directory

IMAGE_SECTION_HEADER

+0x000	Name	: [8] Uchar	
+0x008	VirtualSize	: Uint4B	; Grösse 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?

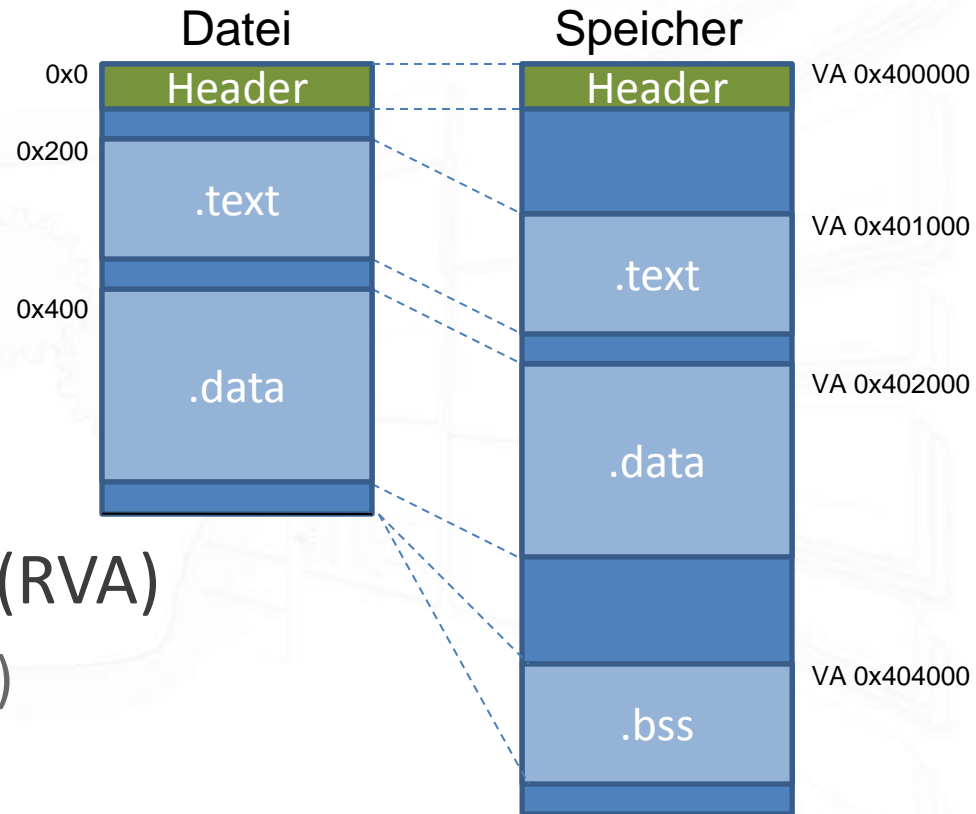
Alignment

- Datei *Alignment*
 - Häufig 0x200 Bytes
 - Segmente starten bei 0x200, 0x400, 0x600, ...
- Speicher *Alignment*
 - Häufig 1 KB (1 Page)
 - Segmente starten bei 0KB, 1KB, 2KB, ...
- .bss-Section (uninitialisierte Daten)
 - 0 Bytes in Datei
 - 1KB (0xF00 Bytes + 0x100 Alignment) im Speicher



Offset, VA, RVA

- Betrachte *.text* Section
- Raw file offset
 - 0x200
- Relative Virtual Address (RVA)
 - 0x1000 (= memory offset)
- Virtual Address (VA)
 - 0x401000



Windows

5.4.3 Export Directory

Export Directory

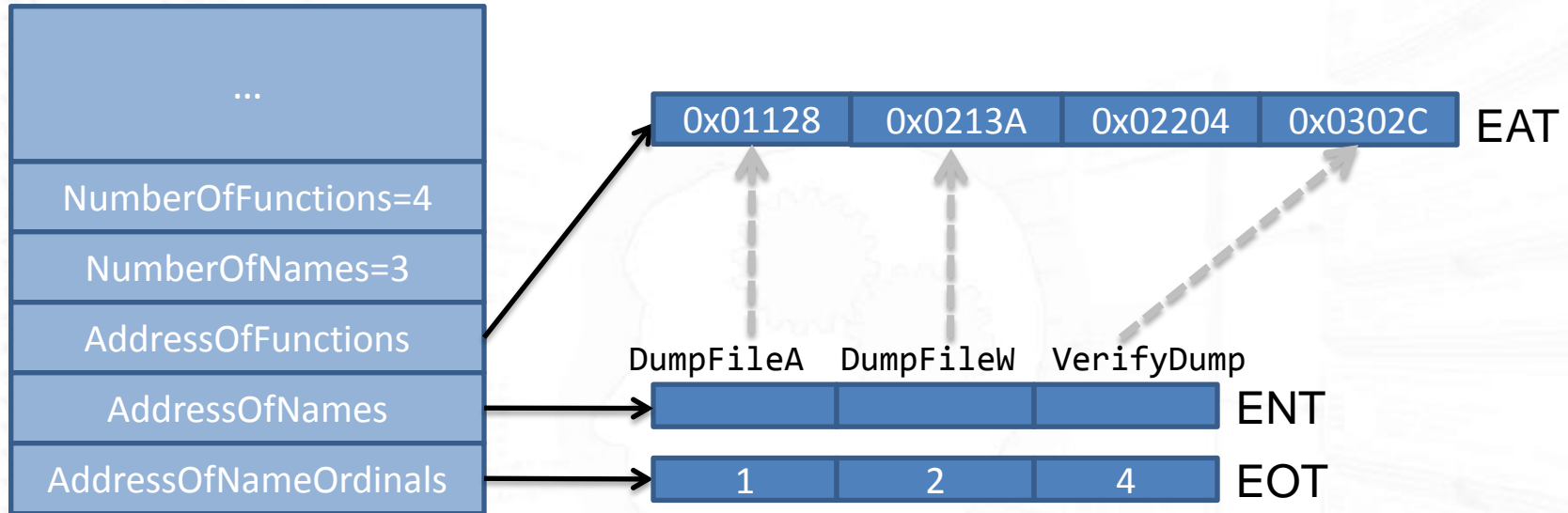
- Erinnerung: Bibliotheken exportieren
 - Symbole (=Funktionen und/oder Daten)
 - Zugriff via Ordinalzahl (16Bit-Wert)
 - Kein Lookup nötig → schneller EAT-Zugriff
 - Aber Ordinale können sich bei Update ändern
 - Oder Namen
- Veröffentlichung der Symbole via *Export Directory*
 - Anzahl der exportierten Symbole
 - Namen der Symbole (falls bekannt)
 - Mapping zwischen Symbolnamen und Ordinalzahlen

Export Directory

- EAT - Export Address Table („AddressOfFunctions“)
 - 1 Eintrag für jedes exportierte Symbol
 - Anzahl der Einträge: „NumberOfFunctions“
- ENT - Export Name Table („AddressOfNames“)
 - 1 Eintrag für jeden exportierten Namen
 - Anzahl der Einträge: „NumberOfNames“
- EOT - Export Ordinal Table („AddressOfNameOrdinals“)
 - 1 Eintrag für jedem ENT-Eintrag
- Alle Tabellen-Einträge sind RVAs

EAT, EOT, ENT

Export Directory



- ENT kann *mehr* oder *weniger* Einträge als EAT haben
 - Aliase: mehrere Namen für eine Function
 - Export nur via Ordinal

IMAGE_EXPORT_DIRECTORY






























```
typedef struct _IMAGE_EXPORT_DIRECTORY
{
    DWORD    Characteristics;
    DWORD    TimeDateStamp;
    WORD     MajorVersion;
    WORD     MinorVersion;
    DWORD    Name;                    // interner Name der DLL
    DWORD    Base;                   // gewünschte 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;
```


Auffinden von Exporten

- Windows-Loader lädt DLL in den Speicher
- Auffinden eines Exports via „GetProcAddress()“
 - API-Funktion

```
FARPROC GetProcAddress( HMODULE hModule, LPCSTR lpProcName );
```
 - lpProcName ist String
 - Suche in ENT, dann EOT, dann EAT
 - lpProcName kann auch ein Ordinal sein
 - HIGH(WORD) = 0, LOW(WORD) = Ordinal
 - Suche dann direkt über EAT
- In EAT steht RVA, ergibt mit DLL-Ladeadresse die VA

EAT – kernel32.dll

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