Native Windows API Programmierung

Johannes Rudolph

27. September 2007

Albert-Ludwigs-Universität Freiburg
Lehrstuhl für Kommunikationssysteme
Prof. Schneider
Dirk von Suchodoletz

Zusammenfassung

In Schulungsumgebungen wird häufig eine Virtualisierungslösung eingesetzt, um identische Arbeitsplätze zu realisieren. Das Betriebssystem wird dabei aus einem Festplattenabbild gestartet. Dabei zeigt es sich, dass es nicht möglich ist, den Computernamen eines Windowsrechners noch nach dem Start des Computers zu ändern. Für diese Arbeit wurde ein „Bootprogramm“ entwickelt, das auf der Windows „Native API“ basiert. Dieses liest den Computernamen schon während des Startens von Windows aus einer Datei aus und schreibt ihn in die Registrierung. So ist gewährleistet, dass jeder Arbeitsplatz trotz ansonsten identischer Ausstattung eine einmalige Identität zugewiesen bekommen kann.

Inhaltsverzeichnis

1 Einleitung

Klone ... Für Schulungen am Computer ist es wünschenswert, im Schulungsraum identische Computerarbeitsplätze zur Verfügung stellen zu können. Gleiche Hardware und gleiche Software an jedem Arbeitsplatz, unzerstörbar eingerichtet, ist der Traum eines jeden Administrators.

Eine gute Lösung ist es, auf jedem Rechner identische Hardware zu emulieren und diese Hardware auf allen Rechnern mit dem gleichen Festplattenabbild zu versorgen.

Solch eine Virtualisierungslösung hat ihren Anspruch gleich zweifach erfüllt: Unabhängig von der Gastgeberhardware wird immer der gleiche Computer simuliert und auch die Software ist an allen Arbeitsplätzen gleich, da sie von einem identischen Abbild auf einem Netzwerklaufwerk geladen wird.

... oder doch eher ungleiche Zwillinge? Ist endlich eine gemeinsame Grundlage geschaffen, ist nun doch wieder ein Schritt zurück nötig: Einzelne Arbeitsplätze sollen gezielt über das Netzwerk angesprochen werden können. Wie kann das realisiert werden? Oder das Betriebssystem soll auf einen bestimmten Nutzer zugeschnitten gestartet werden. Viele Möglichkeiten sind denkbar, welche Optionen für einen bestimmten Arbeitsplatz speziell angepasst werden sollen.

Das Betriebssystem muss also um die Möglichkeit erweitert werden, schon während dem Start auf das Gastgeberbetriebssystem zuzugreifen, die gesetzten Parameter abzurufen und die Anpassung dann durchführen zu können.

Ein pathologischer Fall: das Setzen des Computernamens unter Windows Der Parameter der dieses Projekt nötig gemacht hat, ist der Computernamen. Dieser ist in einem lokalen Windowsnetzwerk die NetBIOS Adresse eines Computers. Als solcher muss dieser Name feststehen, bevor Windows das Netzwerk initialisiert und den Computernamen im Netzwerk verkündet. Eine Folge dessen: Ändert man den Computernamen unter Windows, ist normalerweise ein Neustart unausweichlich.

Der Computernamen wird in der Registrierung gespeichert. Gesucht wird also eine Schnittstelle zum Gastgeber und ein Programm, das

1. Früh genug im Bootvorgang von Windows ausgeführt wird
2. Zugriff auf die Schnittstelle hat
3. Zugriff auf die Registrierung hat

Andere Ansätze Informationen, ob und wie der Computername eines Windowsrechners geändert werden kann, ohne neuzustarten, finden sich auch im Internet nur sehr schwer. Ein Programm, welches von sich behauptet, so etwas zu verwirklichen ist „baptize“ von Jo Berlakovich. Es ist von 1997 und funktioniert nicht mehr. Es ist nur als Binärprogramm verfügbar. Ein Blick in diese Datei, lässt vermuten, dass das Programm mit den Windowsdiensten LanmanServer/-Workstation und Messenger operiert („Arbeitsstationsdienst“/„Server“„Nachrichtendienst“ in der deutschen Windowsversion). Eventuell lässt das Ändern des Computernamens auch verwirklichen, indem

- diese Dienste gestoppt werden
- der Computername in der Registrierung geändert wird
- die Dienste wieder gestartet werden

Diesen Lösungsansatz habe ich allerdings nicht weiter verfolgt, weil es nicht dem Thema dieser Arbeit entsprach (wenn auch dem gestellten Problem).

2 Analyse

In diesem Abschnitt möchte ich einen Überblick über die Grundlagen des Windowsbetriebssystems geben. Nach einer kurzen Einführung in Begriffe und Datenstrukturen in Abschnitt 2.1, werde ich danach die einzelnen Schichten von Programmierschnittstellen näher beleuchten (2.2). Abgeschlossen werden die Betrachtungen durch eine kurze Beschreibung des Windowsbootvorganges im Abschnitt 2.3.

2.1 Grundlagen

2.1.1 Objektmanager

Windows verwaltet die meisten Resourcen des Kernels als Objekte. Diese Struktur wurde eingeführt, um Folgendes zu erreichen[6, S.125]:

- Eine einheitliche (Zugriffs-)Struktur bieten zu können
- Sicherheitsrelevante Details zentral implementieren zu können
- Eine zentrale Buchhaltung der Systemresourcen zu ermöglichen und den Resourcenverbrauch von Prozessen messen und beschränken zu können
- Einen allgemeingültigen Namensraum zu schaffen
- Einheitliche Regeln für den Lebenszyklus von Objekten und den zugehörigen Resourcen implementieren zu können (eine Art Garbage-Collection).

Objekte gibt es auf jeder Stufe des Betriebssystems. Grundlage sind Kernelobjekte, die vom „Executive System“ verwendet und als „Executive Objects“ verwaltet werden. Auf den höheren Ebenen obliegt dem Objektmanager die Verwaltung der Objekte. Er kennt Name, Typ und Anzahl der ausgegebenen Referenzen und „Handles“.

Im user-mode werden Objekte durch eben diese Handles repräsentiert. Einem Handle wurden bei der Erstellung bestimmte Rechte verliehen. Ein Handle ist immer nur für den Prozess gültig, der es erzeugt hat. Ausnahmen sind erbbare Handles, die an Kindprozesse weitergegeben werden können. Hat man die entsprechenden Rechte, kann man ein Handle auch Duplizieren, d.h. ein neues Handle erstellen, das für einen anderen Prozess gilt.

Es gibt Funktionen, die für alle Handles gelten, beispielsweise `DuplicateHandle` zum Duplizieren eines Handles oder `ZwClose` bzw. `CloseHandle` zum Schließen eines Handles.

Es gibt einen globalen Namensraum (siehe auch S. 17), in dem alle benannten Objekte (und auch alle bekannten Typen) des Systems eingetragen sind. Dieser ist eine Baumstruktur. Ein Handle auf ein benanntes Objekt kann häufig mit `ZwCreateFile` bzw. mit `CreateFile(Ex)` erstellt werden.

Alle Dateien und auch die Registrierung sind Teil dieses globalen Namensraumes. Auch alle Geräte sind hier eingetragen. Der Objektnamensraum entspricht also in etwa VFS in Linuxsystemen. (Allerdings mit dem großen Unterschied, dass Windows versucht, dies so gut wie möglich zu verbergen.)

Bei der Erläuterung der Dateinamen(S. 17) gibt es noch einige weitere Details zu diesem Thema.

2.1.2 Registrierung

Die Registrierung ist Windows zentraler Konfigurationsspeicher. Wie ein Dateisystem ist sie als Baumstruktur aufgebaut. Grundsätzlich besteht sie aus zwei Elementen:

- Schlüssel
- Werte

Schlüssel stellen die Knoten des Baumes dar. Jeder Schlüssel hat einen Namen, einen Elternschlüssel und eine beliebige Anzahl von Kindschlüsseln und Werten. Jeder Schlüssel hat eine Access Control List, bzw. erbt diese von seinem Elternschlüssel. Die ACL gibt an, welche Benutzerkonten welchen Zugriff auf diesen Schlüssel und seine Kinder hat. Werte hingegen enthalten die eigentlichen Daten. Werte haben einen Namen, einen Typ und einen binären Datensatz. Häufigste Typen sind `REG_SZ`, ein null-terminierter String und `REG_DWORD` ein Doppelwort. Mit dem Registrierungseditor regedit.exe kann die Registrierung bearbeitet werden.

Es ist möglich einen Schlüssel als symbolischen Link zu einem anderen Schlüssel anzulegen.
Es sind gewöhnlicherweise 5 Wurzelschlüssel in der Registrierung angelegt:

- `HKEY_LOCAL_MACHINE`: Dieser Schlüssel enthält alle Daten, die für den ganzen Rechner gelten. Es enthält alle Systemeinstellungen, gefundene Geräte, Treiber, die geladen werden sollen usw.
- `HKEY_USERS`: Enthält für jeden Benutzer (für den es gerade benötigt wird) einen Unterschlüssel mit Benutzerspezifischen Einstellungen.
- `HKEY_CURRENT_USER` ist ein symbolischer Link zu den Einstellungen des momentanen Benutzers
- `HKEY_CURRENT_CONFIG` enthält einige aktuelle Einstellungen, die allerdings nicht persistent sind.
- `HKEY_CLASSES_ROOT` ist ein symbolischer Link zu HKLM/Software/Classes und enthält Daten wie mit einzelnen Dateitypen umgegangen werden soll und die Deklaration von COM-Klassen und -Interfaces (CLSID usw.)

Die Registrierung existiert hauptsächlich im Speicher. Einen serialisierbaren Zweig(Schlüssel) der Registrierung, der als Binärdatei auf der Festplatte liegt, nennt man Hive. Als Binärdatei liegt die ganze Registrierung verteilt auf der Festplatte vor. Während des Bootvorganges werden die Wurzelknoten angelegt und nach und nach die einzelnen Dateien in die Registrierung (im Speicher) eingeblendet. Der Benutzerregistrierungshive wird bei Bedarf aus seiner eigenen Datei im jeweiligen Benutzerverzeichnis geladen.

2.2 Windows APIs

Es gibt Programmierschnittstellen auf allen Ebenen des Betriebssystems. In der Abbildung 1 sind diese schematisch dargestellt.

Hardware & HAL Grundlage des Rechners ist die Hardware. Die erste Softwareschicht ist das Hardware Abstraction Layer(HAL), diese Schicht stellt grundlegene Funktionen zur Verfügung wie das Lesen und Schreiben von Hardwareports, den Zugriff auf Interrupt-Controller und sofern nötig die Unterstützung von Multiprozessorsystemen. Für verschiedene Architekturen gibt es verschiedene HALs. HALs gibt es beispielsweise für Computer mit oder ohne ACPI/APIC und für Uni- oder Multiprozessorsysteme jeweils. Indem die Details einer Computerarchitektur so vor dem Rest des Betriebssystems versteckt werden, kann eine größere Kompabilität erreicht werden.

Kernel & 'Executive' Die nächste Schicht ist der Kernel. Unter diesem Begriff sind alle Funktionen zusammengefasst, die das Betriebssystem bereithält. Der (Windows-)Kernel wird meist noch differenziert in eigentlichen Kernel und 'Executive'. Der Kernel ist auch abhängig von der Architektur und beinhaltet fundamentale Funktionen, darunter Threadscheduling und -synchonisierung genauso wie Routinen zur Interrupt- und Ausnahmenbehandlung.

Die Ausführungsschicht („Executive“) hingegen verwaltet viele Resourcen und Funktionen des Computers als Objekte. Zugriff auf die Registrierung, Prozesse und Threads, den Speicher, Sicherheitsfunktionen und den Zugriff auf die Treiber werden hier geregelt.

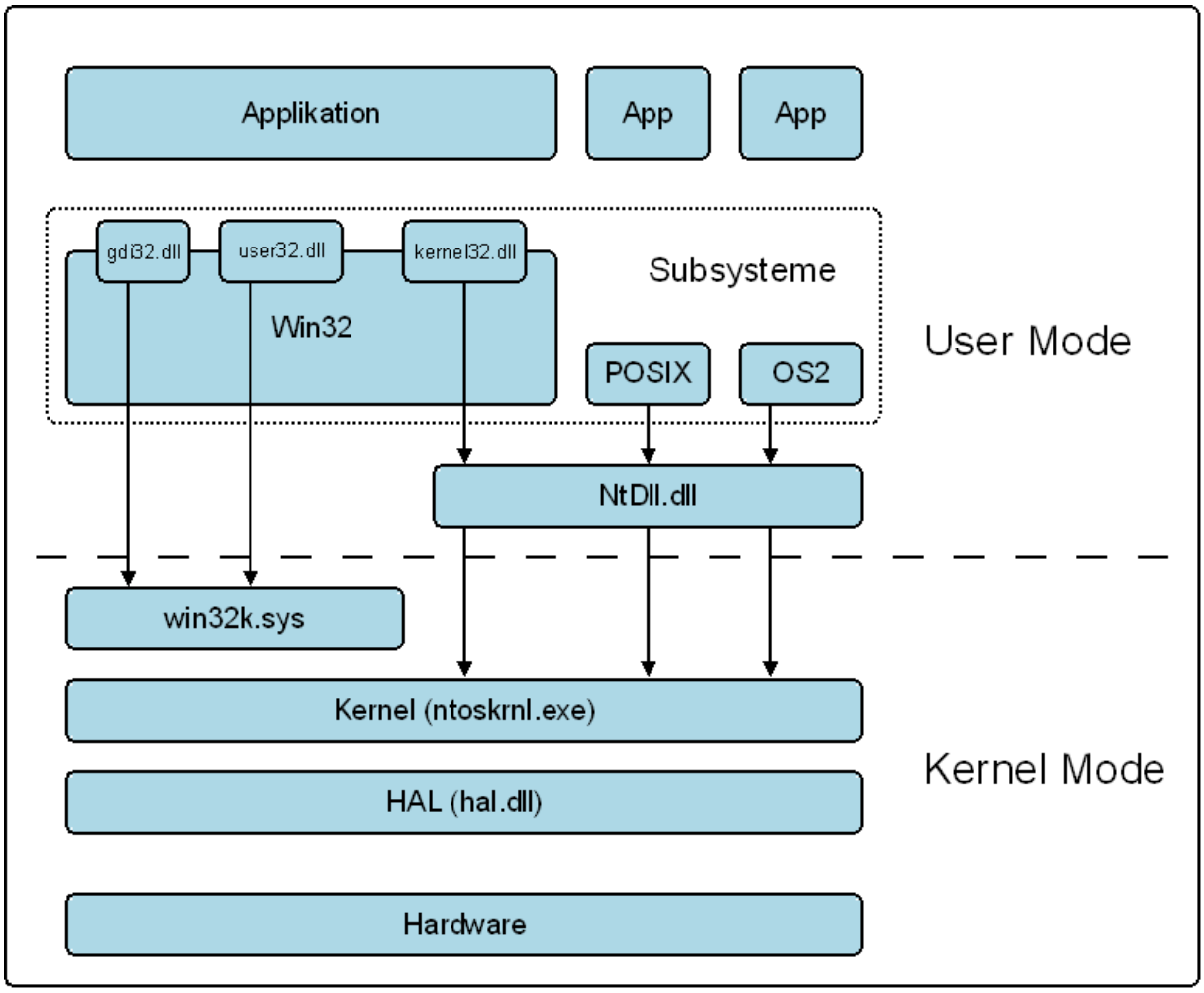

Abbildung 1: Windows API Schichten

Kernelmode, Usermode Moderne Prozessoren verfügen über mehrere Ausführungsmodi. In der x86-Architektur unterscheidet man gewöhnlich zwischen real-mode und protected-mode. Im real-mode hat das Programm das gerade läuft Zugriff auf alle Systemresourcen. Im protected-mode gibt es mehrere Sicherheitsstufen nämlich die 'Ringe' null bis drei („privilege levels"). Ein Programm wird mit steigender Ringzahl restriktiver behandelt. In Ring 0 ablaufend hat ein Programm keinerlei Beschränkungen, während eine Routine in Ring 3 beispielsweise nicht mehr auf I/O-Ports direkt zugreifen kann. Auch wird in Ring 3 der Zugriff auf den Speicher eingeschränkt.

Diese Einteilung in die verschiedene Zugriffsarten verhindert, dass Programme in einem höheren Ring grundlegende Funktionen des Rechners behindern oder zerstören können, und kann somit zu einer höheren Stabilität des Betriebssystems als Ganzem verhelfen. Viele Betriebssystem verwenden nicht alle diese Modi. In Windows wird nur zwischen „user mode" (Ring 3) und „kernel mode" (Ring 0) unterschieden.[1]

Unterbrechungen im Arbeitsablauf Wenn eine Benutzeranwendung (im „user mode") nun eine Funktion benötigt, beispielsweise um auf die Registrierung zuzugreifen, und diese ist im Kernel

[1] Laut [6, Notiz auf Seite 16] geht diese Zweiteilung auf den Umstand zurück, dass Windows in der Vergangenheit kompatibel zu Architekturen sein sollte, die eben nur diese zwei Ebenen bereitstellten. (Compaq Alpha, Silicon Graphics MIPS)

implementiert, muss zu irgendeinem Zeitpunkt in den „kernel mode“ gewechselt werden. Für diesen Wechsel gibt es je nach Architektur verschiedene Möglichkeiten.

Ein Interrupt ist eine davon. Der Interrupt ist eine Unterbrechung des aktuellen Programmes und kann auf zwei Arten ausgelöst werden: Entweder durch ein Signal der Hardware oder softwareseitig (x86 `int` Befehl).

An welcher Stelle soll der Prozessor nun weiterarbeiten? Dazu gibt es im Prozessor die Interrupt Dispatch Table(IDT). Sie enthält für jeden Interrupt die Adresse der Funktion, die den Interrupt verarbeiten muss. Für Javaprogrammierer: Bei der Hardware kann für auftretende Ereignisse - signalisiert durch Interrupts - ein „Listener“ registriert werden, der sich dann um dieses Ereignis kümmern muss.

Zurück zum Aufruf einer Kernelfunktion bedeutet das: Die grundlegende Methode, einen Systemaufruf zu tätigen, ist es, einen Interrupt auszulösen. Dieses Verfahren ist allen bekannten Betriebssystemen gemein. DOS, Windows und auch das BIOS, als grundlegenstes „Betriebssystem“, verwenden diese Methode.

Das „System Service Dispatching“ Eine Möglichkeit, verschiedene Systemfunktionen abzubilden, wäre es nun, für jede Funktion einen Interrupt zu reservieren. Da die IDT aber nur eine begrenzte Größe hat (x86: 256 Adressen) und viele Einträge für Hardwareinterrupts reserviert sind, wird ein zweistufiges Dispatching angewandt.

Soll eine Kernelfunktion aufgerufen werden, wird ein Interrupt ausgelöst und die Funktion als Nummer in einem Register mitgegeben. Der Kernel muss den Interrupt behandeln; er hat wiederum eine Tabelle, in welcher für jede Funktion die Adresse der Implementierung steht - die Nummer im Register entspricht der Stelle in der Tabelle.[2]

Allerdings: Bei der Entwicklung moderner Prozessoren hat man den Systemaufruf optimiert und ihm einen eigenen Maschinenbefehl spendiert (Intel ab Pentium II: sysenter/sysexit, AMD ab K6: syscall/sysret). Die Dispatchtabelle wird dann direkt in einem zugehörigen Prozessorregister gespeichert und ist somit schneller verfügbar.

Eine ausführliche Beschreibung dieser Vorgänge findet sich in [6, S. 119ff].

NtDll.dll Natürlich muss nicht jeder Kernelfunktionsaufruf manuell und in Assembler erfolgen. Es gibt eine Bibliothek, die für jede Schnittstelle des Kernels eine Funktion exportiert: NtDll.dll. Diese Funktion sieht beispielsweise für NtReadFile folgendermaßen aus (abgeschrieben bei Microsoft Windows Internals):

```
ntdll!NtReadFile
mov eax,0xb7
mov edx,0x7ffe0300
call edx
ret 0x24
```

Im Register eax wird die Nummer der Funktion gespeichert(für NtReadFile bei Windows 2000: 0xb7, siehe [7]) und dann die Funktion an der Stelle 0x7ffe0300 aufgerufen. Diese Funktion wird beim Starten des Computers angelegt und enthält je nach Prozessortyp den Code zum Ausführen des kernel-mode-Aufrufs. Beispielhaft für den Pentium M:

```
0x7ffe0300 SharedUserData!SystemCallStub
mov edx,esp
sysenter
ret
```

Der aktuelle Stackpointer wird im edx-Register gespeichert. Die Existenz von NtDll.dll hat noch einen anderen Grund: Die Funktionsnummern können zwischen den einzelnen Windowsversionen variieren. Programme verwenden zur Referenz auf eine Kernelfunktion nie die Nummer, sondern immer den von NtDll.dll exportierten Funktionnamen und bleiben so über Windowsversionen hinweg kompatibel.

[2] Eine beliebte Methode, bestimmte Systemaufrufe abzufangen, ist es, die jeweilige Adresse in der „System Service Dispatch“ Tabelle zu verändern.

Windows Subsysteme Die ursprüngliche Idee bei der Entwicklung von Windows (NT) war, die Kompatibilität zu verschiedenen anderen Betriebssystemen zu gewährleisten. Aus dieser Idee entstand das Konzept der „Environmental Subsystems“. Einer Anwendung, die bspw. für OS/2 programmiert war, sollte eine Umgebung bereitgestellt werden, die der von OS/2 entsprach. Umgebung ist hier die Schnittstelle zum Betriebssystem und die Semantik der einzelnen Betriebssystemfunktionen.

Ein Subsystem ist also zweierlei: Eine Laufzeitumgebung und eine definierte API zu einer Teilmenge der Betriebssystemfunktionen. Windows enthält bzw. enthielt hauptsächlich die folgenden drei Subsysteme: POSIX, OS/2 und Win32.

Win32 ist das „einheimische“ Subsystem von Windows. Es ist aufgebaut aus einem user-mode Teil csrss.exe und einem kernel-mode Treiber win32k.sys. Der user-mode Teil enthält die Funktionen für Konsolenfenster, das Erzeugen von Prozessen und Threads und die Unterstützung für alte DOS-Programme. Der kernel-mode Code enthält den Fenstermanager und das „Graphics Device Interface“(GDI), die Schnittstelle, um Linien, Text usw. auf den Bildschirm zu zeichnen.

Dazu gehören auch einige Bibliotheken wie advapi32.dll, user32.dll, gdi32.dll, kernel32.dll, die letztendlich die subsystem-spezifischen Einsprungpunkte für Betriebssystemfunktionen beinhalten, bzw. die Implementation für Funktionen, die keine Kernelunterstützung benötigen. Wird eine Funktion aus kernel32.dll aufgerufen, bspw. ReadFile, wird in vielen Fällen die Gültigkeit der Parameter überprüft und dann die entsprechende Funktion in NtDll aufgerufen (hier: NtReadFile). Anders sind hingegen Aufrufe, die Fenstermanager(user32.dll) und GDI betreffen. Die entsprechenden Funktionen, im Großen und Ganzen Grafikfunktionen, sind hauptsächlich der Geschwindigkeit wegen im kernel-mode implementiert(win32k.sys). Der Aufruf dieser Funktionen im kernel-mode erfolgt über eine zweite Dispatch-Tabelle, die Methode entspricht im Wesentlichen der vorher beschriebenen für gewöhnliche Systemaufrufe.

Ausführbare Dateien im Windows-Ecosystem haben das .exe-Format. Dieses definiert im Header ein Feld, das angibt, mit welchem Subsystem ein Programm ausgeführt werden soll. Der Microsoft Linker versteht den Parameter „/SUBSYSTEM“. Mit diesem kann beim Linken das Subsystem für ein Programm angegeben werden.

Anwendungen ohne Zuhause: Native Applikationen Es gibt Programme - und das wird später noch wichtig -, die sollen keinem speziellen Subsystem zugeordnet werden. Für ein Bootprogramm beispielsweise ist dies vorgeschrieben, ganz einfach, weil noch kein Subsystem verfügbar ist. Das gleiche gilt auch für Gerätetreiber(die allerdings sowieso im kernel-mode ausgeführt werden). Solche Programme nennt man native Applikationen. Die einzige Schnittstelle, die ihnen zum Kernel bleibt, ist Ntdll.dll. Selbstverständlich sind auch alle Subsysteme selbst native Programme.

2.3 Boot me up, Billy: Der Bootvorgang

Der Windowsbootvorgang besteht aus mehreren Phasen. In jeder Phase hat ein anderer Teil des Betriebsystem die Kontrolle über den Ablauf. Eine grobe Einteilung ist die folgende:

- Start des Computers
- BIOS startet
- BIOS lädt den Bootloader auf dem Bootmedium
- 1. Teil des Windowsbootloader im Bootsektor
- 2. Teil des Windowsbootloader: Ntldr, lädt den Kernel ntoskrnl.exe
- Phase 0:ntoskrnl.exe, der Kernel übernimmt die Kontrolle und initialisiert das System mit ausgeschalteten Interrupts
- Phase 1: die Interrupts werden angeschaltet, der graphische Bootbildschirm wird angezeigt, Treiber werden initialisiert

- Erster user-mode Prozess smss.exe wird gestartet
- smss.exe: Subsysteme werden initialisiert
- Winlogon wird gestartet und wartet auf Benutzeranmeldungen

Ich werde im Folgenden die einzelnen Schritte näher beleuchten. Im Wesentlich ist dies eine Zusammenfassung von [6, Kapitel 5].

BIOS Das **B**asic **I**nput **O**utput **S**ystem ist das Programm das als erstes ausgeführt wird, wenn der Rechner eingeschaltet wird. Beim Start des Rechners kann noch nicht auf die Laufwerke zugegriffen werden. Deswegen liegt es unveränderlich auf einem Chip auf der Hauptplatine. Es kennt Standardschnittstellen zu den Geräten, kann andere Peripherie, die zum Starten benötigt wird rudimentär initialisieren und stellt selber eine Programmierschnittstelle zur Verfügung, derer sich die Bootloader bedienen können (oder auch das Betriebssystem wenn es DOS heißt).

Nachdem das BIOS die Laufwerke des Rechners entdeckt hat, kann es dem Benutzer ein Menu zur Auswahl eines Laufwerks anzeigen oder selbst ein Laufwerk aussuchen, welches den Code zum Booten enthält. Der Bootcode befindet sich im Allgemeinen im Bootsektor. Das ist bei Laufwerken meist der Sektor 0, bei CDROMs ist das Verfahren ein bisschen komplizierter. Nachdem das BIOS seine Initialisierungsarbeit abgeschlossen hat, wird der Bootloader aufgerufen.

Ntldr: Der Windows Bootloader Der Windowsbootloader kommt in zweierlei Portionen daher. Die erste Portion ist im Bootsektor abgelegt und hat deswegen eine stark begrenzte Größe. Dieser erste Teil des Loaders muss den zweiten Teil auf der Festplatte finden können. Allerdings: Der Platz ist knapp. Aus diesem Grund ist im Bootsektor immer nur der Code enthalten, der benötigt wird, um ein *bestimmtes Dateisystem* lesen zu können, nämlich das, mit dem die Festplatte eingerichtet wurde.

Die zweite Portion liegt als Ntldr im Hauptverzeichnis des Laufwerks. Vom Bootsektor aufgespürt übernimmt Ntldr fürs erste die Kontrolle. Ntldr wechselt zunächst erst mal vom real- in den protected-mode. Ntldr enthält wie der Bootsektor nur Code, um das Dateisystem lesen zu können, auf dem es selber liegt (das allerdings in einer erstaunlich reifen Form: es kann Dateien in Unterverzeichnissen lesen). Um dies allerdings durchführen zu können, ist allerdings ein ständiger Wechsel in den real-mode nötig, um das BIOS aufrufen zu können.

Ntldr ist zuständig für die Anzeige des Bootmenus, für das Booten von DOS-Bootsektorabbildern (bootsect.doc) und für das Aufwecken einer Windowsinstanz aus dem Winterschlaf. Dazu muss Ntldr die Sprache der boot.ini verstehen, die Datei, die angibt, welche Bootoptionen sich für den Benutzer ergeben.

Hat sich der Benutzer schließlich für eine Option entschieden, wird ntdetect.com ausgeführt. Ntdetect ist ein kleines 16-bit real-mode Programm, das als erstes seiner Art beginnt, auszuspähen, was der Rechner alles zu bieten hat. Dazu gehören Datum und Zeit aus dem CMOS, die verfügbaren Busse, Anzahl, Größe und Sorte von Festplatten, Nagetiereingabegeräten, Parallelports und Videogeräten. Später werden diese Informationen in der Registrierung unter HKLM\HARDWARE\DESCRIPTION gespeichert.

Jetzt geht es richtig los. Damit die Windowsbenutzer das auch bemerken, gibt es die umgekippte Leiter, die auf dem Boden des Bildschirms liegt. Von links nach rechts wird darauf ersichtlich, welchen Fortschritt die Programmierer von Windows gemacht haben:

- Finden des Kernels(meist ntoskrnl.exe) und der HAL(meist hal.dll)
- Einlesen des SYSTEM-Baums der Registrierung
- Auffinden der Bootgerätetreiber darin
- Laden, nicht aber Initialisierung der Dateisystemtreiber, denn die werden als erstes benötigt
- Laden, nicht aber Initialisierung der Bootgerätetreiber

- Vorbereiten der Prozessorregister auf das was sie jetzt erwartet: die Ausführung des Kernels (ntoskrnl.exe)

Das ist das Ende des Ntldr.

One to rule them all: ntoskrnl.exe Nun übernimmt der NT-Kernel ntoskrnl.exe seine Herrschaft. Der Aufbau dieser hat zwei Phasen. Viele Teilsysteme des Kernels können in beiden Phasen ihre Initialisierung vornehmen.

Hauptmerkmal der Phase 0 ist die Ruhe, keine Unterbrechungen, Interrupts sind ausgeschaltet. Das hat den einfachen Grund, das zu diesem Zeitpunkt noch gar nicht sicher ist, wie man mit Interrupts umgehen soll. In Phase 0 werden Datenstrukturen eingerichtet, für jeden Prozessor werden die Interruptcontroller vorbereitet und der „Clock Timer Interrupt“ eingestellt, der später für die Prozessorzeitbuchhaltung nötig ist. Danach werden noch einige grundlegende Handlangerdienste initialisiert: Speichermanager, Objektmanager und Security reference monitor werden eingerichtet, der Prozessmanager darf die ersten Prozesse managen, nämlich den „Idle“-process und den System-Prozess. Der Systemprozess erhält den ersten Thread: die Routine Phase1Initialization. Was dieser macht, wird erst im übernächsten Abschnitt verraten, denn momentan sind die Interrupts noch nicht aktiviert und damit kann auch der Thread noch nicht loslegen. Abschließend darf der Plug-and-Play-Manager seine Phase 0 Vorbereitungen durchführen.

Nun wirds paradox: Phase 0 endet in der Idle-Loop. „Idle“, Englisch für träge, leer, außer Betrieb, arbeitslos, unbegründet, unwichtig, wertlos, untätig sein, Zeit verschwenden, auf minimaler Geschwindigkeit operieren. Die Idle-Schleife, elektronischer Däumchendreher des 21. Jahrhunderts, Prokrastinator des Informationszeitalters und das schon bevor das System hochgefahren ist?

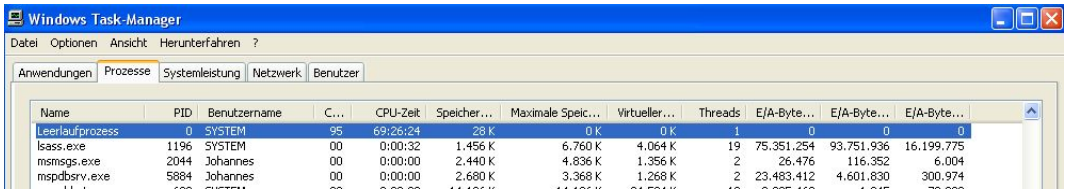

Abbildung 2: Der Leerlaufprozess

Während wir hart arbeiten, ist es der ausdrückliche Job des Leerlaufprozesses, nur dieses zu protokollieren: den Leerlauf. Und dann dafür zu sorgen, dass Arbeit gefunden wird.[3] Arbeit, das ist in diesem Fall der Phase1Initialization Thread. Einmal aufgeweckt verrichtet er fleißig sein Werk: HalInitSystem wird aufgerufen, um die Interrupts einzuschalten. Der Bootgrafiktreiber (bootvid.dll) wird gestartet, dieser zeigt den Windowsstartbildschirm an. Nun wird das ganze System initialisiert. Unter anderem wird die „System Dispatch Table“ eingerichtet, ntdll.dll in den Speicher eingeblendet, im Namensraum wird das \Registry Verzeichnis angelegt und die vorher erstellten Zweige HARDWARE und SYSTEM hineinkopiert. Einen großen Teil der Initialisierung gebührt dem I/O Manager, bei dessen Abschluss auch die DOS Laufwerksbuchstaben angelegt werden. Zu guter Letzt wird „smss.exe“ gestartet

SMSS: user-mode, wir kommen... Nein, kein neuer Serverdienst für Kurznachrichten, smss.exe ist das Session Manager Subsystem. Als Subsystem ist smss eine native Anwendung und der erste Prozess außerhalb der Kernel-Mode-Unterwelt: Willkommen im Benutzermodus. In [6, S. 269] liest sich das so:

> Smss is like any other user-mode process except for two differences: First, Windows considers Smss a trusted part of the operating system. Second, Smss is a native application. Because it's a trusted operating system component, Smss can perform actions

[3] Wie das genau funktioniert, kann man bei ReactOS ([2, KiIdleLoop]) nachlesen (mit Vorbehalt) oder im Kerneldebugger mitverfolgen.

> few other processes can perform, such as creating security tokens. Because it's a native application, Smss doesn't use Windows APIs - it uses only core executive APIs known collectively as *the Windows native API.*[6, Kursivdruck hinzugefügt]

Das erste eingeborene Programm. Und dazu noch eine Art Benutzermodussuperheld... Nun ist es so weit, die Rettung ist nahe. Smss ist der Verwalter der Sessions. Sessions: Man könnte sagen, mindestens für jeden Benutzer, der einen Computer gleichzeitig benutzt, muss eine Session existieren. Sie kapselt den momentanen Zustand einer Benutzerverbindung. Und Smss verwaltet diese. (Natürlich nur wenn die Terminal Services installiert sind. Ansonsten ist es ja auch hübsch, nur *eine* Session zu verwalten.)

Doch noch ist Windows nicht bereit: Smss ruft schließlich den Konfigurationsmanager auf. Dieser weiß, wo sich der Rest der Registrierungsdateien rumtreibt. Wenn er diese eingesammelt hat und in die Registrierung eingeblendet, schreibt er deren Pfad in den HKLM/SYSTEM/CurrentControlSet/Control/hivelist Schlüssel.[4]

Nun werden die letzten wichtigen Schritte getan: Es wird ein „LPC port" Objekt erstellt, \SmApiPort. Dies ist ähnlich wie ein TCP Port, es ist eine Routine, die Anfragen empfangen und bearbeiten kann. In diesem Fall, ein neues Subsystem zu laden oder eine neue Session zu erstellen. DOS-Geräte wie COM1 und LPT1 werden als symbolische Verknüpfungen im Namensraum eingerichtet. Das \Sessions-Verzeichnis wird angelegt. Und nun, zu diesem Zeitpunkt, wird das Bootprogramm ausgeführt, das in „HKLM/SYSTEM/CurrentControlSet/Control/Session Manager/BootExecute" eingetragen ist. Hierauf werden wir später zurückgreifen.

Weitere Aufgaben unseres Superheldens: Hinausgezögerte Lösch- und Umbennungsoperationen werden ausgeführt. „Known Dlls", bekannte und vielgefragte Dlls werden geladen und Zeiger darauf abgelegt. Auslagerungsdateien werden angelegt, wenn nötig. Die Systemumgebungsvariablen werden angelegt und schließlich wird der kernel-mode Teil des Win32 Subsystems geladen(win32k.sys). Dessen Initialisierung lässt die Grafikkarte schließlich in den konfigurierten Videomodus wechseln. Schließlich werden die eigentlichen Subsysteme geladen.

Am Ende des Smssstartmarathons wird der Anmeldeprozess(winlogon.exe) gestartet und LPC ports für Debuggingzwecke eingerichtet.

Winlogon und Gina: Das freundliche Empfangspersonal Ohne noch weiter auf die Details einzugehen: Winlogon verwaltet den Zugang zum System. Im Hintergrund allerdings nur, denn sein Gesicht ist Gina. Ms Gina Di El El. Man trifft sie jeden Tag beim Anmelden, weiß aber ihren Namen nicht. Darf ich vorstellen... Ms Gina. Akronymisch für Fräulein „Graphical Identification and Authentication". Gina ist die Komponente, die tatsächlich die Benutzerdaten abfragt, während Winlogon der Torwächter ist, der den Zugriff versperrt. Auf eine Anfrage von Gina hingegen...

Wie funktioniert Winlogon? Winlogon ist der einzige Prozess, der auf die Secure Attention Sequence(SAS) reagieren kann. Früher Klammergriff heute Secure Attention Sequence ist die Tastenkombination dennoch Ctrl-Alt-Delete geblieben. Empfängt Winlogon diesen Tastendruck, erzeugt es einen neuen Desktop und lässt Gina dort ihr Werk verrichten.

Es hat einen Grund, dass winlogon und Gina getrennt ihr Dasein fristen: Gina ist auswechselbar. Man stelle sich vor irgendein intelligenter Programmierer, hat eine Methode entwickelt, die Benutzeridentität anhand eines optischen Gewebestrukturtests des Benutzerpullovers ermitteln zu können. Dieser benötigt natürlich eine andere Benutzeroberfläche („Bitte setzen sie sich zentriert vor die Kamera.", usw.). Eingetragen in der Registrierung[5]: Vielen Dank und auf Wiedersehen Ms Gina, willkommen Opurs[6].

Ein weiterer wichtiger Aspekt von Winlogon ist, dass es das „Local security authentication subsystem"(lsass.exe) und den Service Control Manager(SCM, services.exe) startet. Der SCM lädt alle Services, die in der Registrierung eingetragen sind. Dazu gehören unter anderen auch der LanmanWorkstation und der LanmanServer Dienst. Es ist wohl einer dieser Services, die den Computernamen im Netzwerk bekannt machen.

[4] Hivelist = Bienenstockliste, Windows hat etwas Naturalistisches: Bienenstöcke, Winterschlaf, Eingeborene...
[5] HKLM/Software/Microsoft/WindowsNT/CurrentVersion/WinLogon/GinaDLL
[6] Optical Pullover User Recognition System

Abbildung 3: Gina in Aktion

Unser Programm muss vor diesen Services starten, um den Computernamen zu ändern. Die Möglichkeit, das zu erreichen, ist ein Bootprogramm, dessen Funktionsweise und Aufbau wir uns im nächsten Abschnitt widmen möchten.

3 Implementation

Halten wir uns noch einmal die Anforderungen vor Augen:
Das Programm muss

1. auf Daten vom Gastgeberbetriebssystem zugreifen können
2. Zugriff auf die Registrierung haben
3. rechtzeitig ausgeführt werden, dass wichtige Systemfunktionen wie das Netzwerk noch nicht initialisiert sind

Wurmloch zum Gastgeber Wie kann möglichst einfach auf Daten vom Gastgeber zugegriffen werden? Bei der Virtualisierung eines Rechners muss natürlich auch die Peripherie emuliert werden. Eingabegeräte Tastatur, Maus usw. werden meistens durch die tatsächliche Hardware realisiert. D.h. Eingaben werden vom Gastgeber direkt an den virtuellen Gastrechner weitergeleitet. Das gleiche kann auch für logische oder physikalische Laufwerke gelten. In unserem Fall ist das jedoch nicht wünschenswert, da ja bestimmte vorher hergestellte Festplattenabbilder verwendet werden.

Eine Möglichkeit wäre es nun, vor dem Starten des (virtuellen) Rechners, die im Abbild binär vorliegende Registrierung direkt zu verändern. Das Problem mit diesem Ansatz ist klar: Man müsste das jeweilige Dateisystem lesen und schreiben können (wie kompliziert das ist, wenn das Dateisystem nicht vollständig dokumentiert ist, kann man sich anhand der Geschichte der NTFS-Treiber für Linux vergegenwärtigen...) und das gleiche gilt für die binären Strukturen der Registrierung, die einem Dateisystem gleichkommen.[7]

Eine andere Lösung hat sich als sinnvoller erwiesen: Es wird vor dem Start des Gastrechners ein Diskettenabbild erstellt und dieses wird im Gast als Diskette zur Verfügung gestellt. Ein Programm muss beim Starten des Betriebssystem eine Datei auf dieser Diskette lesen und die erforderlichen Änderungen in der Registrierung tätigen.

[7] Der Zugriff auf die Registrierung wäre auch leichter zu bewerkstelligen: Es gibt eine API in Windows mit dieser können beliebige Registrierungsdateien an einer Stelle in der aktuellen Registrierung eingeblendet werden. Auf einem Windowsrechner ist es also möglich eine beliebige Registrierungsdatei zu bearbeiten. Der Nachteil allerdings: Damit das dynamisch funktioniert, müsste das Gastgebersystem ein Windowssystem sein.

Bootprogramm Wie vorher beschrieben, werden beim Starten des Computers Programme ausgeführt, die im Wert mit dem Registrierungspfad „HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\BootExecute“ aufgeführt sind. In der Abbildung wird dies angezeigt:

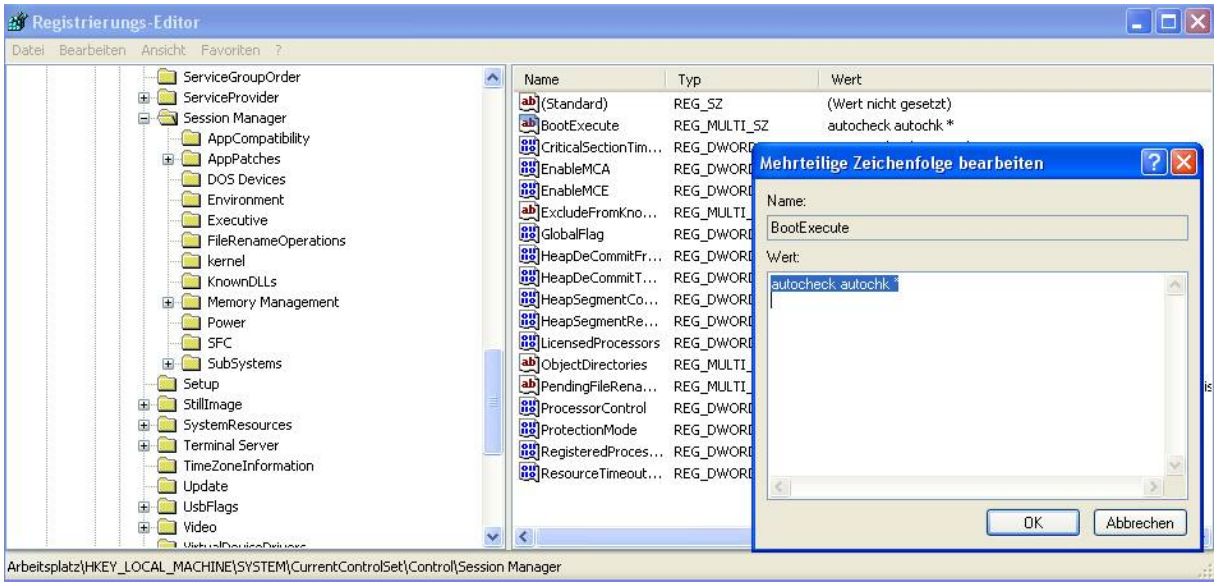

Abbildung 4: Editieren des BootExecute-Schlüssels

Jede Zeile entspricht einem Aufruf eines Programms. Es können Kommandozeilenparameter angegeben werden. Im Normalfall enthält dieser Registrierungswert nur einen Eintrag: autocheck. Dieses Programm überprüft bei jedem Start die Dateisystemintegrität der einzelnen Festplatten. Es wird als Bootprogramm ausgeführt, weil die Festplatten zu diesem Zeitpunkt kaum in Benutzung sind. Das gewährleistet, dass während der Ausführung von autocheck nicht andere Programme gleichzeitig auf die Festplatte zugreifen und so bei Bedarf die Integrität der Festplatte ungestört wiederhergestellt werden kann.

Das einzige uns bekannte veröffentlichte native Bootprogramm ist eine simple Beispielanwendung auf der Homepage von Sysinternals[1] (gehört inzwischen zu Microsoft).

Ein Bootprogramm kann nicht im Win32-Subsystem ausgeführt werden. Der Versuch ein solches zu starten, endet mit der Fehlermeldung "Die Anwendung <Name> kann nicht im Win32-Modus ausgeführt werden".

3.1 Aufbau

Bei Entwicklung eines Bootprogrammes ist es wichtig eine ordentliche Testumgebung zur Verfügung zu haben. Am besten ist es, man testet das Programm auch in einem virtuellen Rechner, das beschleunigt den Entwicklungszyklus. Außerdem verhindert es eine Windowsinstallation zu zerstören, indem man beispielsweise ein Bootprogramm definiert, das nicht endet und so den Computer „aufhängt“.

Trotz virtuellem Server ist ja dennoch für jeden Testlauf ein Neustart des virtuellen Systems notwendig. Ich habe mich deshalb entschieden, gleichzeitig eine Win32-Anwendung zu pflegen, mit welcher man die Abläufe (mit gewohntem Debugger) testen kann. Auch für Win32-Anwendungen stehen die Funktionen der NtDll.dll bereit. Die Win32-Varianten der Funktionen (in kernel32.dll) sind meistens einfacher zu verwenden und sind zuverlässiger, da die Parameter strenger überprüft werden, aber grundsätzlich stehen auch die NtDll-Methoden zur Verfügung (sie sind allerdings kaum dokumentiert).

Das Programm besteht im wesentlich aus drei Teilen:

computername.cpp enthält die Funktionen zum Laden und Lesen der Datei auf dem Diskettenlaufwerk und zum Schreiben des Registrierungsschlüssels

main.cpp Implementation des Parameterparsens und einer Shell, diese macht es möglich auch in der Bootprogrammumgebung interaktive Tests durchzuführen

io.cpp/h das Input/Output-Modul, es enthält einige Funktionen zum Ausgeben und Einlesen von der Konsole und Funktionen zur Speicherverwaltung

Es wurde darauf Wert gelegt, dass viel Code gleichzeitig in der Win32 und in der Nativen Umgebung verwendet werden kann. High-Level I/O-Funktionen werden in der Klasse IO definiert. Die Low-Level-Funktionen müssen jeweils von einer Unterklasse definiert werden. Diese Funktionen umfassen:

- malloc und free zur Speicherallozііerung auf dem Heap
- internalPrint zum Ausgeben von Text auf der Konsole
- getChar und handleCharEcho für das Einlesen von Text von der Tastatur

Im Folgenden werde ich mich auf die Details konzentrieren, die speziell für native Applikationen gelten und die bis jetzt noch kaum im Internet dokumentiert sind.

Exkursion: Strings ... Ein interessantes Thema ist, wie Strings dargestellt werden sollen und in welcher Form sie vom Betriebssystem verwendet werden. Eine grundsätzliche Unterscheidung besteht zwischen Stringkodierungen die ein Byte pro Zeichen benötigen und denen die zwei benötigen.

Windows benutzt im Kernel hauptsächlich UNICODE Strings und damit 16 Bits pro Zeichen. Die Win32 APIs hingegen stellen häufig UNICODE und Ascii Varianten für jede Funktion zur Verfügung. Die Ascii-Strings werden noch im user-mode in UNICODE Zeichenketten umgewandelt und als solche dann an die entsprechenden NtDll-Funktion weitergegeben. Die verschiedenen Funktionen können am Suffix „A“ und „W“ erkannt werden. Die Windows SDK Headerdateien verstecken diese Tatsache meistens, indem sie je nach gewählter UNICODE-Unterstützung per Präprozessordirektiven die richtige Variante auswählen.

Im Code tauchen folgende Stringarten auf:

- `char *`: normale einbytige nullterminierte Characterpointer
- `wchar_t *`: zweibytige nullterminierte Widecharpointer für UNICODE Strings
- `UNICODE_STRING`: Eine „verwaltete“ Struktur, die außer den UNICODE Stringdaten auch die Größe des Zeichenpuffers und auch die tatsächliche Länge der Zeichenkette *in Bytes* enthält. Die Windows Laufzeitbibliotheken stellen einige Funktionen zur Manipulation dieser Strings zur Verfügung.

3.2 NtProcessStartup: Eintritt ins native Wunderland

NtProcessStartup ist der Entrypoint für native Applikationen. Nach dem Erzeugen des Prozesses, wird die Kontrolle an diese Funktion übergeben. Anders als bei Win32-Programmen ist es allerdings die Aufgabe des nativen Programms, seinen eigenen Prozess am Ende wieder zu zerstören.[1][8]

Abbildung 5 zeigt die Funktion.

Der Parameter von NtProcessStartup ist vom Typ `PPEB`. `PEB` ist der Process Environment Block. Jeder Windowsprozess hat einen PEB an einer bestimmten Stelle seines Speichers abgelegt. Er enthält wichtige Laufzeitinformationen für den Prozess. Dazu gehören: Handle für Standard-

[8] Was wird sonst passieren? Vermutlich wird einfach beim `ret`-Befehl am Ende der Funktion, kein sinnvoller Rücksprungpunkt mehr auf dem Stack sein und deswegen vermutlich eine Ausnahme ausgelöst.

```
extern "C" void NtProcessStartup(::PPEB peb )
{
    NativeBootIO io;
    myIO=&io;

    UNICODE_STRING &cmdLine = peb->ProcessParameters->CommandLine;

    char **arguments;
    int argc;
    arguments=split_args(io,cmdLine.Buffer,cmdLine.Length/2,&argc);

    Main main(io,argc,arguments);

    main.addCommand("break",debugBreak);
    main.addCommand("setComputerNameFromFile",setCompnameFromFile);
    main.addCommand("setComputerName",setComputerNameCmd);

    main.showSplashScreen();

#ifdef INTERACTIVE
    if (startupWithKey(io,2,'v'))
        main.rpl();
    else
#endif
        setCompnameFromFile(io,0);

    NtTerminateProcess( NtCurrentProcess(), 0 );
}
```

Abbildung 5: NtProcessStartup

Input/Output/Error, Flags, Strukturen zur Heapverwaltung, Informationen zur Betriebssystemversion usw. Es sind nicht alle Felder dokumentiert. Viele Felder sind beschrieben bei [4, 5, 6, 3]. Unter anderem sind in einer Struktur(`RTL_USER_PROCESS_PARAMETERS`) alle Parameter für den Prozess abgelegt, die beim Starten übergeben wurden, unter anderem auch die Kommandozeile (peb->ProcessParameters->CommandLine).

Der ursprüngliche Beispielcode von Mark Russinovich hatte mehrere Fehler bezüglich der Kommandozeile und hat nur zufällig funktioniert. Näheres dazu im Anhang A.

Zu Beginn wird also die Kommandozeile in die einzelnen Parameter eingeteilt. Danach wird `main` instanziiert, der Controller für die Shell. Dann werden zum Testen einige Shellbefehle definiert. Schließlich wird - wenn das beim Builden angegeben wurde - die Shell gestartet. Ansonsten wird die Standardprozedur eingeleitet, die den Computernamen aus den per Kommandozeilenargument übergebenen Dateien ausliest und dann in die Registrierung schreibt.

Am Ende wird mit NtTerminateProcess der Prozess beendet.

3.3 Ändern des Computernamens

Der Vorgang zum Ändern des Computernamens setzt sich aus zwei Teilschritten zusammen:

- Lesen der Datei, die den neuen Namen enthält und die Extraktion dessen
- Setzen des Computernamens

Man kann dem Bootprogramm mehrere Dateinamen als Parameter mitgeben, der Reihe nach wird überprüft, ob eine dieser Dateien geöffnet werden und der neue Computername ausgelesen werden kann. Diese Logik ist in computername.cpp ab Zeile 386 definiert.

3.4 Öffnen und Lesen einer Datei

Die Datei computername.cpp enthält diesen Code zum Einlesen der Datei (Abb. 3.4).

Wie erkennbar ist, haben die NtDll.dll Funktionen immer sehr viele Parameter. Das hat den Grund, dass diese Befehle häufig mehrere Verwendungszwecke haben. Bekanntes Beispiel ist `ZwCreateFile`. Diese Funktion kann verwendet werden, um beinahe alles was einen Namen hat in Windows öffnen (oder erzeugen) zu können. Dazu gehören Dateien, Pipes und Geräte. Für viele NtDll Funktionen gibt es auf der Subsystemebene einfachere Varianten. Beispielsweise gibt es in kernel32.dll die Versionen `CreateFile` und `CreateFileEx`, die im Endeffekt aber beide auf die selbe Ntdll Funktion `ZwCreateFile` zurückgreifen.

Der Ablauf ist schematisch dieser:

- Initialisieren der benötigten Parameter (`RtlInitUnicodeString`,`InitializeObjectAttributes`)
- Öffnen der Datei (`ZwCreateFile`)
- Lesen der Datei (`ZwReadFile`)
- Schließen des Dateihandles (`ZwClose`)

Ich gehe kurz auf die einzelnen Schritte ein. Wie vorher beschrieben, ist `UNICODE_STRING` eine nicht null-terminierte String Struktur. Beim Initialisieren eines solchen Strings muss die Zeichenkette erst einmal vermessen werden. Dieses macht RtlInitUnicodeString. Es zählt die Größe des Puffers bis zum ersten Nullzeichen. Vorsicht Falle: Das Feld `UNICODE_STRING`.Length gibt eben nicht die Länge der Zeichenkette in Zeichen an, sondern den Platz, den der String im Puffer belegt (bei UNICODE: 2 * Anzahl der Zeichen).

`OBJECT_ATTRIBUTES` ist eine Struktur, die vielfältig verwendet werden kann, um Attribute von Objekten zu transportieren. Als Eingabeparameter zu ZwCreateFile muss es den Dateinamen enthalten.

Die Parameter zu ZwCreateFile sind im Code dokumentiert. Interessante Parameter sind:

- desired access: Wir möchten Lesezugriff, also `GENERIC_READ`. `SYNCHRONIZE` bedeutet, dass man das Dateihandle, welches ZwCreateFile erzeugt, mit den Windows Threadsynchronisationsmethoden(waitForSingleObject et al.) verwenden kann. Dieses Flag ist Voraussetzung für `FILE_SYNCHRONOUS_IO_NONALERT`.
- share access: Gibt an, ob andere Prozesse auf diese Datei Zugriff erhalten dürfen, während unser Dateihandle gültig ist. 0 bedeutet exklusiven Zugriff für unser Dateihandle
- create options: `FILE_SYNCHRONOUS_IO_NONALERT` bedeutet, dass alle Lese- und Schreibzugriffe mit diesem Handle synchron ausgeführt werden. Das bedeutet, dass ZwReadFile (und andere) erst zum Aufrufer zurückkehren, wenn die Operation vollständig abgeschlossen ist.

Wurde die Datei erfolgreich geöffnet, wird ein Puffer angelegt. Der Einfachheit halber werden grundsätzlich nur die ersten 256 Zeichen gelesen. Dies geschieht mit ZwReadFile. Dieses nimmt als Parameter das Handle, dann drei Parameter, die nur für die asynchrone Ausführung interessant sind, und hier einfach ignoriert werden. Iosb ist ein `IO_STATUS_BLOCK`. Dieser wird bei allen IO-Operationen dazu verwendet, Informationen über die Ausführung dieser Operation zu erhalten. Beim Erzeugen einer Datei mit ZwCreateFile wird darin gespeichert, ob eine neue Datei angelegt wurde oder ob eine alte Datei überschrieben. Bei ZwReadFile empfängt Iosb - für uns wichtig - die Anzahl der gelesenen Zeichen.

Im Anschluss daran und aus dem Codeausschnitt entfernt folgt das Parsen des Dateiinhaltes und das Suchen nach dem Computernamen darin. Eine einfache Funktion, die simple reguläre Ausdrücke versteht, erledigt diese Aufgabe (`parseComputerNameFile`).

Schließlich folgt ZwClose, das ein Handle wieder schließt und damit die Resourcen wieder freigibt.

Dateinamen in Windows Ein eigenes Kapitel hätten Dateinamen in Windows verdient. Die verschiedenen Schichten von Windows interpretieren Dateinamen unterschiedlich. Auf der obersten Ebene, die des Benutzers, sind alle Dateinamen DOS kompatibel. Es gelten die alten „8+3"-Konventionen und die Erweiterung für lange Dateinamen. Eine Beachtung der Groß- und Kleinschreibung ist nicht nötig. Laufwerken sind Buchstaben zugeordnet. Diese Interpretation wird hauptsächlich von Win32 implementiert.

Unter der Oberfläche funktioniert Windows ganz anders: Es gibt einen allgemeinen Namensraum, in dem alle Objekte eintragen sind. Festplatten und Disketten sind über mehrere Namen erreichbar. Um auf eine Datei auf einer Diskette zugreifen zu können, gibt es mehrere Möglichkeiten, diese umfassen:

- \Device\Floppy0\compname.txt
- \??\A:\compname.txt

Device ist ein Verzeichnis im Namensraum, in welchem alle Geräte des Computers eingetragen sind. Floppy0 entspricht da dem ersten Diskettenlaufwerk. Die zweite Notation ist die aus Kompabilitätsgründen eingeführte: Es gibt im Namensraum ein Verzeichnis „??", auch angegeben mit „Global??", das symbolische Links auf häufig benötigte Objekte enthält. Wie beispielsweise den Buchstaben für die Laufwerke. Die Win32-Schicht überprüft, ob ein gültiger (DOS-)Pfad vorliegt und hängt dann „\??\" davor. So wird daraus ein gültiger Name für den Namensraum und für den Aufruf der entsprechenden NtDll-Funktion.

Es ist auch möglich Laufwerksbuchstaben pro Benutzer/Session anzulegen. Diese werden dann im Namespace unter \Sessions\<x>\DosDevices hinterlegt (x ist die Nummer der Session).

Grundsätzlich ist auf der unteren Ebene Groß- und Kleinschreibung nicht egal. Allerdings kann wohl ein Flag(`OBJ_CASE_INSENSITIVE`) gesetzt werden, mit dem angegeben wird, ob Namen case-sensitive behandelt werden sollen oder nicht. Das sysinternals Programm WinObj[9] kann dazu verwendet werden, alle Objekte und ihre Sicherheitseinstellungen im Namensraum von Windows anzuzeigen.

Auf dem Screenshot(Abbildung 7) sind die Laufwerk A:,C: und D: erkennbar, die jeweils zu verschiedenen physischen bzw. logischen Laufwerken verknüpft sind.

3.5 Setzen des Computernamens

Der Computername ist in mehreren Schlüsseln in der Registrierung gespeichert. Uns bekannt sind die folgenden:

- \Registry\Machine\SYSTEM\CurrentControlSet\Control\ComputerName\ComputerName
- \Registry\Machine\SYSTEM\CurrentControlSet\Control\ComputerName\ActiveComputerName
- \Registry\Machine\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters

An diesen Stellen muss der Computername eingesetzt werden. Die Funktion, die dieses übernimmt ist `setRegistryValue`(Abb. 3.5).

Auch diese Funktion führt mehrere, ähnliche Schritte durch, wie zuvor beim Lesen der Datei:

- Initialisieren der Parameter
- Öffnen des Registrierungsschlüssels
- Schreiben des Computernamens
- Schließen des Handles

[9] http://www.microsoft.com/technet/sysinternals/utilities/WinObj.mspx

Das Öffnen eines Schlüssels ist analog zum Öffnen einer Datei. Der Name des Schlüssels wird in den `ObjectAttributes` gespeichert. Mit `KEY_ALL_ACCESS` wird der Zugriffsmodus angegeben. `REG_OPTION_NON_VOLATILE` gibt an, dass Änderungen am Schlüssel nicht-flüchtig sind, d.h. in die Registrierungsdatei geschrieben werden und auch noch nach dem nächsten Neustart vorhanden sind. Disposition empfängt ein Flag, das angibt, ob der Schlüssel neu erzeugt werden musste oder ob ein vorhandener Schlüssel geöffnet wurde.

`ZwSetValueKey` setzt den entsprechenden Registrierungswert. `SoftwareKeyHandle` ist das oben geöffnete Handle des Schlüssel. `ValueName` ist der Name des Schlüssels. `REG_SZ` ist der Typ des Wertes (null-terminierter String). Außerdem wird ein Zeiger auf die binären Daten (in unserem Fall, der null-terminierte String) und dessen Länge übergeben.

Auch diese Funktion endet mit `ZwClose`, das ein geöffnetes Handle wieder schließt.

4 Fazit

In dieser Arbeit habe ich einige Grundlagen des Windowsbetriebssystems erläutert, den Aufbau der Windows APIs betrachtet und einen Überlick über den Bootprozess gegeben. Besonderer Augenmerk lag auf der „Windows Native API“.

Mit diesem Wissen wurde ein Bootprogramm entwickelt, dass den Computernamen aus einer Datei auslesen und die Registrierung schreiben kann. Es hat sich gezeigt, dass es durchaus möglich ist, sinnvolle Programme mit dieser grundlegensten der Windows User-Mode APIs zu entwickeln. Das Hauptproblem bei der Benutzung dieser Schnittstelle ist deren mangelnde Dokumentation. Im Internet findet man heutzutage einige Tips, viele kommen von Homepages, die sich mit Rootkits und Ähnlichem beschäftigen. Da stellt sich einmal wieder die Frage, ob es sinnvoll ist die Details eines Betriebssystems geheim zu halten und somit vor allen anderen den (gut- oder bösartigen) „Computersicherheitsfachleuten“ zu überlassen...

Die sysinternals Homepage war (und ist) seit vielen Jahren schon die einzige verlässliche Quelle von derartigen Informationen. So ist auch das Buch der Betreiber dieser Seite[6] mein ständiger Begleiter gewesen.

Ob Microsofts Erwerb von Sysinternals ein Schritt in die richtige Richtung gewesen ist, ist nicht leicht zu beantworten. Microsoft hat sich mit diesem Schritt einen großen Teil des Out-Of-Microsoft Insiderwissens wieder einverleibt.

Das eigentliche Ziel des Projektes ist erreicht: das Ändern des Computernamens ohne Neustart. Eine einfache Erweiterung des Systemes ist auch schon abzusehen: Das Setzen beliebiger Registrierungswerte beim Booten, beispielsweise der Benutzerdaten für das automatische Logon.

A CommandLine Parsing Bug im Sysinternals Beispielcode

Der Beispielcode von sysinternals.net verwendete als Parameter für NtProcessStartup einen Zeiger zur Struktur `STARTUP_ARGUMENTS`, so definiert:

```
//
// Environment information, which includes command line and
// image file name
//
typedef struct {
       ULONG            Unknown[21];
       UNICODE_STRING   CommandLine;
       UNICODE_STRING   ImageFile;
} ENVIRONMENT_INFORMATION, *PENVIRONMENT_INFORMATION;

//
// This structure is passed as NtProcessStartup's parameter
//
typedef struct {
       ULONG                     Unknown[3];
       PENVIRONMENT_INFORMATION  Environment;
} STARTUP_ARGUMENT, *PSTARTUP_ARGUMENT;
```

Dieser Parameter wurde dann dazu verwendet, um das Kommandozeilenargument zu extrahieren:

```
extern "C" void NtProcessStartup( PSTARTUP_ARGUMENT Argument ) {
    //
    // Point at command line
    //
    commandLine = &Argument->Environment->CommandLine;

    //
    // Locate the argument
    //
    argPtr = commandLine->Buffer;
    while( *argPtr != L'␣' ) argPtr++;
    argPtr++;

    //...
}
```

Als ich ähnlichen Code verwendete, um die Kommandozeile zu lesen, allerdings sicherheitshalber auch das `commandLine->Length`-Feld benutzte, funktionierte das Parsen der Kommandozeile nicht mehr. Ich extrahierte immer den vollen Pfad des Programms, und nicht die Kommandozeile. Wie ich dann feststellte, zeigte `commandLine->Buffer` überhaupt nicht auf die Kommandozeile, sondern auf den Buffer, der den Pfad des Images enthielt. Mit dem Debugger konnte ich im Speicher erkennen, was in Abbildung 9 gezeigt wird.

In einer Email an Mark Russinovich schrieb ich(Anfang April 2006):

> What you've done is not quite correct (at least not secure), because you move over the string not caring about any terminating zero characters. I took the save way and operated on the string using `UNICODE_STRING`'s Length field. But when I tried to get it to work, I always got the full image path instead of the command line. That was strange, because your sample code works without doubt.

Warum funktionierte der ursprüngliche Code? Zufällig liegen im Speicher ImagePath String und CommandLine String immer hintereinander. Da das Beispielprogramm nur nach Leerzeichen suchte, um den ersten Parameter zu finden, und ImagePath keine enthielt, suchte die Schleife einfach weiter und fand welche in der anschließenden Kommandozeile. So einfach.

Doch das war nicht der einzige Zufall: In der jetzigen Version verwendet das Bootprogramm als Parameter `PEB`, den Process Environment Block, wie das an einigen Stellen vorgeschlagen wird.

```
typedef struct _PEB {
                BOOLEAN                 InheritedAddressSpace;
                BOOLEAN                 ReadImageFileExecOptions;
                BOOLEAN                 BeingDebugged;
                BOOLEAN                 Spare;
                HANDLE                  Mutant;
                PVOID                   ImageBaseAddress;
                PVOID/*PPEB_LDR_DATA*/  LoaderData;
                PRTL_USER_PROCESS_PARAMETERS ProcessParameters;
                //...
    }PEB, *PPEB;
```

Vergleicht man `PEB` und `STARTUP_ARGUMENTS` sieht man, dass der Zeiger `LoaderData` dem Zeiger auf `Environment` entspricht. Wie aber in der jetztigen Fassung von NtProcessStartup (Abb. 5) zu sehen ist, ist die Kommandozeile erst ein Feld der Struktur `ProcessParameters`. Zufällig liegen aber auch diese beiden, `LoaderData` und `ProcessParameters` direkt hintereinander im Speicher und so genügt es mit `ULONG Unknown[21]` genügend Speicher zu überspringen.
Mark Russinovich antwortete mir:

```
Hi Johannes,

Thanks for pointing that out. The command-line reading was basically
just a hack.
```

```
We actually wrote a full-blown command-line/recovery environment as
a native APP with our (Winternals) first relase of ERD Commander
back in 1998. O&O Software (http://www.oandosoftware.com/) copied us
and still ships such an app as their O&O Bluecon (we've since moved
to using Windows PE).

-Mark
```

Vielleicht wollte Sysinternals damals nicht zu viele Informationen rausgeben, indem sie `STARTUP_ARGUMENTS` einbauten und nicht `PEB`?

Literatur

[1] *Inside Native Applications.* http://www.microsoft.com/technet/sysinternals/information/NativeApplications.mspx

[2] *ReactOS ctxSwitch.S Revision 28153.* http://svn.reactos.org/svn/reactos/trunk/reactos/ntoskrnl/ke/i386/ctxswitch.S?revision=28153&view=markup

[3] *ReactOS winternl.h.* http://svn.reactos.org/svn/reactos/trunk/reactos/include/psdk/winternl.h?view=markup

[4] *Undocumented Functions for Microsoft Windows NT/2000.* http://undocumented.ntinternals.net/

[5] Nebbett, Gary: *Windows NT/2000 Native API Reference.* Thousand Oaks, CA, USA : New Riders Publishing, 2000. – ISBN 1578701996

[6] Russinovich, Mark E. ; Solomon, David A.: *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server 2003, Windows XP, and Windows 2000.* Fourth Edition. Microsoft Press, 2005

[7] *Windows System Call Table (NT/2000/XP/2003/Vista).* http://metasploit.com/users/opcode/syscalls.html

```
wchar_t *readComputerNameFromFile(IO &io,wchar_t *fileName)
{
        Indenter i(io);
        NTSTATUS Status;
        UNICODE_STRING UnicodeFilespec;
        OBJECT_ATTRIBUTES ObjectAttributes;
        HANDLE FileHandle;
        IO_STATUS_BLOCK Iosb;
        char *buffer;
        PWCHAR buffer2;
        ULONG converted;

        RtlInitUnicodeString(&UnicodeFilespec,fileName);

        InitializeObjectAttributes(
                &ObjectAttributes,          // ptr to structure
                &UnicodeFilespec,           // ptr to file spec
                OBJ_CASE_INSENSITIVE,       // attributes
                NULL,                       // root directory handle
                NULL );                     // ptr to security descriptor

        Status = ZwCreateFile(
                &FileHandle,                    // returned file handle
                (GENERIC_READ | SYNCHRONIZE),   // desired access
                &ObjectAttributes,              // ptr to object attributes
                &Iosb,                          // ptr to I/O status block
                0,                              // allocation size
                FILE_ATTRIBUTE_NORMAL,          // file attributes
                0,                              // share access
                FILE_OPEN,                      // create disposition
                FILE_SYNCHRONOUS_IO_NONALERT,   // create options
                NULL,                           // ptr to extended attributes
                0);                             // length of ea buffer


        CHECK_STATUSA(Status,Öffnen der Computernamensdatei)
        RETURN_NULL_IF_STATUS_UNSUCCESSFUL

        buffer = (char*)io.malloc(256);
        Status = ZwReadFile(FileHandle,0,NULL,NULL,&Iosb,buffer,256,0,NULL);

        // ...

        Status = ZwClose(FileHandle);
}
```

Abbildung 6: readComputerNameFromFile

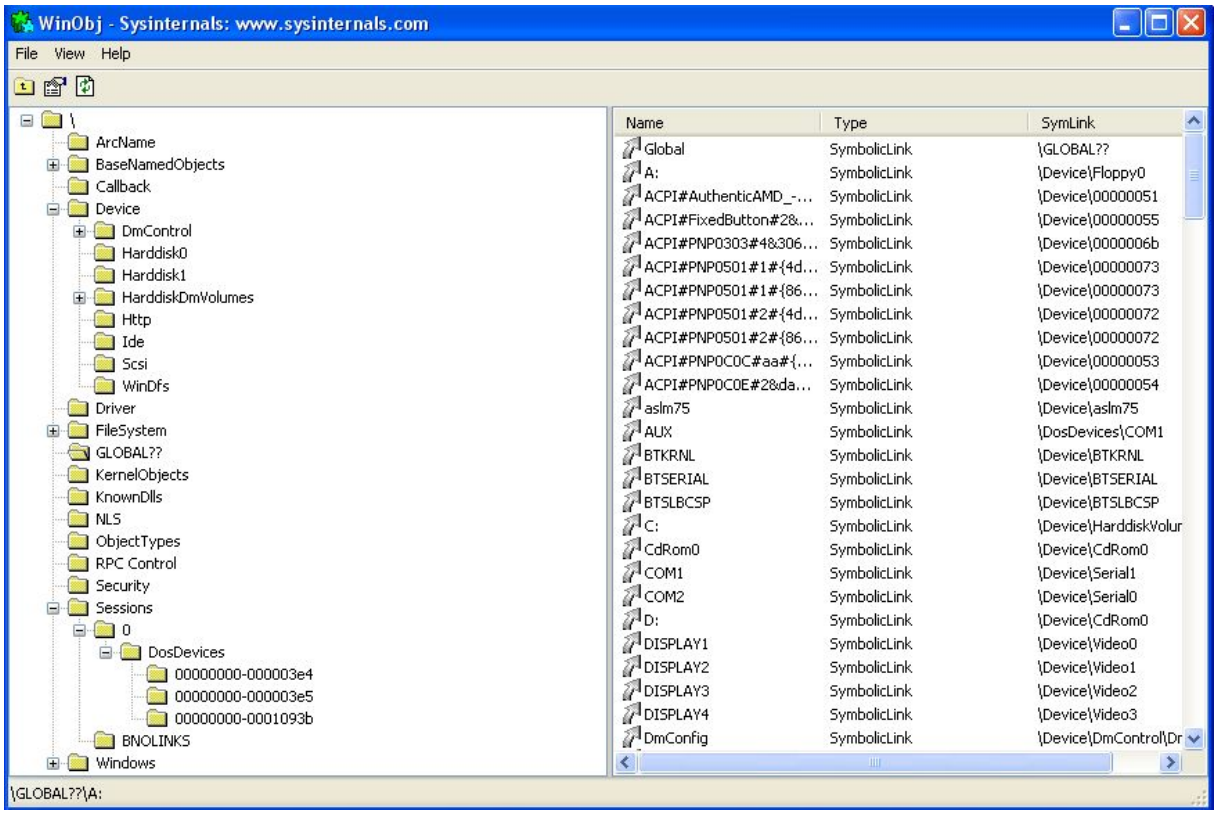

Abbildung 7: sysinternals' WinObj

```
void setRegistryValue(IO &io,WCHAR *keyName,WCHAR *valueName,WCHAR *value)
{
        Indenter i(io);
        UNICODE_STRING KeyName, ValueName;
        HANDLE SoftwareKeyHandle;
        ULONG Status;
        OBJECT_ATTRIBUTES ObjectAttributes;
        ULONG Disposition;

        //
        // Open the Software key
        //
        NT::RtlInitUnicodeString(&KeyName,keyName);
        InitializeObjectAttributes(
                &ObjectAttributes,
                &KeyName,
                OBJ_CASE_INSENSITIVE,
                NULL,
                NULL);
        Status = ZwCreateKey(
                &SoftwareKeyHandle,
                KEY_ALL_ACCESS,
                &ObjectAttributes,
                0,
                NULL,
                REG_OPTION_NON_VOLATILE,
                &Disposition);

        CHECK_STATUS(Status,Öffnen des Schlüssels)

        NT::RtlInitUnicodeString(&ValueName,valueName);

        Status = ZwSetValueKey(
                SoftwareKeyHandle,
                &ValueName,
                0,
                REG_SZ,
                value,
                (wcslen( value )+1) * sizeof(WCHAR));

        CHECK_STATUSA(Status,Setzen des Schlüssels);

        Status = ZwClose(SoftwareKeyHandle);

        CHECK_STATUS(Status,Schließen des Schlüssels);
}
```

Abbildung 8: setRegistryValue

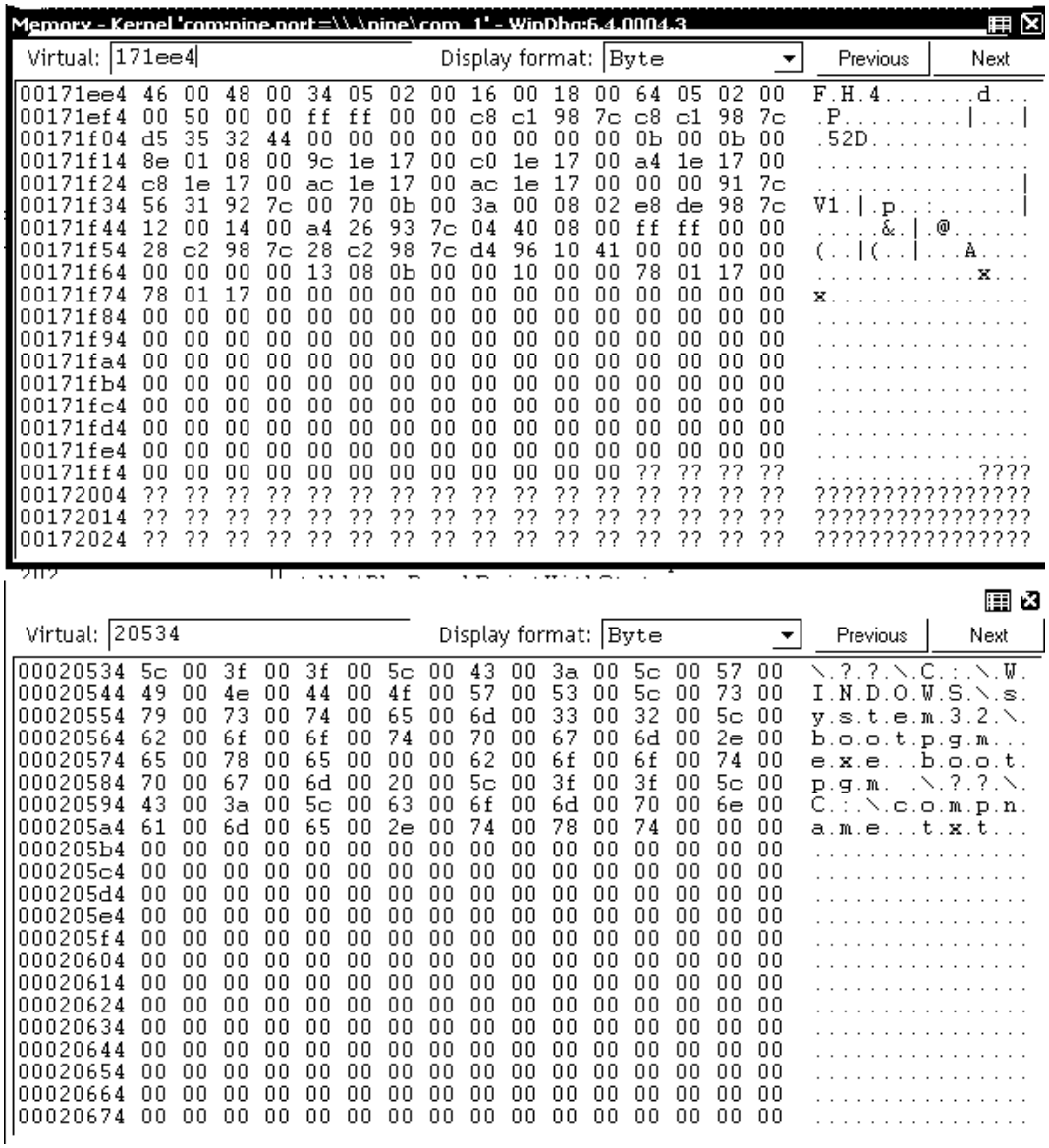

Abbildung 9: Speicher