PEARSON Education

ADDISON-WESLEY

Greg Hoglung, James Butler

Rootkits

Windows-Kernel unterwandern

ADDISON-WESLEY

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

3 Die Hardware-Connection

»Ein Ring sie zu knechten, sie alle zu finden, ins Dunkel zu treiben und ewig zu binden.«

– J.R.R. TOLKIEN,
»DER HERR DER RINGE: DIE GEFÄHRTEN«

Software und Hardware gehören zusammen. Ohne Software wäre Hardware einfach nur totes Silizium, ohne Hardware könnte Software nicht existieren. Die Software steuert letztlich den Computer, aber unter der Oberfläche ist es die Hardware, die den Softwarecode ausführt.

Darüber hinaus ist die Hardware die Instanz, die letzten Endes die Softwaresicherheit durchsetzt. Ohne Unterstützung durch die Hardware wäre Software völlig unsicher. Viele Bücher behandeln Softwareentwicklung, ohne sich jemals um die zugrunde liegende Hardware zu kümmern. Das mag für Entwickler von Unternehmensanwendungen angehen, aber nicht für Rootkit-Programmierer. Letztere müssen sich mit Reverse-Engineering-Problemen, mit handkodierten Assemblersprachen und technisch ausgefeilten Angriffen gegen Softwaretools auf dem Zielsystem herumplagen. Nur mit einem Verständnis der zugrunde liegenden Hardware können Sie diese Probleme angehen. In diesem Buch werden Ihnen überall Begriffe und Code begegnen, für deren Verständnis Sie ein Minimum an Hardwarekenntnissen mitbringen müssen. Daher sollten Sie dieses Kapitel nicht überspringen.

Letztlich ist die gesamte Zugriffssteuerung in der Hardware implementiert. Die bekannteste Art der Prozesstrennung wird durch »Ringe« auf der Intel x86-Hardware herbeigeführt. Hätte eine Intel-CPU keine Mechanismen für die Zugriffssteuerung, so würde der gesamten Software auf dem System vertraut. Das bedeutet, dass jedes Programm, das abstürzt, das gesamte System mit in den Abgrund reißen könnte. Jedes Programm hätte die Möglichkeit, Hardware zu lesen oder zu beschreiben, auf beliebige Dateien zuzugreifen oder den Speicher eines anderen Prozesses zu verändern. Hört sich das bekannt an? Obwohl die Intel-Prozessorfamilie schon viele Jahre lang die Fähigkeit zur Zugriffssteuerung mitbrachte, hat Microsoft bis zur Veröffentlichung von Windows NT keinen Gebrauch davon gemacht.

In diesem Kapitel beschreiben wir die Hardwaremechanismen, die hinter den Kulissen ablaufen, um Sicherheit und Speicherzugriff in Windows

durchzusetzen. Wir beginnen, indem wir uns anschauen, wie Intel x86-Prozessoren die Zugriffssteuerung durchführen, und beschreiben dann, wie sich der Prozessor über Lookup-Tabellen auf dem Laufenden hält. Außerdem besprechen wir Register und, was das Wichtigste ist, die Funktionsweise von Speicherseiten.

Ring 0

Die Intel x86-Familie nutzt das Prinzip von *Ringen* für die Zugriffssteuerung. Es gibt vier Ringe, wobei Ring 0 der mit den höchsten Privilegien ist und Ring 3 derjenige mit den geringsten. Intern wird jeder Ring als Zahl gespeichert; es gibt keine physischen Ringe in dem Mikrochip.

Der gesamte Kernelcode von Windows wird in Ring 0 ausgeführt, weshalb auch Rootkits, die im Kernel laufen, in diesem Ring ausgeführt werden. User-Mode-Programme, die nicht im Kernel laufen (z.B. ein Tabellenkalkulationsprogramm), werden manchmal auch *Ring-3-Programme* genannt. Viele Betriebssysteme, darunter Windows und Linux, nutzen lediglich Ring 0 und 3 eines Intel x86-Mikrochips, nicht aber Ring 1 und 2.[1] Da Ring 0 der Ring mit den meisten Privilegien und den meisten Möglichkeiten ist, sind Rootkit-Entwickler stolz darauf, wenn ihr Code in Ring 0 ausgeführt wird.

Die CPU ist dafür verantwortlich, darüber Buch zu führen, welcher Softwarecode und welcher Speicher welchem Ring zugewiesen werden, und die Zugriffsbeschränkungen der einzelnen Ringe durchzusetzen. Gewöhnlich wird jedem Softwareprogramm eine Ringnummer zugewiesen, sodass es auf keinen Ring mit einer geringeren Nummer zugreifen kann. Daher hat ein Ring-3-Programm keinen Zugriff auf ein Ring-0-Programm. Wenn es versucht, auf Ring-0-Speicher zuzugreifen, löst die CPU einen Interrupt aus. In den meisten Fällen dieser Art erlaubt das Betriebssystem den Zugriff nicht. Der Versuch kann sogar dazu führen, dass das Ring-3-Programm beendet wird.

Diese Zugriffseinschränkungen werden von einer ziemlichen Menge Code gesteuert. Es gibt auch Code, der Programmen unter bestimmten Umständen erlaubt, auf Ringe mit geringerer Nummer zuzugreifen. Das Laden eines Druckertreibers in den Kernel erfordert zum Beispiel, dass ein Administratorprogramm (Ring 3) Zugriff auf die geladenen Gerätetreiber (Kernel in Ring 0) hat. Nachdem ein Kernel-Mode-Rootkit geladen ist, wird sein Code in Ring 0 ausgeführt, sodass diese Einschränkungen nicht mehr gelten.

1. Ring 1 und 2 können zwar verwendet werden, sind aber für die Architektur von Windows nicht notwendig.

Viele Tools, die Rootkits entdecken können, laufen als Administratorprogramme in Ring 3. Ein Rootkit-Entwickler sollte wissen, wie er die Tatsache ausnutzen kann, dass sein Rootkit höhere Privilegien hat als das Administratortool. Das Rootkit kann sich dadurch z.B. vor dem Tool verstecken oder es wirkungslos machen. Außerdem werden Rootkits gewöhnlich mit Ladeprogrammen installiert (besprochen in Kapitel 2). Diese Ladeprogramme sind Ring-3-Anwendungen. Um ein Rootkit in den Kernel einzuführen, muss das Ladeprogramm besondere Funktionsaufrufe verwenden, die ihm den Zugriff auf Ring 0 erlauben.

Abbildung 3.1 zeigt die Ringe eines Intel x86-Prozessors und gibt an, wo User- und Kernel-Mode-Programme ausgeführt werden.

Neben den Einschränkungen für den Speicherzugriff gibt es noch andere Sicherheitsmaßnahmen. Einige Anweisungen werden als privilegiert betrachtet und können nur in Ring 0 ausgeführt werden. Gewöhnlich sind das die Anweisungen, die das Verhalten der CPU ändern oder direkt auf Hardware zugreifen. Die folgenden x86-Anweisungen sind nur in Ring 0 erlaubt:

- `cli` – Interruptverarbeitung beenden (auf der aktuellen CPU)
- `sti` – Interruptverarbeitung starten (auf der aktuellen CPU)
- `in` – Daten von einem Hardwareanschluss lesen
- `out` – Daten an einen Hardwareanschluss schreiben

Ein Rootkit in Ring 0 auszuführen, bietet viele Vorteile. Es solches Rootkit kann nicht nur andere Hardware manipulieren, sondern auch die Umgebung, in der die andere Software läuft. Das ist entscheidend für die Durchführung getarnter Operationen auf einem Computer.

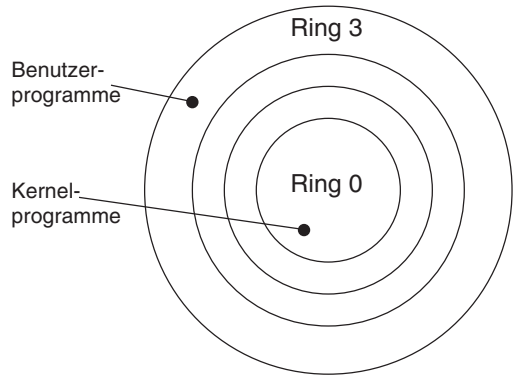


Abbildung 3-1 Die Ringe eines Intel x86-Prozessors

Nachdem wir erklärt haben, wie die CPU die Zugriffssteuerung durchsetzt, schauen wir uns an, wie sie sich über wichtige Daten auf dem Laufenden hält.

Tabellen, Tabellen und noch einmal Tabellen

Die CPU muss sich nicht nur um die Ringe kümmern, sondern auch andere Entscheidungen treffen, z.B., was zu tun ist, wenn ein Interrupt ausgelöst wird, wenn ein Softwareprogramm abstürzt, wenn die Hardware Signale aussendet, um auf sich aufmerksam zu machen, wenn User-Mode-Programme versuchen, mit Kernel-Mode-Programmen zu kommunizieren, und wenn Programme von einem Thread zum anderen umschalten. Natürlich muss der Code des Betriebssystems für all diese Vorfälle Sorge tragen – aber die CPU hat damit als Erste zu tun.

Bei jedem wichtigen Ereignis muss die CPU herausfinden, welche Softwareroutine sich darum kümmert. Da sich jede Softwareroutine im Arbeitsspeicher befindet, ist es sinnvoll, dass die CPU die Adressen der wichtigen Routinen speichert. Genauer gesagt, muss die CPU wissen, wo sie die Adresse einer wichtigen Softwareroutine *finden* kann. Sie kann nicht alle Adressen intern speichern, sondern muss die Werte nachschlagen, und zwar in Adresstabellen. Wenn ein Ereignis wie ein Interrupt eintritt, sucht die CPU dieses Ereignis in der Tabelle und findet die zugehörige Adresse einer Software, die mit diesem Ereignis umgehen kann. Die einzige Information, die die CPU benötigt, ist die Adresse dieser Tabelle im Speicher.

Es gibt viele wichtige CPU-Tabellen, darunter die folgenden:

- Globale Deskriptortabelle (GDT) zur Zuordnung von Adressen
- Lokale Deskriptortabelle (LDT) zur Zuordnung von Adressen
- Seitenverzeichnis (Page Directory) zur Zuordnung von Adressen
- Interrupt-Deskriptortabelle (IDT) zur Suche nach Interrupt-Handlern

Zusätzlich zu denen der CPU kann das Betriebssystem eigene Tabellen unterhalten. Sie werden nicht direkt von der CPU unterstützt, sodass das Betriebssystem besondere Funktionen und Code enthalten muss, um sie zu verwalten.

Eine wichtige durch das Betriebssystem implementierte Tabelle ist folgende:

- System Service Dispatch Table (SSDT) zur Handhabung von Systemaufrufen durch Windows

Diese Tabellen werden auf mannigfaltige Weise verwendet. In den folgenden Abschnitten kommen wir auf diese Tabellen zurück und erklären,

wie sie funktionieren. Außerdem stellen wir Möglichkeiten vor, mit denen ein Rootkit-Entwickler diese Tabellen ändern oder mit Hooks versehen kann, um sich zu tarnen oder Daten zu erfassen.

Speicherseiten

Der gesamte Arbeitsspeicher ist wie ein Buch in Seiten aufgeteilt, wobei jede Seite eine bestimmte Anzahl von Zeichen enthalten kann. Jeder Prozess kann über eine eigene Lookup-Tabelle verfügen, um diese Speicherseiten zu finden.

Stellen Sie sich den Arbeitsspeicher als eine gigantische Bibliothek vor, in der jeder Prozess seinen eigenen Katalog hat, um die einzelnen Bücher zu finden. Die verschiedenen Lookup-Tabellen können dazu führen, dass der Speicher von jedem Prozess ganz anders gesehen wird. Aus diesem Grund kann ein Prozess den Speicher an der Adresse `0x00401122` lesen und dort `GREG` sehen, während ein anderer an derselben Adresse `JAMES` findet. Jeder Prozess hat seine eigene »Sicht« des Speichers.

Für die Speicherseiten gibt es eine Zugriffssteuerung. Um unsere Bibliotheksmetapher weiterzuführen, stellen Sie sich die CPU als übereifrigen Bibliothekar vor, der einem Prozess nur erlaubt, einige wenige Bücher einzusehen. Um im Speicher zu lesen oder zu schreiben, muss der Prozess zuerst das richtige »Buch« finden und dann die genaue »Seite« des gewünschten Speicherinhalts. Wenn die CPU keinen Blick in das betreffende Buch oder die Seite erlaubt, wird der Zugriff verweigert.

Das Verfahren, um eine Seite zu finden, ist dabei sehr lang und kompliziert, denn Zugriffseinschränkungen werden in mehreren Phasen dieses Vorgangs durchgesetzt. Als Erstes prüft die CPU, ob der Prozess das gewünschte Buch öffnen darf (*Deskriptor*-Test), dann, ob er ein bestimmtes Kapitel in dem Buch lesen kann (der Test des *Seitenverzeichnisses*). Zum Schluss erfolgt die Prüfung, ob der Prozess eine bestimmte Seite in diesem Kapitel lesen darf (der *Seiten*-Test). Eine ganze Menge Arbeit!

Nur wenn der Prozess alle drei Sicherheitsüberprüfungen besteht, wird ihm erlaubt, die Seite zu lesen.

Selbst wenn alle CPU-Tests bestanden sind, kann es immer noch sein, dass die Seite als schreibgeschützt gekennzeichnet ist. Das bedeutet, dass der Prozess die Seite zwar lesen, aber nicht beschreiben darf. Auf diese Weise wird die Integrität der Daten erhalten. Rootkit-Entwickler verhalten sich wie Vandalen in dieser Bibliothek, da sie versuchen, alle Bücher zu bekritzeln. Wir müssen also lernen, wie Sie die Zugriffssteuerung manipulieren können.

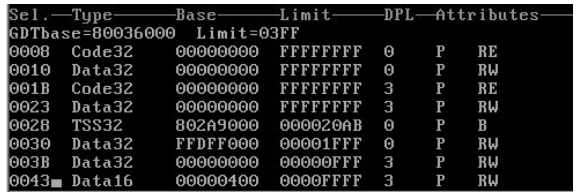
Einzelheiten der Tests beim Speicherzugriff

Beim Zugriff auf eine Speicherseite führt der x86-Prozessor folgende Tests in der angegebenen Reihenfolge durch:

- Deskriptortest (oder *Segmenttest*): Dabei wird gewöhnlich die globale Deskriptortabelle (GDT) aufgerufen und der Segmentdeskriptor überprüft. Dieser Deskriptor enthält einen Wert, der als *Descriptor Privilege Level* (DPL) bekannt ist. Der DPL enthält die Ringnummer (0 oder 3), die der aufrufende Prozess haben muss. Ist der DPL kleiner als der Ringlevel des aufrufenden Prozesses (manchmal *Current Privilege Level* oder CPL genannt), wird der Zugriff verweigert und die Speicherüberprüfung an dieser Stelle beendet.
- Seitenverzeichnistest: Ein User/Supervisor-Bit für eine gesamte Seitentabelle (also einen kompletten Bereich von Speicherseiten) wird überprüft. Ist das Bit auf 0 gesetzt, können nur Supervisor-Programme (Ringe 0, 1 und 2) auf diesen Bereich zugreifen. Falls der aufrufende Prozess dann kein Supervisor ist, wird die Speicherüberprüfung hier beendet. Ist das User/Supervisor-Bit auf 1 gesetzt, hat jedes Programm Zugang zu diesem Bereich der Speicherseiten.
- Seitentest: Dieser Test wird für einzelne Speicherseiten ausgeführt. Nach dem Bestehen des Seitenverzeichnistests erfolgt der Test für die angeforderte Seite. Wie das Seitenverzeichnis hat jede einzelne Seite ein User/Supervisor-Bit. Ist das Bit auf 0 gesetzt, können nur Supervisor-Programme (Ringe 0, 1 und 2) auf diesen Bereich zugreifen, hat es den Wert 1, so hat jedes Programm Zugang zu diesem Bereich der Speicherseiten. Ein Prozess kann auf eine Speicherseite nur zugreifen, wenn er sämtliche Tests ohne Zugriffsverweigerung besteht.

Windows-Betriebssysteme führen keinen Deskriptortest durch, sondern verlassen sich auf die Trennung von Ring 0 und 3 (*Kernel-Mode* und *User-Mode*). Dadurch steuert das User/Superviser-Bit in der Seitentabelle allein den Zugriff auf den Speicher. Kernel-Mode-Programme in Ring 0 haben immer Zugriff auf den Speicher, User-Mode-Programme in Ring 3 können nur auf Speicher zugreifen, der als »user« gekennzeichnet ist.

Abbildung 3.2 zeigt einen (mit SoftIce erstellten) Dump einer GDT von Windows 2000 mit dem DPL für jeden Eintrag. Die ersten vier Einträge (08, 10, 1B und 23) umfassen den gesamten Speicherbereich für Daten und Code und sowohl Ring-0- als auch Ring-3-Programme, weshalb die GDT keinerlei Sicherheit für das System bietet. Die Sicherheit muss tiefer unten in den Seitentabellen durchgesetzt werden. Um das in allen Einzelheiten zu verstehen, müssen Sie wissen, wie virtuelle in physische Speicheradressen übersetzt werden. Das erklären wir im nächsten Abschnitt.



```
Sel.—Type———Base———Limit———DPL—Attributes———
GDTbase=80036000  Limit=03FF
0008  Code32    00000000  FFFFFFFF  0    P   RE
0010  Data32    00000000  FFFFFFFF  0    P   RW
001B  Code32    00000000  FFFFFFFF  3    P   RE
0023  Data32    00000000  FFFFFFFF  3    P   RW
0028  TSS32     802A9000  000020AB  0    P   B
0030  Data32    FFDFF000  00001FFF  0    P   RW
003B  Data32    00000000  00000FFF  3    P   RW
0043  Data16    00000400  0000FFFF  3    P   RW
```

Abbildung 3-2 Die GDT von Windows 2000

Auslagerung und Adressübersetzung

Der Speicherschutzmechanismus wird nicht nur für die Sicherheit verwendet. Die meisten modernen Betriebssysteme unterstützen virtuellen Speicher, wodurch jedes Programm auf dem System seinen eigenen Adressraum verwenden kann. Außerdem kann ein Programm dadurch mehr Speicher nutzen, als im »Hauptspeicher« tatsächlich zur Verfügung steht. Auf einem Computer mit 256 Mbyte RAM sind die Programme nicht auf 256 Mbyte Speicher beschränkt, sondern können mit Leichtigkeit ein Gbyte Speicher verwenden, wenn sie das wollen: Der zusätzliche Speicher wird einfach in einer Datei (der *Auslagerungsdatei*) auf der Festplatte abgelegt. Virtueller Speicher erlaubt es, mehrere Prozesse gleichzeitig durchzuführen, jeden mit seinem eigenen Speicher, wenn die von allen Prozessen verwendete Gesamtmenge größer ist als der installierte physische RAM.

Speicherseiten können als *ausgelagert* gekennzeichnet sein (also auf der Festplatte gespeichert statt im RAM). Wenn nach einer solchen Seite gesucht wird, löst der Prozessor einen Interrupt aus, woraufhin der Interrupt-Handler die Seite zurück in den Speicher liest. Die meisten Systeme erlauben es nur, einen geringen Prozentsatz des verfügbaren Arbeitsspeichers auszulagern. Ein Computer mit wenig physischem RAM hat eine große Auslagerungsdatei, die ständig genutzt wird. Mehr physischer RAM bedeutet dagegen weniger Notwendigkeit für den Zugriff auf die Auslagerungsdatei.

Wenn ein Programm etwas aus dem Speicher liest, muss es eine Adresse angeben. Diese Adresse muss für jeden Prozess in die tatsächliche physische Speicheradresse *übersetzt* werden. Das ist wichtig, da die vom Prozess verwendete Adresse *nicht* mit der physischen Adresse des Ortes identisch ist, an dem sich die Daten tatsächlich befinden. Daher ist eine Übersetzungsroutine notwendig, die den genauen physischen Speicherort feststellt.

Ein Beispiel: Wenn *Notepad.exe* nach Speicherinhalten unter der virtuellen Adresse `0x0041FF10` sucht, kann die tatsächliche physische Adresse `0x01EE2F10` lauten. Wird die Anweisung `mov eax, 0x0041FF10` ausgeführt, so ist der in `EAX` eingelesene Wert tatsächlich unter der physischen Adresse `0x01EE2F10` gespeichert. Die virtuelle Adresse wird in die physische übersetzt (siehe Abbildung 3.3).

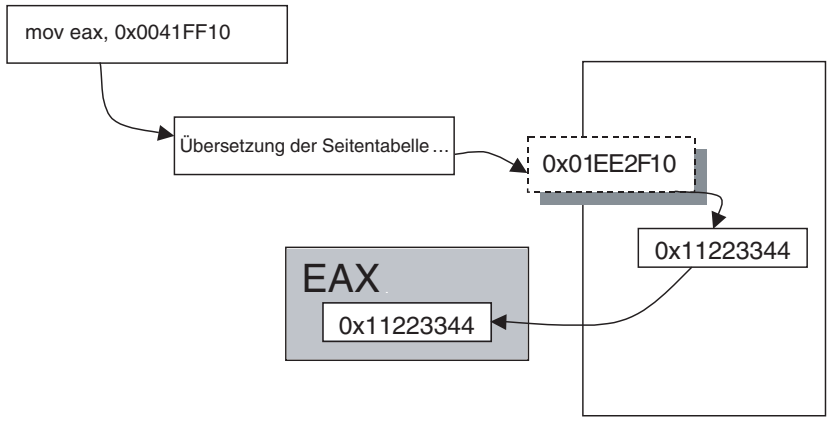


Abbildung 3-3 Übersetzung der Adresse bei einer mov-Anweisung

Seitentabellen-Lookups

Die Übersetzung von Speicheradressen erfolgt über eine besondere Tabelle, das *Seitentabellenverzeichnis*. Eine Intel x86-CPU speichert einen Zeiger auf dieses Verzeichnis in einem besonderen Register namens CR3. Dieses Register wiederum zeigt auf ein Array aus 1024 32-Bit-Werten, die das *Seitenverzeichnis* bilden. Jeder 32-Bit-Wert (*Seitenverzeichniseintrag*) gibt die Basisadresse einer Seitentabelle im physischen Speicher an und enthält ein Statusbit, das anzeigt, ob die Seitentabelle zurzeit im Speicher vorhanden ist. Aus der Seitentabelle können die tatsächlichen physischen Adressen abgelesen werden.

Abbildung 3.4 zeigt die verschiedenen Tabellen, die bei der Suche nach einer physischen Speicheradresse nachgeschlagen werden. Während dieses Lookups werden verschiedene Teile der in der Anforderung verwendeten Adresse (der *virtuellen Adresse*) genutzt. Abbildung 3.5 zeigt, wie die einzelnen Bestandteile verwendet werden.

Um die virtuelle Adresse in einer Anforderung in eine physische Speicheradresse zu übersetzen, führen das Betriebssystem und die CPU folgende Schritte durch:

- Die CPU schaut in CR3 nach, um die Basis des Seitentabellenverzeichnisses zu finden.
- Die virtuelle Adresse in der Anforderung wird, wie in Abbildung 3.5 gezeigt, in drei Teile aufgespaltet.
- Die ersten 10 Bits werden verwendet, um den Speicherort des Seitentabellenverzeichnisses zu finden (siehe Abbildung 3.4).

- Nachdem der Seitenverzeichniseintrag gefunden worden ist, wird die entsprechende Seitentabelle im Speicher ermittelt.
- Die mittleren 10 Bits der Adresse werden verwendet, um den Index in der Seitentabelle zu finden (siehe Abbildung 3.4).
- Die zugehörige physische Speicheradresse (Pageframe) für die Seite wird gefunden.
- Die letzten 12 Bits der Adresse werden verwendet, um den Offset im physischen Pageframe-Speicher zu bestimmen (bis zu 4096 Bytes). Die resultierende physische Adresse enthält die angeforderten Daten.

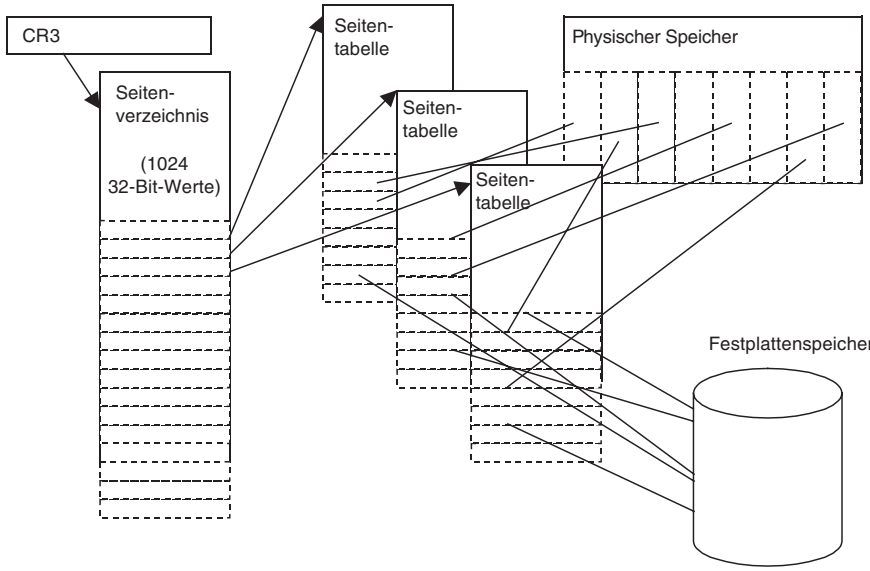


Abbildung 3-4 Eine Seite im Speicher finden

31 22	21 12	11 0
Index Seitenverzeichnis (1024 mögliche Werte)	Index Seitentabelle (1024 mögliche Werte)	Position auf der Seite (4096 mögliche Werte)

Abbildung 3-5 Die einzelnen Teile der Adresse in der Anforderung[2]

2. Bei einer 4-Mbyte-Seite geben die Bits 22–31 die Basisadresse der physischen Seite an und die Bits 0–21 den Offset auf dieser Seite.

Die in der Anforderung genannte Adresse wird als *virtuelle Adresse* bezeichnet – virtuell in dem Sinne, dass sie erst in eine reale (physische) Speicheradresse übersetzt werden muss, bevor sie verwendet werden kann. Wie Sie sehen, sind einige Tricks und Kniffe notwendig, um eine virtuelle Adresse in die tatsächliche physische Speicheradresse zu übersetzen. Jeder Schritt erfordert es, Informationen aus einer Tabelle abzurufen. Alle diese Daten können von einem Rootkit geändert oder genutzt werden.

Der Eintrag im Seitenverzeichnis

Wie gesagt, zeigt das Register CR3 auf die Basis des Seitenverzeichnisses. Dieses Verzeichnis ist ein Array von Seitenverzeichniseinträgen (siehe Abbildung 3.6). Bei Zugriff auf einen solchen Eintrag wird das U-Bit (Bit 2) überprüft. Wenn U auf 0 gesetzt ist, steht die zugehörige Seitentabelle nur für den Kernel zur Verfügung.

Auch das W-Bit (Bit 1) wird überprüft. Lautet es 0, so ist der zugehörige Speicher schreibgeschützt (also kein Lese/Schreib-Speicher). Denken Sie daran, dass ein Seitenverzeichniseintrag auf eine gesamte Seitentabelle zeigt (Abbildung 3.7) – also eine Sammlung von Seiten. Die Einstellungen in einem Seitenverzeichniseintrag gelten für einen gesamten Bereich von Speicherseiten.

Beachten Sie, dass das Programm, das das Seitenverzeichnis nachschlägt, in Ring 0 laufen muss.

31 12	11 9	8	7	6	5	4	3	2	1	0
Basisadresse der Seitentabelle		0	PS	0	A	PCD	PWT	U	W	P

Abbildung 3-6 Ein Seitenverzeichniseintrag

31 12	11 9	8	7	6	5	4	3	2	1	0
Basisadresse der Seite		0	0	D	A	PCD	PWT	U	W	P

Abbildung 3-7 Ein Seitentabelleneintrag[3]

3. Das Format des Seitentabelleneintrags kann je nach Betriebssystem unterschiedlich aussehen.

Der Eintrag in die Seitentabelle

Der Seitentabelleneintrag betrifft nur eine einzige Speicherseite. Auch hier wird das U-Bit geprüft, und wenn es auf 0 gesetzt ist, können nur Programme im Kernel-Mode auf die Seite zugreifen. Für den Lese/Schreib-Zugriff wird auch das W-Bit überprüft. Besonders zu beachten ist das P-Bit: Wenn es auf 0 gesetzt ist, so ist die Seite zurzeit ausgelagert (wohingegen sie beim Wert 1 im Speicher zur Verfügung steht). Ist eine Seite ausgelagert, muss der Speicher-Manager sie vor einem erfolgreichen Zugriff wieder in den Speicher zurückübertragen.

Schreibgeschützter Zugriff auf wichtige Tabellen

Bei Windows XP und höher sind die Speicherseiten mit der SSDT und der IDT in der Seitentabelle als schreibgeschützt gekennzeichnet. Wenn ein Angreifer den Inhalt dieser Seiten ändern will, muss er zunächst diesen Schreibschutz aufheben. Die beste Möglichkeit, dies mit einem Rootkit zu erreichen, ist der so genannte *CR0-Trick*, den wir weiter hinten in diesem Kapitel beschreiben. Sie können diese Tabellen aber auch beschreibbar machen, indem Sie zwei Registrierungsschlüssel ändern. Wenn Sie den Schreibschutz dauerhaft aufheben möchten, können Sie die folgenden Registrierungsschlüssel modifizieren und dann einen Neustart durchführen:[4]

```
HKLM\ SYSTEM\CurrentControlSet\Control\Session Manager\Memory
Management\EnforceWriteProtection = 0

HKLM\ SYSTEM\CurrentControlSet\Control\Session Manager\Memory
Management\DisablePagingExecutive = 1
```

(Der erste dieser beiden Schlüssel ist in einer frischen XP-Installation nicht vorhanden, sondern muss von Ihnen manuell hinzugefügt werden.)

Selbst wenn diese Registrierungsschlüssel nicht verändert werden, bilden sie keinen Schutz gegen Rootkits, da ein Rootkit die Seitentabellen direkt ändern oder den CR0-Trick einsetzen kann, um Zugriffseinschränkungen »on the fly« zu aktivieren oder zu deaktivieren.

Mehrere Prozesse, mehrere Seitenverzeichnisse

Theoretisch kann ein Betriebssystem mit nur einem Seitenverzeichnis mehrere Prozesse, die Speichertrennung zwischen den Prozessen und die Auslagerungsdatei auf der Festplatte verwalten. Allerdings gäbe es dann nur eine einzige Übersetzungszuordnung für den virtuellen Speicher. Das bedeu-

4. Vielen Dank an Rob Beck für diese Information.

tet, dass alle Prozesse denselben Speicher teilen müssten. Unter Windows NT/2000/XP/2003 hat jeder Prozess seinen eigenen Speicher. Eine gemeinsame Nutzung findet nicht statt.

Die Startadresse der meisten ausführbaren Dateien lautet `0x00400000`. Wie können mehrere Prozesse dieselbe virtuelle Adresse nutzen, ohne dass es zu Überschneidungen im physischen Speicher kommt? Die Lösung bilden mehrere Seitenverzeichnisse.

Jeder Prozess auf dem System unterhält ein eigenes Seitenverzeichnis und hat seinen eigenen, privaten Wert für das CR3-Register. Das bedeutet, dass jeder Prozess über eine eigene, eindeutige Zuordnung für den virtuellen Speicher verfügt. Damit können zwei Prozesse auf die Speicheradresse `0x00400000` zugreifen und sie in zwei verschiedene physische Speicheradressen übersetzen lassen. Das ist auch der Grund, weshalb ein Prozess den Speicher eines anderen nicht »sehen« kann.

Selbst wenn jeder Prozess seine eigene Seitentabelle führt, wird der Speicher oberhalb von `0x7FFFFFFF` gewöhnlich über alle Prozesse hinweg gleich zugeordnet. Dieser Speicherbereich ist für den Kernel reserviert, und der Kernelspeicher bleibt konsistent, unabhängig davon, welcher Prozess gerade ausgeführt wird.

Auch wenn ein Prozess in Ring 0 ausgeführt wird, gibt es einen *Kontext für den aktiven Prozess*. Dieser Kontext enthält den Rechnerstatus für den Prozess (z.B. die gespeicherten Register), die Umgebung und das Sicherheitstoken des Prozesses sowie andere Parameter. Für unsere Zwecke reicht es, sich vorzustellen, dass der Kontext das CR3-Register enthält und damit das Seitenverzeichnis des aktiven Prozesses. Ein Rootkit-Entwickler sollte daran denken, dass Änderungen an den Seitentabellen eines Prozesses nicht nur den Prozess im User-Mode beeinflussen, sondern auch den Kernel, wenn der Prozess im Kontext läuft. Das lässt sich für erweiterte Tarntechniken nutzen.

Prozesse und Threads

Rootkit-Entwickler müssen wissen, dass der grundlegende Mechanismus für die Verwaltung von laufendem Code der Thread ist, nicht der Prozess. Der Windows-Kernel führt die Zeitplanung der Prozesse aufgrund der Anzahl der Threads durch, nicht der Prozesse. Wenn es zwei Prozesse gibt, von denen der eine nur einen, der andere aber neun Threads aufweist, so erhält der Prozess mit einem Thread 10% der CPU-Zeit, der andere dagegen 90%. Dieses Beispiel ist natürlich stark vereinfacht, da bei dieser Zeitplanung auch noch andere Faktoren eine Rolle spielen (etwa die Priorität). Tatsache aber ist, dass die Zeitplanung vollständig auf der Anzahl der Threads und nicht der Prozesse beruht, wenn alle anderen Faktoren gleich sind.

Aber was ist ein *Prozess*? Unter Windows, handelt es sich bei einem Prozess einfach um eine Möglichkeit, über die eine Gruppe von Threads die folgenden Daten gemeinsam nutzen kann:

- Virtueller Adressraum (also der Wert für CR3)
- Zugriffstoken einschließlich SID[5]
- Handle-Tabelle für Win32-Kernelobjekte
- Arbeitssatz (physischer Speicher im »Besitz« des Prozesses)

Aus mehreren Gründen müssen sich Rootkits um Threads und Threadstrukturen kümmern, unter anderem zur Tarnung und für die Einführung von Code. Anstatt neue Prozesse zu erstellen, kann ein Rootkit neue Threads anlegen und sie bestehenden Prozessen zuweisen. Nur selten muss ein komplett neuer Prozess eingerichtet werden.

Wenn ein Kontextwechsel zu einem neuen Thread auftritt, wird der alte Threadstatus gespeichert. Jeder Thread hat seinen eigenen Kernelstack, wobei der Threadstatus an der Spitze dieses Stacks platziert wird. Wenn der neue Thread zu einem anderen Prozess gehört, wird eine neue Seitenverzeichnisadresse in CR3 geladen. Diese Adresse wird aus der Struktur `KPROCESS` des Prozesses ermittelt. Nachdem der neue Thread-Kernelstack gefunden worden ist, wird der alte Status des neuen Threads von der Spitze dieses Stacks entfernt, woraufhin der neue Thread mit der Ausführung beginnt. Wenn ein Rootkit die Seitentabellen eines Prozesses ändert, gilt diese Änderung für alle Threads dieses Prozesses, da sie alle denselben CR3-Wert nutzen.

In Kapitel 7, *Direkte Manipulation von Kernel-Objekten*, schauen wir uns Thread-und Prozessstrukturen ausführlicher an.

Die Deskriptortabellen des Speichers

Einige der Tabellen, die die CPU verwendet, um sich auf dem Laufenden zu halten, können Deskriptoren enthalten. Es gibt verschiedene Arten von Deskriptoren, die aber alle von einem Rootkit eingefügt oder geändert werden können.

Die globale Deskriptortabelle

Eine Reihe interessanter Tricks lässt sich über die GDT durchführen. Diese Tabelle wird verwendet, um verschiedene Adressbereiche zuzuordnen, kann aber auch eingesetzt werden, um Taskwechsel durchzuführen. Die Basisadresse der GDT wird mithilfe der Anweisung `SGDT` ermittelt. Sie können den Speicherort der GDT über die Anweisung `LGDT` ändern.

5. Ein Thread kann auch sein eigenes Zugriffstoken aufweisen, das dann das Token des Prozesses überschreibt.

Die lokale Deskriptortabelle

Die LDT erlaubt einem Task einen eigenen Satz eindeutiger Deskriptoren. Das so genannte *Tabellenindikator-Bit* wählt dann zwischen der GDT und der LDT, wenn ein Segment angegeben wird. Die LDT kann dieselben Arten von Deskriptoren enthalten wie die GDT.

Codesegmente

Beim Zugriff auf Codespeicher verwendet die CPU das im Codesegment-Register (CS) angegebene Segment. Ein Codesegment kann in der Deskriptortabelle festgelegt werden. Jedes Programm, auch ein Rootkit, kann das CS-Register durch einen Befehl für einen *fernen Aufruf*, *fernen Sprung* oder eine *ferne Rückgabe* (*far call, far jump* bzw. *far return*) ändern, wodurch CS von der Spitze des Stacks entfernt wird.[6] Bemerkenswerterweise können Sie Ihren Code einfach dadurch ausführen, dass Sie das R-Bit im Deskriptor auf 0 setzen.

Call-Gates

In der LDT oder der GDT kann sich auch eine besondere Art von Deskriptor befinden, nämlich ein Call-Gate. Ein Programm kann einen fernen Aufruf durchführen, bei dem der Deskriptor auf das Call-Gate gesetzt ist. Bei einem Aufruf kann dann ein neuer Ring angegeben werden. Mit Call-Gates ist es möglich, dass User-Mode-Programme Funktionsaufrufe im Kernel-Mode vornehmen, was eine verlockende Hintertür für Rootkit-Programme darstellt. Derselbe Mechanismus kann auch bei einem entfernten Sprung eingesetzt werden, aber nur dann, wenn das Call-Gate dieselben oder geringere Privilegien aufweist wie der Prozess, der den Sprung durchführt.[7]

Bei der Verwendung eines Call-Gates wird die Adresse ignoriert – nur die Deskriptornummer ist von Bedeutung. Die Datenstruktur des Call-Gates teilt der CPU mit, wo sich der Code für die aufgerufene Funktion befindet. Optional können Argumente vom Stack gelesen werden. So lässt sich z.B. ein Call-Gate erstellen, bei dem der Aufrufende geheime Befehlsargumente auf den Stack legt.

Die Interrupt-Deskriptortabelle

Das *Interrupt-Deskriptortabellenregister* (IDTR) enthält die Basisadresse (Startadresse) der *Interrupt-Deskriptortabelle* (IDT) im Arbeitsspeicher. Die IDT wird dazu verwendet, die Softwarefunktion zu finden, die zum Umgang

6. Sie können auch eine IRET-Anweisung verwenden.
7. Eine Ausnahme besteht bei einem weiten Sprung zu einem »zugehörigen« Codesegment.

mit einem Interrupt gebraucht wird, und ist daher sehr wichtig.[8] Interrupts werden für verschiedene maschinennahe Funktionen eines Computers verwendet. So wird z.B. immer dann ein Interrupt ausgelöst, wenn eine Taste gedrückt wird.

Die IDT ist ein Array mit 256 Einträgen – einem für jeden Interrupt. Das bedeutet, dass jeder Prozessor bis zu 256 Interrupts verwenden kann. Da jeder Prozessor sein eigenes IDTR hat, verfügt er auch über seine eigene Interrupttabelle. Bei Rootkits für Computer mit mehreren CPUs müssen Sie bedenken, dass jeder dieser Prozessoren eine eigene Interrupttabelle hat.

Wenn ein Interrupt auftritt, wird seine Nummer aus der Interrupt-Anweisung oder vom programmierbaren Interrupt-Controller (PIC) gewonnen. In jedem Fall wird die Interrupttabelle herangezogen, um die Softwarefunktion zu finden, die nun aufgerufen werden soll. Diese Funktion wird auch als *Vektor* oder *Interrupt-Dienstroutine* (Interrupt Service Routine, ISR) bezeichnet.

Befindet sich der Prozessor im geschützten Modus, so ist die Interrupttabelle ein Array aus 256 8-Byte-Einträgen. Jeder Eintrag besteht aus der Adresse der ISR und einigen anderen sicherheitsrelevanten Informationen.

Um die Adresse der Tabelle im Speicher zu finden, müssen Sie das IDTR lesen, was Sie durch die Anweisung `SIDT` erledigen (Store Interrupt Descriptor Table). Mit der Anweisung `LIDT` (Load Interrupt Descriptor Table) können Sie den Inhalt des IDTR auch ändern. Nähere Einzelheiten zu dieser Technik finden Sie in Kapitel 8.

Ein in Rootkits angewendeter Trick besteht darin, eine neue Interrupttabelle zu erstellen, um Änderungen an der ursprünglichen Tabelle zu verbergen. Ein Virenscanner überprüft zwar die Integrität der ursprünglichen IDT, aber das Rootkit kann eine Kopie davon anlegen, das IDTR ändern und dann die kopierte IDT fröhlich ändern, ohne entdeckt zu werden.

Die `SIDT`-Anweisung speichert den Inhalt der IDTR im folgenden Format:

```
/* SIDT gibt die IDT in diesem Format zurück. */
typedef struct
{
   unsigned short IDTLimit;
   unsigned short LowIDTbase;
   unsigned short HiIDTbase;
}  IDTINFO;
```

8. Damit die CPU mit Interrupts umgehen kann, muss außerdem das IF-Bit in ihrem EFlags-Register gesetzt sein.

Mit den durch die Anweisung `SIDT` zur Verfügung gestellten Daten kann ein Angreifer die Basis der IDT finden und einen Dump von ihrem Inhalt erstellen.

Denken Sie daran, dass die IDT bis zu 256 Einträge umfassen kann, wobei jeder dieser Einträge einen Zeiger auf eine Interrupt-Dienstroutine enthält. Die Einträge weisen die folgende Struktur auf:

```
// Eintrag in der IDT, wird manchmal auch als "Interrupt-Gate" bezeichnet
#pragma pack(1)
typedef struct
{
      unsigned short LowOffset;
      unsigned short selector;
      unsigned char unused_lo;
      unsigned char segment_type:4;        // 0x0E ist ein Interrupt-Gate
      unsigned char system_segment_flag:1;
      unsigned char DPL:2;                 // Aktueller DPL (Descriptor
                                           // Privilege Level)
      unsigned char P:1;
      unsigned short HiOffset;
}  IDTENTRY;
#pragma pack()
```

Diese Datenstruktur, ein so genanntes *Interrupt-Gate*, wird verwendet, um die Funktion im Speicher zu finden, die sich des Interrupt-Ereignisses annimmt. Mithilfe eines Interrupt-Gates kann ein User-Mode-Programm Routinen im Kernel-Mode aufrufen. So liegt das Ziel eines Interrupts für einen Systemaufruf z. B. beim Offset `0x2E` in der IDT-Tabelle.

Ein Systemaufruf kann zwar im User-Mode ausgelöst werden, wird aber im Kernel-Mode verarbeitet. Ein Rootkit kann zusätzliche Interrupt-Gates als Hintertür platzieren oder an vorhandenen Interrupt-Gates einen Hook anbringen.

Der folgende Beispielcode zeigt, wie Sie Zugriff auf die IDT erlangen können:

```
#define MAKELONG(a, b)
((unsigned long) (((unsigned short) (a)) | ((unsigned long) ((unsigned short) (b))) <<
16))
```

Die Höchstzahl von Einträgen in der IDT beträgt 256.

```
#define MAX_IDT_ENTRIES 0xFF
```

Zur Veranschaulichung implementieren wir einen Parser innerhalb der `DriverEntry`-Routine unseres Beispiel-Rootkits.

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT theDriverObject,
                     IN PUNICODE_STRING theRegistryPath )
{
      IDTINFO   idt_info;    // Dieser Struktur wird durch den Aufruf von
                             // STORE IDT (sidt) erhalten.
      IDTENTRY* idt_entries; // Anschließend wird dieser Zeiger aus
                             // idt_info erhalten.
      unsigned long count;

      // idt_info laden
      __asm sidt, idt_info
```

Wir verwenden die von der `SIDT`-Anweisung zurückgegebenen Daten, um die Basis der IDT zu ermitteln. Anschließend bewegen wir uns in einer Schleife durch die einzelnen Einträge und schicken einige Daten an die Debug-Ausgabe.

```
      idt_entries = (IDTENTRY*)
MAKELONG(idt_info.LowIDTbase,idt_info.HiIDTbase);
      for(count = 0;count <= MAX_IDT_ENTRIES;count++)
      {
            char _t[255];
            IDTENTRY *i = &idt_entries[count];
            unsigned long addr = 0;
            addr = MAKELONG(i->LowOffset, i->HiOffset);

            _snprintf(_t,
                      253,
                   "Interrupt %d: ISR 0x%08X", count, addr);
            DbgPrint(_t);
      }
      return STATUS_SUCCESS;
}
```

Dieser Code zeigt, wie Sie die IDT parsen. Hierbei werden noch keine Änderungen an der Tabelle vorgenommen, allerdings kann dieses Beispiel sehr gut die Grundlage für einen komplexeren Code sein.

Ausgefeiltere Verfahren mit Interrupts werden in den Kapiteln 5 und 8 behandelt.

Andere Arten von Gates

Neben Interrupt- kann die IDT auch *Task-* und *Trap-Gates* enthalten. Ein Trap-Gate unterscheidet sich von einem Interrupt-Gate nur dadurch, dass es durch maskierbare Interrupts unterbrochen werden kann. Ein Task-Gate dagegen ist ein eher altmodisches Prozessormerkmal, das dazu verwendet werden kann, einen x86-Taskwechsel herbeizuführen. Da diese Funktion von Windows nicht genutzt wird, geben wir dafür kein Beispiel an.

Ein *Task* sollte nicht mit einem *Prozess* unter Windows verwechselt werden. Ein Task für die x86-CPU wird vom Task Switch Segment (TSS) verwaltet – einer Vorrichtung, die ursprünglich dazu da war, um Tasks mithilfe der Hardware zu verwalten. Linux, Windows und viele andere Betriebssysteme implementieren Taskwechsel in der Software und nutzen den zugrunde liegenden Hardwaremechanismus praktisch gar nicht.

Die System Service Dispatch Table (SSDT)

Die SSDT wird verwendet, um Funktionen nachzuschlagen, die zur Handhabung eines Systemaufrufs erforderlich sind. Diese Vorkehrung ist im Betriebssystem implementiert, nicht in der CPU. Es gibt zwei Wege, auf denen ein Programm einen Systemaufruf durchführen kann: durch die Verwendung des Interrupts `0x2E` und die Anweisung `SYSENTER`.

Bei Windows XP und höher verwenden Programme gewöhnlich `SYSENTER`, während ältere Plattformen den Interrupt `0x2E` einsetzen. Die beiden Mechanismen sind vollständig verschieden, führen aber zum selben Ergebnis.

Ein Systemaufruf führt dazu, dass die Funktion `KiSystemService` im Kernel aufgerufen wird. Sie liest die Nummer des Systemaufrufs aus dem EAX-Register und schlägt den Aufruf in der SSDT nach. Dabei kopiert sie auch die Argumente des Systemaufrufs vom User- in den Kernel-Mode-Stack. Auf diese Argumente zeigt das EAX-Register. Manche Rootkits bringen an dieser Verarbeitungskette einen Hook an, um Daten auszuspionieren, Datenargumente zu ändern oder den Systemaufruf umzuleiten. Diese Technik beschreiben wir ausführlich in Kapitel 4.

Die Steuerregister

Neben den Systemtabellen steuern auch einige besondere Register wichtige Funktionen der CPU. Diese Register können von Rootkits verwendet werden.

Steuerregister 0 (CR0)

Das Steuerregister enthält Bits, die festlegen, wie sich der Prozessor verhält. Eine beliebte Methode zur Deaktivierung von Speicherzugriffseinschränkungen für den Kernel besteht darin, das Steuerregister CR0 zu ändern.

Dieses Steuerregister wurde zuerst in den frühen 286er-Prozessoren eingeführt und zunächst *Maschinenstatuswort* genannt. Mit der Veröffentlichung der 386er-Prozessorfamilie wurde es in *Control Register Zero* (CR0) umbenannt. Erst mit der 486er-Prozessorserie wurde das WP-Bit für den Schreibschutz (Write Protect) dem Register hinzugefügt. Es gibt an, ob der Prozessor Schreibvorgänge auf Speicherseiten zulässt, die als schreibgeschützt gekennzeichnet sind. Wird das WP-Bit auf 0 gesetzt, so wird damit der Speicherschutz deaktiviert. Das ist sehr wichtig für Kernel-Rootkits, die in Datenstrukturen des Betriebssystems schreiben sollen.

Der folgende Code zeigt, wie Sie den Speicherschutz mit dem CR0-Trick deaktivieren und reaktivieren:

```
// Speicherschutz deaktivieren
__asm
{
      push eax
      mov  eax, CR0
      and  eax, 0FFFEFFFFh
      mov  CR0, eax
      pop  eax
}
// Beliebige Vorgänge
// Speicherschutz reaktivieren
__asm
{
      push eax
      mov  eax, CR0
      or   eax, NOT 0FFFEFFFFh
      mov  CR0, eax
      pop  eax
}
```

Andere Steuerregister

Es gibt vier weitere Steuerregister für andere Funktionen des Prozessors. CR1 wird nicht genutzt und ist undokumentiert. CR2 wird verwendet, wenn sich der Prozessor im geschützten Modus befindet; es speichert die letzte Adresse, die einen Seitenfehler hervorgerufen hat. CR3 speichert die Adresse des Seitenverzeichnisses. CR4 wurde erst mit den Pentium-Prozessoren implementiert (und mit späten Versionen des 486ers); es kümmert sich um Vorgänge wie die Aktivierung des virtuellen 8086-Modus, der verwen-

det wird, wenn ein altes DOS-Programm unter Windows NT ausgeführt wird. In diesem Modus fängt der Prozessor privilegierte Anweisungen wie `CLI`, `STI` und `INT` ab. Diese zusätzlichen Register sind für Rootkits normalerweise nicht nützlich.

Das EFlags-Register

Das EFlags-Register ist ebenfalls wichtig. Erstens handhabt es das *Trap-Flag*. Wenn dieses Flag gesetzt ist, arbeitet der Prozessor im Einzelschrittmodus. Ein Rootkit kann Funktionen wie den Einzelschrittmodus verwenden, um herauszufinden, ob ein Debugger läuft, oder um sich vor einem Virenscanner zu verstecken. Sie können Interrupts deaktivieren, indem Sie das *Interrupt-Flag* löschen. Außerdem können Sie die Einstellung für die Ebene der E/A-Privilegien verwenden, um ringgestützte Schutzsysteme zu ändern, die von den meisten Betriebssystemen für die Intel-Plattform genutzt werden.

Multiprozessor-Systeme

Multiprozessor-Systeme (manchmal auch als symmetrische Multiprozessor-Systeme [SMP] bezeichnet) und Hyperthread-Systeme warten mit ganz eigenen Problemstellungen auf. Die Hauptschwierigkeit, vor die sie Rootkit-Entwickler stellen, ist die Synchronisation. Wenn Sie Multithread-Anwendungen geschrieben haben, wissen Sie (hoffentlich!) etwas über Threadsicherheit und darüber, was geschehen kann, wenn zwei Threads zur selben Zeit auf ein Datenobjekt zugreifen. Wenn nicht, sollte es an dieser Stelle ausreichen zu wissen, dass das Datenobjekt in diesem Fall beschädigt wird. Das ist so wie mit den zu vielen Köchen und dem Brei.

Multiprozessor-Systeme ähneln in gewisser Hinsicht Multithread-Umgebungen, da Code auf zwei oder mehr CPUs gleichzeitig ausgeführt werden kann. Kapitel 7, Direkte Manipulation von Kernel-Objekten, behandelt die Multiprozessor-Synchronisation.

Abbildung 3.8 zeigt das Layout eines typischen Multiprozessor-Systems: Mehrere CPUs teilen sich den Zugriff auf einen einzigen Speicherbereich, einen Satz von Controllern und eine Gruppe von Geräten.

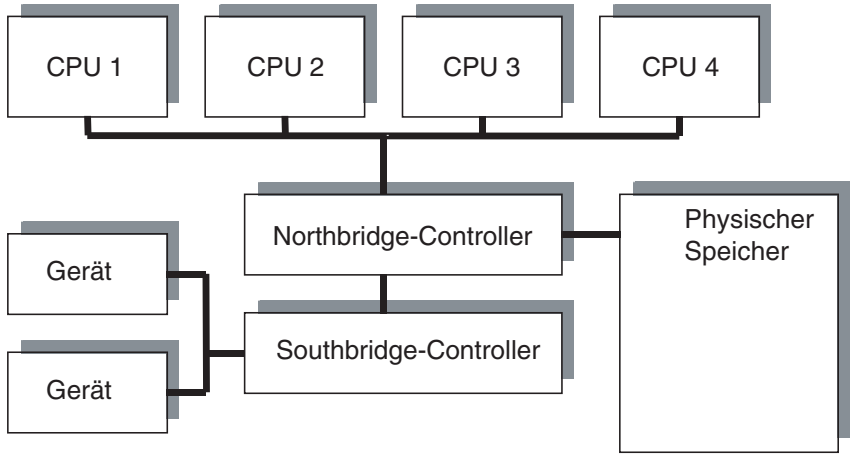


Abbildung 3-8 Typisches Layout eines Multiprozessor-Busses

Einige Punkte müssen Sie sich zum Thema Multiprozessor-Systeme merken:

- Jede CPU hat ihre eigene Interrupttabelle. Wenn Sie einen Hook an der Interrupttabelle anbringen möchten, dann denken Sie daran, das bei allen CPUs zu machen! Wenn Sie das nicht tun, gilt Ihr Hook nur für eine einzige CPU. Das kann durchaus Absicht sein, wenn Sie keine 100%ige Kontrolle über einen Interrupt brauchen – aber so etwas ist nur selten der Fall.
- Ein Treiber, der auf einem Einzelprozessor-System gut läuft, kann auf einem Multiprozessor abstürzen (Bluescreen). Beim Testen müssen Sie Multiprozessor-Systeme einbeziehen.
- Multiprozessor-Systeme stellen Interlock-Routinen, Spinlocks und Mutexe bereit. Das sind Tools des Systems, die dabei helfen, den Zugriff auf die Daten zu synchronisieren. Einzelheiten über ihre Verwendung finden Sie in der DDK-Dokumentation.
- Implementieren Sie keine eigenen Sperrmechanismen, sondern verwenden Sie die Tools, die das System bereitstellt. Wenn Sie diese Mechanismen wirklich selbst erstellen müssen, machen Sie sich mit den Speicherbarrieren (`KeMemoryBarrier` usw.) und der Hardware-Umordnung der Anweisungen vertraut. Diese Themen würden jedoch den Rahmen dieses Buches sprengen.
- Stellen Sie fest, auf welchem Prozessor das Rootkit ausgeführt wird. Sie können einen Aufruf wie `KeGetCurrentProcessorNumber` verwenden, um zu

bestimmen, auf was für einem Prozessor Ihr Code gerade läuft. Sie können auch mit `KeGetActiveProcessors` feststellen, über wie viele aktive Prozessoren das System verfügt.

- Erzwingen Sie die Ausführung auf einem bestimmten Prozessor. Sie können Code so einrichten, dass er auf einem bestimmten Prozessor läuft. Schauen Sie in der DDK-Dokumentation unter `KeSetTargetProcessorDPC` nach.

Zusammenfassung

Dieses Kapitel hat Sie in die Hardwaremechanismen eingeführt, die hinter den Kulissen ablaufen, um die Betriebssystemfunktionen für Sicherheit und Speicherzugriff durchzusetzen. Sie haben auch einige Einzelheiten über die Verwendung der Interrupttabelle erhalten. Auf der Grundlage dieses Wissens können Sie ein tiefes Verständnis der Computermanipulation gewinnen. Da letztlich die Hardware für die Implementierung der Software verantwortlich ist, unterliegt die gesamte Software den Manipulationen, die Sie auf der Ebene der Hardware angebracht haben. Ein gründliches Verständnis dieser Vorgänge ist der Ausgangspunkt für die geschickte Anwendung von Rootkits und die Fähigkeit, jegliche andere Software auf dem System zu unterwandern.