Diplomarbeit

Robuste Integration kryptographischer Anwendungen in Java und Windows 2000

Marcel Winandy

21. Juni 2002

Institut für Informatik III
Rheinische Friedrich-Wilhelms-Universität Bonn

Für Brigitte
für ihre Liebe und ihr Verständnis

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

> *If I take a letter and lock it in a safe, and then give you the safe along with the design specifications of the safe and a hundred identical safes with their combinations so that you and the world's best safecrackers can study the locking mechanism – and you still can't open the safe and read the letter – that's security.*
>
> Bruce Schneier [Sch96]

Die vertrauliche und unverfälschte Übermittlung von Nachrichten dürfte eines der ältesten Probleme der Menschheitsgeschichte sein. Während es früher vorwiegend nur im militärischen Bereich darum ging, Nachrichten an die eigenen Truppen zu schicken, ohne daß der Feind sie im Klartext lesen konnte, müssen heute auch im privaten und kommerziellen Bereich Daten vor unbefugtem Zugriff und Manipulation geschützt werden. Dies gilt sowohl für Menschen als auch für Computersysteme. Eine Geld-Überweisung soll korrekt durchgeführt, ein elektronisch abgeschlossener Kaufvertrag nicht nachträglich manipuliert und eine private E-Mail nur vom Empfänger gelesen werden können.

Kryptographische Anwendungen ermöglichen es, Daten vertraulich und unter Bewahrung der Integrität zu speichern oder an Kommunikationspartner zu übermitteln. Eine Verschlüsselung der Daten soll ausschließen, daß Parteien sie lesen können, die nicht im Besitz des Schlüssels sind. Die Signierung der Daten soll verhindern, daß sie unbemerkt manipuliert werden können. Eine Modifikation würde durch die Verifikation der Signatur erkannt werden.

Das Ziel der Integrität und der Vertraulichkeit kann jedoch nur erreicht werden, wenn die verwendeten kryptographischen Algorithmen stark genug sind, so daß die Sicherheit allein auf der Länge der Schlüssel beruht und ein Angreifer die Schlüssel nicht in angemessener Zeit berechnen kann. Letzteres impliziert, daß auch das Software-Design der Anwendung und die Integration ins Computersystem robust gegenüber Angriffen sein muß. Erlangt ein Angreifer in irgendeiner Form Zugriff auf geheime Schlüssel oder die Daten der Schlüsselgenerierung, kann er verschlüsselte Nachrichten entschlüsseln und Signaturen fälschen.

Wenn man davon ausgehen kann, daß das Verhalten des Benutzers vertrauenswürdig ist, er die kryptographische Anwendung unverändert erhält und in einer vertrauenswürdigen Umgebung ausführt, ist dieses System bezogen auf Integrität und Vertraulichkeit keinen Bedrohungen ausgesetzt. Eine Untersuchung zusätzlicher Schutzmaßnahmen im Software-Design ist nicht nötig.

Muß man aber davon ausgehen, daß eine Installation zusätzlicher, nichtvertrauenswürdiger Programme erlaubt ist, besteht die Gefahr, daß Trojanische Pferde (Trojaner) auf das System gelangen. Ein Trojaner ist ein Programm, das eine (scheinbar) nützliche Funktion ausführt, aber auch versteckt eine schädliche Funktion enthält.

Nachdem eine kryptographische Anwendung auf einem produktiven System installiert wurde, ist sie folglich mehreren Bedrohungen ausgesetzt. Einige Bedrohungen ergeben sich direkt aus der verwendeten Programmiersprache, mit der die Anwendung entwickelt wurde. Ein nachträglich installierter Trojaner könnte versuchen, einen direkten Angriff auf die Anwendung durchzuführen, indem er Schwächen im Software-Design ausnutzt. Es gilt zu verhindern, daß der Trojaner durch geschicktes Ausnutzen von verschiedenen Funktionsaufrufen der Anwendung an den privaten Schlüssel des Anwenders gelangt oder an Daten, die zur Schlüsselgenerierung verwendet wurden. Andere Bedrohungen betreffen die Ausführungsumgebung, d.h. das Betriebssystem, in dem die Anwendung eingesetzt wird. Ein Trojaner könnte wichtige Systemkomponenten austauschen, um beispielsweise Betriebssystemaufrufe der Anwendung abzufangen und durch andere zu ersetzen.

Diese Arbeit beschäftigt sich mit der Frage, wie eine kryptographische Anwendung robust in Java und Microsoft Windows 2000 integriert werden kann. Java wurde als Vertreter einer modernen Programmiersprache gewählt, da es eine zunehmende Bedeutung sowohl bei Server- als auch bei Desktop-Anwendungen erfährt. Neben der Objekt-Orientierung bietet Java vor allem eingebaute Sicherheitskonzepte. So werden durch die Typsicherheit und den Verzicht auf Zeigerarithmetik automatisch einige Programmierfehler vermieden, die sonst zu Sicherheitslücken führen könnten. Java bietet darüber hinaus eine flexible Zugriffskontrolle, die dafür sorgt, daß ein bestimmter Programmcode nur auf bestimmte Ressourcen zugreifen darf. Es wird untersucht, welche Möglichkeiten einem Entwickler dadurch geboten werden, um die Integrität der Anwendung zu bewahren oder zumindest Manipulationen erkennen zu können.

Windows 2000 ist ein modernes Betriebssystem mit relativ weiter Verbreitung, sein Vorgänger *Windows NT* wurde und wird in vielen Unternehmensnetzwerken eingesetzt. Durch den Nachfolger *Windows XP* dürften die Konzepte und die Architektur auch vermehrt auf Heimrechnern Einzug halten. Windows 2000 unterscheidet zwischen *Benutzermodus* und *Kernelmodus*. Anwendungen, die im Kernelmodus laufen, haben uneingeschränkten Zugriff auf die Hardware und auf Daten im Systemspeicher. Anwendungen im Benutzermodus haben dagegen keinen di-

rekten Zugriff, sie müssen Betriebssystemfunktionen aufrufen, die mit einer Zugriffskontrolle geschützt werden.

Im Kernelmodus laufen neben den wichtigsten Betriebssystembestandteilen von Windows 2000 auch Treiber. Treiber dienen zur Ansteuerung von Hardware-Komponenten und können nachträglich dem System hinzugefügt werden. Ein Trojaner in Form eines Treibers könnte alle Sicherheitsmaßnahmen des Betriebssystems umgehen und Betriebssystemaufrufe von anderen Anwendungen abfangen oder manipulieren.

Ein Computersystem wird oft gleichzeitig für verschiedene Aufgaben verwendet, beispielsweise zur Textverarbeitung, zum „Surfen“ im Internet, zur Darstellung von Multimedia-Inhalten und zur Ausführung von kryptographischen Anwendungen. Diese Aufgaben haben jedoch unterschiedliche Anforderungen an die Sicherheit des Systems. Eine zusätzliche Programminstallation erfordert deshalb häufig administrative Privilegien. Mit dem Programm könnten auch Treiber installiert oder wichtige Systemeinstellungen verändert werden. Auf Einzelplatzrechnern (vor allem zu Hause) ist der Benutzer jedoch meistens gleichzeitig der Administrator oder kann sich mit administrativen Privilegien einloggen. Damit besteht die Gefahr, daß Trojanische Pferde installiert und mit privilegierten Rechten ausgeführt werden.

Überblick

Kapitel 2 stellt grundlegende Modelle der Computersicherheit vor und liefert einen Überblick über bisherige Konzepte sicherer Softwareentwicklung. Auf die Architektur und die Schutzmöglichkeiten von Java und Windows 2000 wird näher eingegangen. Kapitel 3 beschreibt das bleibende Bedrohungsszenario. In Kapitel 4 werden Schutzmöglichkeiten von Java für eine integritätsbewahrende Softwareentwicklung analysiert. Im Rahmen dieser Arbeit wurde ein Integritätsprüfprogramm als Windows 2000 Dienst entwickelt, der in Kapitel 5 vorgestellt wird. Die Manipulationsfestigkeit von Diensten wird untersucht, und spezifische Bedrohungen werden identifiziert. Es wird ein Ansatz präsentiert, um die Integrität des Systems bei der Installation nichtvertrauenswürdiger Programme zu bewahren.

Im Rahmen dieser Arbeit wurde eine Java-Implementation des Standards IEEE P1363 für *Public-Key*-Kryptographie entwickelt. Kapitel 6 zeigt die Architektur einer darauf basierenden kryptographischen Anwendung und ihre Integration ins Betriebssystem Windows 2000. Eine Architektur zur exklusiven Kommunikation von Programmen wird skizziert und ein Ansatz beschrieben, wie sie in Java und Windows 2000 implementiert werden kann. Abschließend liefert Kapitel 7 eine Zusammenfassung und einen Ausblick über die Thematik.

2. Bisherige Konzepte sicherer Softwareentwicklung

2.1. Theoretische Ansätze

2.1.1. Das Bell-LaPadula-Modell

Das *Bell-LaPadula-Modell* (kurz BLP) ist ein formales Zugriffskontrollmodell [Lee99, Lan81]. Allgemein gibt es zwei Formen der Zugriffskontrolle von Subjekten (Benutzern[1]) auf Objekte (Daten): eine Zugriffkontrolle nach freiem Ermessen der Besitzer von Objekten (*Discretionary Access Control*, DAC) und eine verbindliche Zugriffskontrolle (*Mandatory Access Control*, MAC), in der eine Sicherheitspolitik bestimmt, wer wie Zugriff auf welche Objekte haben darf. Eine *Zugriffsmatrix* definiert, welche Subjekte auf welche Objekte welchen Zugriff haben. In einem DAC-System können Subjekte die Zugriffsrechte für Objekte, die sie besitzen, ändern, in einem MAC-System dagegen nicht.

BLP ist ein Modell für ein MAC-System. Die zeitliche Änderung des Systems wird als Folge von Zuständen modelliert. Zustände korrespondieren zu sicherheitsbezogenen Ereignissen, wie z.B. die Änderung der Zugriffsmatrix. Regeln spezifizieren die Aktionen, die das System von einen Zustand in den anderen überführen. Das gesamte System gilt als sicher, wenn es in allen Zuständen sicher ist. Das Modell definiert formal, wann ein Zustand sicher ist, und beschreibt, welche Zustandsübergänge erlaubt sind, so daß von einem sicheren Zustand niemals ein unsicherer Zustand erreicht werden kann.

Vereinfacht dargestellt ist ein Systemzustand ein Paar (B, f), wobei B die Menge der aktuell möglichen Zugriffe ist. Formal ist B eine Teilmenge von $S \times O \times A$, wobei S die Menge der Subjekte, O die Menge der Objekte und $A = \{r, w, x, a\}$ die Menge der Zugriffsformen gemäß folgender Tabelle ist:

[1] Subjekte sind nicht direkt mit Benutzern identisch, sie stellen nur eine angemeldete Sitzung eines Benutzers dar. Ein Benutzer kann folglich die Identität mehrerer Subjekte annehmen.

Zugriffsform	Objekt lesbar	Objekt veränderbar
execute (x)	nein	nein
read (r)	ja	nein
append (a)	nein	ja
write (w)	ja	ja

f ist die *Sicherheitsfunktion*, die jedem Subjekt und Objekt eine Sicherheitsstufe zuordnet: $f : S \cup O \rightarrow L$. Auf L existiert eine Ordnung $\leq$, so daß sich die Sicherheitsstufen von Subjekten und Objekten miteinander vergleichen lassen. Beispielsweise wäre im militärischen Bereich $L = \{unclassified, confidential, secret, topsecret\}$.

Ein Systemzustand (B, f) ist genau dann *sicher*, wenn er folgende Eigenschaften erfüllt:

1. *simple security property*:

   a) $(s, o, x) \in B \Rightarrow f(s) \leq f(o)$

   b) $(s, o, r) \in B \Rightarrow f(s) \geq f(o)$

2. **-property*[2]:

   a) $(s, o, a) \in B \Rightarrow f(s) \leq f(o)$

   b) $(s, o, w) \in B \Rightarrow f(s) = f(o)$

Die erste Eigenschaft verhindert, daß Subjekte Programme mit einer niedrigeren Sicherheitsstufe ausführen oder Daten einer höheren Stufe lesen können („*no read-up*“). Die zweite Eigenschaft verhindert, daß Subjekte Daten in Objekte einer niedrigeren Sicherheitsstufe schreiben können („*no write-down*“).

Zustandsänderungen des Systems ergeben sich nun aus der Änderung der Sicherheitsstufe eines Subjekts oder Objekts sowie aus dem Löschen oder Erzeugen eines Objektes. Das Modell beinhaltet noch Erweiterungen, wie eine Zugriffskontrollmatrix, mit der sich die Zugriffsrechte von Subjekten weiter einschränken lassen, und eine Objekthierarchie, bei der sich beim Hinabsteigen niemals die Sicherheitsstufe verringert.

Das Biba-Modell

Das BLP-Modell behandelt als Sicherheitsaspekt nur die Vertraulichkeit. Das *Biba-Modell* [Lan81] hingegen erweitert das BLP-Modell um den Aspekt der Integrität, indem neben der

[2] Ausgesprochen: „star-property“.

Sicherheitsfunktion f noch eine Integritätsfunktion g definiert wird, die jedem Subjekt und Objekt eine Integritätsstufe zuordnet. Mit der sogenannten „strikten Integritätspolitik“ ist das Biba-Modell dual zu BLP: Ein Subjekt s darf nur dann ein Objekt o lesen, wenn das Objekt eine mindestens ebenso hohe Integritätsstufe hat, d.h. $g(s) \leq g(o)$; und ein Subjekt darf nur dann ein Objekt modifizieren, wenn das Objekt höchstens eine ebenso hohe Integritätsstufe hat, d.h. $g(s) \geq g(o)$.

Betriebssysteme wie Windows NT/2000 und Unix sind jedoch DAC-Systeme, weder das Bell-LaPadula- noch das Biba-Modell lassen sich direkt darauf anwenden.

2.1.2. Das Clark-Wilson-Modell

Beim *Clark-Wilson-Modell* [Lee99] handelt es sich um ein Zugriffskontrollmodell, bei dem es vor allem um die Bewahrung der Datenintegrität geht. Die Kernkonzepte sind wohlgeformte Transaktionen und eine Trennung der Aufgaben (*separation of duty*).

Daten dürfen in diesem Modell nicht beliebig geändert werden, sondern nur auf eingeschränkte Weise, so daß die Integrität gewahrt bleibt: wohlgeformte Transaktionen führen die Modifikationen an Daten durch, und ein Journal protokolliert alle Datenänderungen. Die Aufgabentrennung verlangt, daß Operationen in einzelne Bestandteile zerlegt werden, die jeweils von verschiedenen Benutzern ausgeführt werden. Beispielsweise dürfen Personen, die wohlgeformte Transaktionen kreieren oder zertifizieren, diese Transaktionen nicht selbst ausführen. Kollusionen zwischen Benutzern werden vom Modell ausgeschlossen.

Um in einem Computersystem wohlgeformte Transaktionen zu unterstützen, dürfen bestimmte Daten nur durch bestimmte Programme verändert werden. Eine Textverarbeitung darf z.B. nur Textdateien ändern und keine ausführbaren Programmdateien. Einzelne Benutzer dürfen zudem nur eine bestimmte Menge von Programmen ausführen, so daß eine Aufgabentrennung gewährleistet werden kann. So dürfte z.B. eine Sekretärin die Textverarbeitung benutzen, nicht jedoch einen Compiler, um Programmdateien mit ausführbarem Code zu erzeugen.

Damit ergibt sich eine programmbasierte Zugriffskontrolle; die möglichen Zugriffe sind nicht mehr eine Teilmenge von $S \times O \times A$, sondern von $S \times P \times O \times A$, wobei P die Menge der Programme ist. Subjekte bekommen nicht das Recht, auf bestimmte Objekte zugreifen zu dürfen, sondern das Recht, bestimmte Programme auf bestimmte Objekte ausführen zu dürfen.

Das Clark-Wilson-Modell verwendet zur Einhaltung dieser Zuordnungen ebenfalls eine verbindliche Zugriffskontrolle (MAC). Denn wenn ein Benutzer die Liste der Programme, die bestimmte Daten modifizieren dürfen, verändern kann oder die Liste der Benutzer, die bestimmte Programme ausführen dürfen, könnte eine Verletzung der Datenintegrität nicht mehr kontrolliert werden. Der Besitzer einer Anwendung und die Kontrollmechanismen des Systems sind dafür

verantwortlich, daß alle Programme wohlgeformte Transaktionen darstellen. Das System muß dafür sorgen, daß nur die Programme die Daten ändern dürfen, wie es in der Sicherheitspolitik definiert wurde. Benutzer müssen vom System authentifiziert werden und sollten zur strikten Einhaltung der Aufgabentrennung nicht mehrere verschiedene Rollen annehmen dürfen.

Ein Nachteil dieses Modells ist, daß die Definition der Aufgabentrennung in einem komplexen Computersystem sehr aufwendig werden kann. Zudem finden sich keine Implementierungsdetails zur Überprüfung der wohlgeformten Transaktionen; ein Trojaner-Programm könnte immer noch gewisse Daten ändern. Der Vorteil dieses Modells ist die programmbasierte Zugriffskontrolle, durch die die Manipulationsmöglichkeiten eines Trojaners eingeschränkt werden. In Kombination mit dem Journal sind Manipulationen erkennbar und nachvollziehbar. Allerdings wird die Praktikabilität des Journals durch begrenzten Speicherplatz eingeschränkt.

Foley [Fol97] gibt ein Rahmenwerk an zur Spezifikation von Sicherheitsregeln bezüglich Vertraulichkeit und Integrität. Er erweitert dabei das Modell von Clark-Wilson um eine dynamische Trennung von Aufgaben. Die dynamische Trennung geschieht im wesentlichen dadurch, daß die Sicherheitsstufen eines Benutzers temporär an bestimmte Transaktionen und Daten angepaßt werden, wenn er die Transaktionen ausführen will – sofern der Benutzer das von der statischen Sicht her überhaupt darf. Ansonsten wird der Benutzer standardmäßig auf ein „Nullniveau“ gesetzt, was seine Sicherheitsstufe anbelangt.

2.1.3. Sicherheitseigenschaften von Systemen

In [ZL97] stellen Zakinthinos und Lee einen allgemeinen Formalismus vor, um theoretische Sicherheitseigenschaften von Systemen als Prädikate darzustellen. Sie betrachten dabei nur den Aspekt der Vertraulichkeit, genauer die Untersuchung der Möglichkeit, ob ein niedrig privilegierter Benutzer aus einer Folge von für ihn beobachtbaren Ereignissen die für ihn nicht beobachtbaren Ereignisse in einer Ereigniskette herleiten kann. Daraus könnte der Benutzer auf die Aktivitäten eines höher privilegierten Benutzers schließen. Sicherheitseigenschaften sind die Eigenschaften eines Systems, die es haben muß, um diese Möglichkeit zu verhindern.

Zakinthinos und Lee zeigen die schwächste Sicherheitseigenschaft, die erfüllt sein muß, damit kein Informationsfluß von höher privilegierten Benutzern zu niedriger privilegierten Benutzern stattfinden kann. Sie nennen diese Eigenschaft PSP (*Perfect Security Property*). Zudem zeigen sie, daß die Komposition zweier Systeme, die jeweils die PSP erfüllen, wieder ein System ist, daß die PSP erfüllt.

Um aber überprüfen zu können, ob ein System solch ein PSP-Prädikat erfüllt, müssen alle Ereignisse und alle möglichen Ereignisketten bekannt sein. Die Praktikabilität bezogen auf ein Java-System unter Windows 2000 (oder auch einem anderen Betriebssystem) mit seinen vielen

Ereignissen ist dadurch leider eingeschränkt, weshalb die Ergebnisse von Zakinthinos und Lee hier keine weitere Verwendung finden.

2.2. Allgemeine Konzepte und Verfahren

2.2.1. Sicherer Entwicklungsprozeß

Die Beziehung zwischen dem Entwicklungsprozeß und der Sicherheit des entwickelten Systems untersucht Payne [Pay00]. Dabei spielt vor allem eine Rolle, welche Aspekte und Formen von sicherheitsbezogener Arbeit zu einer besseren Sicherheit des Systems führen. Die Untersuchung erfolgte anhand von drei Unix-Systemen. Hintergrund für diese Untersuchung war, daß die Anwendung formaler Methoden und Modelle in der Praxis oft nicht angewendet würden, da diese meist zu komplex oder zu sehr auf militärische Aspekte ausgerichtet seien. Die Schwierigkeit und Komplexität dieses Problems mache eine Betrachtung des gesamten Entwicklungsprozesses erforderlich.

Die Untersuchung unterteilte die Entwicklungsprozesse nach dem „Wasserfall“-Modell in vier Phasen und wendete jeweils unterschiedliche Kriterien an:

1. *Analysephase*: Kriterien sind die Ziele und Anforderungen des Projekts.
2. *Designphase*: Die sicherheitsbezogene Systemfunktionalität (Sicherheitsfeatures, sichere Standardkonfiguration) ist hier ausschlaggebend.
3. *Implementationsphase*: Kriterien sind das Vorhandensein von Programmierrichtlinien, die Ausbildung und das Wissen der Entwickler über sicheres Programmieren.
4. *Wartungsphase*: Test- und Überwachungsprozeduren sowie das Management von Sicherheitsupdates wird bewertet.

Die Messung erfolgte in vier Dimensionen: Vertraulichkeit, Integrität, Verfügbarkeit und Auditing[3]. Innerhalb jeder Dimension wurden Sicherheitsfeatures nach ihrer Effektivität und Relevanz gemessen sowie Sicherheitslücken nach Ausmaß, Ausnutzbarkeit und Reparierbarkeit. Bei der Untersuchung der Entwicklungsprozesse wurden die jeweilige verfügbare Dokumentation genutzt, Interviews mit den Entwicklern geführt und Tests der Systeme selbst durchgeführt.

Als Ergebnis wird angegeben, daß jede Entwicklungsphase eine bestimmte Rolle spielt in Bezug auf die Sicherheit des Systems. Eine durchgehende, konsequente Beachtung von Sicherheitsaspekten in jeder Phase liefere ein sichereres System als ohne Beachtung. Im einzelnen seien folgende Punkte in der jeweiligen Phase zu beachten:

[3] automatische Code-Überwachung, Aufzeichnung von Veränderungen und Zuordnungen

- **Analyse**: Es lasse sich zwar keine direkte Verbindung zu einem speziellen Sicherheitsaspekt finden, aber die Definition von Sicherheit als wichtige *Anforderung* beeinflusse entscheidend die anderen Phasen. Dazu sei eine Analyse der späteren Einsatzumgebung und möglicher Angriffe nötig.

- **Design**: Diese Phase entscheidet über Qualität und Anzahl der Sicherheitsfeatures. Eine frühe Integration von Sicherheitsfeatures, die den Sicherheitsanforderungen aus der Analyse genügen, sei empfehlenswert, da nachträgliches Hinzufügen die Effektivität einschränken könne.

- **Implementation**: Programmierfehler können zu Sicherheitslücken führen. Die Beziehung zur Sicherheit des Gesamtsystems sei allerdings nicht so stark, da die Ausbildung der Entwickler nicht zwangsläufig zu besseren und das Fehlen von Programmierrichtlinien nicht zu schlechteren Ergebnissen geführt hätten. Statt dessen sei ein besseres Verständnis des Verhaltens von Systemaufrufen und Programmroutinen wichtiger.

- **Wartung**: Neben dem Verständnis von Programmroutinen habe kontinuierliches und intensives Code Auditing zum Finden und Beheben von Fehlern einen sehr großen Einfluß auf die Sicherheit des Systems.

Neumann [Neu00] zeigt allgemeine Anforderungen für die Entwicklung robuster und sicherer Systeme auf. Dazu gehört strenge Disziplin während des gesamten Entwicklungsprozesses, gutes Software Engineering und wiederholtes Testen des gesamten Systems. Das Ziel sollte eine robuste, erweiterbare und interoperable Architektur sein, wobei eine zu starke Abhängigkeit von nichtvertrauenswürdigen Komponenten vermieden werden sollte.

Jovalekic und Rist [JR98] beschreiben den Einfluß von Objektorientierung auf die Entwicklung von Gefahrenmeldeanlagen. Die Sicherheitsanforderungen sind hier vor allem bezüglich Verfügbarkeit und Integrität ausgerichtet, da Systemfehler schwerwiegende Folgen haben können. Das Entwicklungsmodell, das beschrieben wird, basiert auf objektorientierter Methodik und vor allem auf OMT[4]. (Dieser Ansatz ist allerdings schon etwas älter und durch den Unified Process[5] und UML[6] mittlerweile verbessert und verallgemeinert worden.)

Es wird beobachtet, daß strikte Qualitätsanforderungen eine bessere Qualität der Dokumentation erfordern, als es bei Projekten mit weniger Sicherheitsrelevanz der Fall ist. Ein einfache, effektive Technik dazu sei das Aufstellen von *Klassenspezifikationen*, in denen Fähigkeiten und Funkionen der Klasse beschrieben werden, ohne zu sehr in Implementationsdetails zu geraten.

[4] Object Modelling Technique
[5] Objektorientierter inkrementeller Entwicklungsprozeß, siehe dazu [JBR99].
[6] Unified Modeling Language, siehe dazu auch [FS00].

Damit erhöhe sich die Wiederverwendbarkeit von Software. Alternativ sei die Verwendung von OMT-Diagrammen zu empfehlen. (Diese Beobachtung deckt sich mit der Erkenntnis von Payne, daß das Verständnis des Verhaltens von Routinen wichtig ist für die Sicherheit des Systems.)

Weiterhin sei für die Qualitätssicherung die *„Traceability“* der Komponenten wichtig, d.h. die Möglichkeit, konkrete Softwarekomponenten bestimmten Anforderungen aus der Analysephase zuordnen zu können[7].

2.2.2. Sichere Software-Architektur

Einen Ansatz zum sicheren Systemdesign liefern Moriconi et al. [MQR97]. Sie empfehlen, die Integration von Sicherheitsaspekten direkt in die Architektur einer Software mit aufzunehmen und nicht erst später daraufhin zu erweitern. Die Autoren geben drei allgemeine Schritte an:

1. Formalisierung der Systemarchitektur mittels allgemeiner, abstrakter Architektur.

2. Verfeinerung der Systemarchitektur in spezialisierte Architekturen, die unter unterschiedlichen Annahmen über die Sicherheit der Systemkomponenten für die Implementation geeignet sind.

3. Genaue Überprüfung, ob jede Implementation der (spezialisierten) Architektur die intendierte Sicherheitspolitik erfüllt.

Dabei stellen sie fest, daß zum einen die Integration von Sicherheitsaspekten in die Architektur die Praktikabilität des Systems beeinträchtigen kann. Zum anderen sollte man statt *einer* Architektur das System als *mehrfache* Architekturen modellieren, wobei jeweils andere Annahmen über die Sicherheit von Systemkomponenten und Protokollen gemacht werden können.

Sie demonstrieren dies an einem Beispiel unter dem Aspekt der Zugriffskontrolle nach dem Modell von Bell-LaPadula. Die Bestandteile ihres Softwaresystems

- Schnittstellen

- Architektur (Art und Weise der Verbindung von Komponenten)

- Protokolle (Aufrufsequenzen zur Nutzung der Schnittstellen)

werden mittels einer Architektur-Beschreibungssprache SADL (*Structural Architecture Definition Language*) formalisiert. Im wesentlichen werden Schnittstellen als *Typen* definiert und mit

[7] siehe dazu ebenfalls [JBR99]

Constraints versehen, d.h. mit Bedingungen bezüglich der Protokolle und der Architektur. Auf diese Weise werden die Schnittstellen miteinander „verdrahtet“.

Die Autoren geben für das Beispiel mehrere spezialisierte Architekturen an, je nach Annahme über das Vorhandensein von Komponenten, die mehrere Sicherheitsstufen (*Multi Level Security*, MLS) unterstützen. Um zu verhindern, daß Informationen von einer Komponente mit hoher Sicherheitsstufe zu einer Komponente mit niedriger Sicherheitsstufe gelangen, werden Filter-Komponenten eingeführt oder Schnittstellen um „Label“-Parameter erweitert, die angeben, welche Sicherheitsstufe die aufrufende Komponente hat.

Die formalisierten Architekturen werden dann in Logik überführt, so daß Architekturen äquivalent zu logischen Theorien sind. Damit lassen sich Sicherheitseigenschaften der Architektur beweisen. Wenn eine Architektur sicher ist, dann sind auch alle gültigen Instanzen der Architektur sicher. Gültig sind Instanzen, wenn sie korrekt implementiert sind. Das Korrektheitskriterium geben die Autoren wie folgt an:

Sei Θ die Theorie der abstrakten Architektur. Sei Θ' die Theorie der konkreten (implementierten) Architektur und $I : \Theta \rightarrow \Theta'$ eine Theorieinterpretation. Die Implementation ist *korrekt*, wenn für alle Sätze F gilt:

$$(F \in \Theta \Rightarrow I(F) \in \Theta') \ \wedge \ (F \notin \Theta \Rightarrow I(F) \notin \Theta')$$

Das heißt, daß die konkrete Architektur nicht mehr und nicht weniger implementiert, als die abstrakte Architektur vorschreibt. Es kann aber durchaus gelten, daß Θ' eine echte Obermenge von $I(\Theta)$ ist, aber diese zusätzlichen Sätze dürfen keine in Θ ausdrückbaren Konsequenzen haben! Die Architektur-Instanziierung ist damit semantisch äquivalent zur Theorie-Instanziierung.

Die Autoren arbeiten unter zwei Annahmen. Zum einen nehmen sie an, daß MLS-Komponenten sich auch MLS-konform verhalten. Das heißt, es gibt hier keinen Schutz vor nicht vertrauenswürdigem Code. Zum anderen nehmen sie an, daß zur Kommunikation zwischen den Komponenten sichere Kanäle vorhanden sind.

Ein weiterer Kritikpunkt betrifft die Praktikabilität des Korrektheitskriteriums: Um die Korrektheit einer Implementation zu beweisen, müssen alle Sätze der als Theorie dargestellten abstrakten Architektur überprüft werden, ob sie sich in der konkreten Architektur wiederfinden. Dies könnte für komplexe Architekturen sehr schwierig werden. Zudem werden keine genaueren Aussagen über die Theorieinterpretation I gemacht. Zu befürchten ist, daß mit einer ungeeigneten Interpretation die Korrektheit fälschlicherweise nachgewiesen werden kann.

2.2.3. Schutz von Programmen durch Integritätsprüfer

Einen Ansatz für ein *Intrusion Detection System* liefern Ghosh et al. [GWC98]. Sie verwenden ein Neuronales Netzwerk zur Erkennung von Anomalien im Verhalten von Programmen. Um Anomalien erkennen zu können, muß zuvor eine dynamische Analyse des betreffenden Prozesses unter Normalbedingungen gemacht werden. Das Neuronale Netzwerk lernt dabei durch Rückmeldungen, ob das untersuchte Verhalten normal war oder nicht.

Als Nachteil des Verfahrens geben die Autoren an, daß die Lernphase unter Umständen mehrere Stunden oder Tage dauern kann. Wenn neue Daten hinzukommen, muß der gesamte Trainingsprozeß über alle Trainingsdaten erneut durchgeführt werden. Sie empfehlen für das Training für ein Programm, einen breiten repräsentativen Satz von Eingaben und internen Zuständen zu wählen. Wie diese Wahl zu treffen ist und wie die internen Zuständen gemessen werden sollen, wird aber nicht näher spezifiziert.

Tripwire [KS94a, KS94b] ist ein Integritätsprüfprogramm für Windows und Unix, das anhand von kryptographischen Hashwerten der Dateien eine Integritätsverletzung erkennen kann. Dazu werden die aktuellen Hashwerte mit zuvor erstellten Referenzwerten verglichen, die in einer Datenbankdatei gespeichert sind. Die Datenbankdatei muß auf einem sicheren Medium gespeichert werden, z.B. auf einer schreibgeschützten Diskette. Die Liste der Dateien, die geprüft werden sollen, wird ebenfalls in der Datenbankdatei gespeichert. Es können sowohl Programm- als auch Datendateien geprüft werden. Allerdings muß die Prüfung vom Benutzer explizit gestartet werden, es existiert keine automatische Prüfung von Programmdateien, wenn diese vom Betriebssystem geladen werden. *Tripwire* kann nicht verhindern, daß es selbst durch einen Trojaner ausgetauscht wird.

Mohay und Zellers [MZ97] haben einen Kernel und eine Shell für Unix entwickelt, die nicht nur Modifikationen von Programmen erkennen können, sondern auch deren Ausführung verhindern. Jede Anwendung wird zusammen mit einer kryptographischen Signatur gespeichert. Der Kernel bzw. die Shell führt Programme nur dann aus, wenn deren Signatur korrekt verifiziert werden kann. Die Shell kann zusätzlich beschränkt werden, so daß bestimmte Befehle nicht aufgerufen werden können.

Iglio und Bordoni [IB99] haben ebenfalls einen Kernel (für Linux) entwickelt, der modifizierte Programmdateien erkennen kann und deren Ausführung verhindert. Sie machen eine Unterscheidung des Systems in einen zuverlässigen Zustand (*trusted state*) und einen unzuverlässigen Zustand (*untrusted state*). Benutzer müssen in den zuverlässigen Zustand wechseln, wenn sie sicherheitskritische Aufgaben durchführen wollen. Dazu wird das System mit einem „TrustedBox“ (TB) Kernel neu gebootet, der sich auf einem speziellen Bootmedium befindet, das nicht manipuliert werden kann (z.B. eine schreibgeschützte Diskette).

Der TB Kernel prüft beim Start alle Systemkomponenten anhand von kryptographischen Hashwerten, die mit zuvor generierten Referenzwerten verglichen werden. Die Referenzwerte sind in einer kryptographisch signierten Datenbankdatei gespeichert, der Schlüssel zum Verifizieren der Signatur befindet sich auf dem Bootmedium des TB Kernels. Außerdem kann der *Super User* im zuverlässigen Zustand nicht jedes Programm ausführen.

2.2.4. Schutz von Programmen durch Verschlüsselung

In [ST98] wird ein theoretischer Ansatz diskutiert, mobilen Code (z.B. Java Applets oder mobile Softwareagenten) durch Kryptographie derart zu schützen, daß er auf nichtvertrauenswürdigen Umgebungen ausgeführt werden kann. Der Code des Prozesses wird so verschlüsselt, daß die verschlüsselte Form noch ausführbar ist und ihren Zweck erfüllt. Der ausführende Rechner wäre dann nicht mehr in der Lage, diesen Code zielorientiert zu verändern, ohne daß die Korrektheit der Ausführung des Codes beeinträchtigt wird.

Die Ergebnisse werden hier nicht weiter beachtet, da es in dieser Arbeit nicht um den Schutz von Software vor nichtvertrauenswürdiger Hardware geht, sondern umgekehrt um den Schutz eines Systems vor nichtvertrauenswürdigen Programmen.

2.2.5. Klassifizierung von Sicherheitssystemen

Das Bundesamt für Sicherheit in der Informationstechnik (BSI) nennt drei Schutzklassen zur Unterscheidung von sicheren Systemen nach ihrer Schutzwirkung [Heu97]:

1. *Einfache Schutzwirkung*: Ziel ist es, erfolgreiche Angriffe mit einfachen Methoden zu verhindern, damit keine vertraulichen Daten in fremde Hände geraten.

2. *Mittlere Schutzwirkung*: Die Möglichkeit eines Angriffs mit qualifizierten Methoden soll vermindert werden. Ein geringes Restrisiko wird akzeptiert. Allerdings müssen Manipulationsangriffe erkennbar sein.

3. *Hohe Schutzwirkung*: Hier dürfen schließlich keine Restrisiken mehr bestehen. Das System muß „immun“ gegen Manipulationsangriffe sein.

Diese Schutzklassen sollen als Orientierungsrahmen dienen, da eine so genaue Einteilung von Systemen in der Praxis nicht möglich sei. Ferner werden die wichtigsten Kriterien genannt, nach denen ein System klassifiziert wird:

- **Chiffriersicherheit** betrifft die mathematisch-kryptologische Sicherheit und stellt die Frage, *welche* Algorithmen zur Anwendung kommen. Zur Klasse 1 gehören kommerzielle, hinreichend geprüfte Verfahren, zur Klasse 2 und 3 die nicht publizierten, proprietären Verfahren.

- **Schlüsselmanagement** bezieht die Herstellung, Verteilung, Beglaubigung, Aufbewahrung, den Einsatz und die Vernichtung von Schlüsseln in ihrer Gesamtheit ein. Präzisere Angaben werden leider nicht gemacht, sondern nur das Beispiel genannt, daß bei der Schüsselerzeugung für Klasse 2 und 3 „verläßliche physikalische Rauschgeneratoren“ verwendet werden müssen.

- **Fehlbedienungs- und Fehlfunktionssicherheit** beinhalten die Frage nach der Möglichkeit, ob der Benutzer das System so falsch bedienen kann oder ob das System eine Fehlfunktion so ausführen kann, daß es zu einem Sicherheitsloch führt.

- **Manipulationssicherheit** fordert, daß das System Manipulationsangriffe erkennen (Klasse 2) oder sogar verhindern (Klasse 3) kann.

- Die **Vertrauenswürdigkeit der Umgebung von Entwicklung, Fertigung und Wartung** bezieht sich auf den Hersteller des Systems. Für Klasse 2 oder 3 kommen laut BSI nur deutsche Firmen in Frage, die vom Geheimschutz betreut werden.

- **Unterbindung verdeckter Kanäle**: Hier wird nur als Beispiel die elektromagnetische Abstrahlung des Systems genannt, deren Messung zu Rückschlüssen oder Angriffen führen könnte. In der Klasse 1 sind keine Filterungsmaßnahmen zu erwarten, in Klasse 2 dagegen schon die natürliche Dämpfung durch Gebäude und Entfernung.

Die Angabe einer vollständigen Kriterienliste ist laut BSI ohne Betrachtung des Einzelfalls nicht möglich. Zudem können die einzelnen Elemente in verschiedenen Fällen unterschiedlich gewichtet werden.

Zu kritisieren ist die Vorgabe bei der Chiffriersicherheit für Klasse 2 und 3. Daß nur proprietäre, nicht publizierte Verfahren angewendet dürfen, bedeutet *security by obscurity*, d.h. die Sicherheit der Verfahren beruht im wesentlichen darauf, daß niemand das Verfahren kennt. Wenn allerdings einmal doch das Verfahren bekannt werden sollte, sind sämtliche damit entwickelten Systeme plötzlich angreifbar und damit unbrauchbar. Besser ist es, wenn die Sicherheit nicht auf den Verfahren, sondern auf den verwendeten kryptographischen Schlüsseln beruht. Wenn einmal ein Schlüssel bekannt werden sollte, kann dieser einfach durch einen neuen ersetzt werden. Zudem sind nicht gleich alle Systeme betroffen, die ein solches Verfahren verwenden, da sie in der Regel jeweils andere Schlüssel benutzen.

2.3. Die Programmiersprache Java

2.3.1. Das Java Sicherheitsmodell

Sterbenz [Ste96] untersucht das Sicherheitsmodell von Java allgemein[8]. Wichtige Eigenschaften sind die automatische Überprüfung von *Array*-Grenzen zur Laufzeit, die Überprüfung von Typumwandlungen (*type casts*) auf Gültigkeit und die automatische Speicherverwaltung (*Garbage Collection*). Das Sicherheitsmodell wird in vier Schichten unterteilt:

1. *Die Java Programmiersprache*. Sie bietet schon eine Basissicherheit über ihre Spezifikation. Der Speicherzugriffsschutz wird von der Sprache her gewährleistet, d.h. es gibt keine Zeigerarithmetik.

2. *Die Java Virtual Machine*. Sie erzwingt die Einhaltung der Einschränkungen, die die Spezifikation der Java Sprache definiert.

3. *Die Laufzeitbibliotheken*. Der Zugriff auf Systemressourcen ist nur über definierte Schnittstellen möglich. Der Aufruf von Methoden dieser Schnittstellen führt zu impliziten Sicherheitsüberprüfungen.

4. *Die Laufzeitumgebung*. Die Sicherheitspolitik wird definiert, deren Einhaltung mit Hilfe der vorherigen Schichten erzwungen wird.

Kritisiert wird jedoch, daß durch die Verwendung von *native* deklarierten Methoden[9] ein Sicherheitsloch entstehen kann. Denn diese Methoden haben immer vollen Zugriff auf alle Systemressourcen. Besondere Aufmerksamkeit muß nach Sterbenz der *Sichtbarkeit*[10] von Objektattributen zukommen. Wenn Attribute z.B. als *public* deklariert wurden, können die eigentlich zu verwendenden Schnittstellen und damit die Sicherheitsüberprüfungen umgangen werden, indem direkt auf die Attribute zugegriffen wird. Wenn *Security Exceptions* aufgefangen werden, die bei einer Sicherheitsüberprüfung geworfen werden, weil eine gewünschte Operation nicht erlaubt ist, bemerkt der Benutzer nicht, daß die Durchführung der Operation versucht wurde.

[Pos98] behandelt ebenfalls das Sicherheitsmodell von Java. Hier wird das Modell in nur zwei Teile untergliedert: das *Sprachdesign*, verantwortlich für die Typsicherheit, und die *Sicherheitsarchitektur* – bestehend aus Bytecode-Verifikation, Klassenlader, Security Manager und Java Virtual Machine. Es werden allerdings nur Java Applets und keine Applications betrachtet. [Kok97] betrachtet neben möglichen Gefahren durch Applets vor allem Implementierungsfehler in der Java Virtual Machine oder im JDK.

[8] Der Artikel bezieht sich zwar auf JDK 1.0.1, die meisten Ergebnisse sind aber auch für JDK 1.4 gültig.
[9] Diese Methoden werden als Maschinencode (z.B. aus C erstellt) in das Java-Programm eingebunden.
[10] Gemeint ist hier, wie der Zugriff deklariert wurde: *private*, *public*, *protected* oder keine Deklaration.

2.3.2. Java Security Architecture

Policy und Permissions

Das Sicherheitsverhalten der Java Laufzeitumgebung wird im wesentlichen bestimmt durch seine Sicherheitspolitik, wie in [Gon99] beschrieben wird. Die Sicherheitspolitik bestimmt, welche individuellen Zugriffsrechte auf Ressourcen einem laufenden Programm zugewiesen werden. Das *Policy*-Objekt repräsentiert diese Sicherheitspolitik in der *Java Virtual Machine.* Der Inhalt für das *Policy*-Objekt wird standardmäßig von einer Klartextdatei aus dem Dateisystem eingelesen. Wenn keine Sicherheitsregeln definiert werden, arbeitet die JVM standardmäßig mit dem „Sandbox"-Modell, d.h. lokaler Code hat vollen Zugriff auf die Systemressourcen und entfernter Code (über das Netzwerk geladen) erhält nur beschränkte Zugriffsrechte. Dies bezieht sich sowohl auf Java *Applets* als auch auf Java *Applications*.

Die **`Policy`**-Klasse ist abstrakt, d.h. eine Unterklasse davon muß in die JVM geladen werden. Damit sind eigene Implementationen möglich, z.B. könnte der Inhalt für das *Policy*-Objekt aus einer Datenbank oder von einem Server im Netzwerk geladen werden. Das *Policy*-Objekt ist im wesentlichen nur eine Zugriffskontrollmatrix, d.h. es wird eine Zuordnung von `CodeSource` zu `Permission`s gespeichert. Ein **`CodeSource`**-Objekt enthält die URL und ggf. noch Zertifikate vom Programmcode. Die wichtigsten Methoden dieses Objekts sind `equals()`, die angibt, ob der genannte Code die gleiche URL und die gleiche Menge von Zertifikaten hat, und `implies()`, die eine Implikationsbeziehung ausdrückt: `a.implies(b)` bedeutet, daß `a` allgemeiner ist als `b`, d.h. die Rechte, die `a` hat, hat auch `b`[11].

Eine **`Permission`**-Klasse repräsentiert das Zugriffsrecht auf eine Ressource. Alle Permission-Klassen in Java sind positiv, d.h. es kann Zugriff nur *gewährt*, nicht aber verboten werden. Es gibt eine Hierarchie von *Permission*-Klassen, die abstrakte Oberklasse von allen ist `java.security.Permission`. Unterklassen können somit zur Repräsentation von (anwendungs)spezifischen Ressourcen definiert werden. Einige sind schon in den Standardpaketen des JDK enthalten, z.B. `java.io.FilePermission`. Diese *Permission*-Objekte sind im Zustand unveränderbar (*„immutable")*. Eigene *Permission*-Unterklassen sollten diese Regel auch nicht verletzen!

Permission-Objekte haben ebenfalls eine `implies()`-Methode zur Überprüfung von Implikationen. Zwei *Permissions* sind äquivalent, wenn die eine die andere impliziert und umgekehrt. Bei der Definition von *Permissions* werden meistens ein *Ziel* und die möglichen *Aktionen* darauf angegeben. `java.security.BasicPermission` ist eine abstrakte Oberklasse für „benannte" *Permissions*, wo Ziel und Aktion zusammengefaßt werden, z.B. *'exitVM'*, *'setFactory'*, *'closeWindow'* oder *'readDisplayPixels'*.

[11] z.B. `a` = http://meine.domain.de/classes/ und `b` = http://meine.domain.de/classes/prog.jar

Mittels `java.security.UnresolvedPermission` kann ein Platzhalter für noch nicht geladene *Permission*-Objekte definiert werden. Dies kann für die Performanz und Effizienz eines Systems wichtig sein, wenn sehr viele *Permission*-Objekte vorhanden sind, da diese nicht sofort geladen und instanziiert werden müssen. Sie werden erst geladen, wenn eine Zugriffskontrolle tatsächlich erfolgt.

Die Vergabe von *Permissions* muß mit Vorsicht geschehen, insbesondere für folgende:

- `java.security.AllPermission`
- *Permissions* für das Setzen von *System Properties*.
- Definieren von Paketen
- Laden von nativen Klassenbibliotheken
- Erzeugen von Klassenladern

Die *Permission* `AllPermission` impliziert alle anderen. Ein somit ausgestattetes Objekt hat alle Rechte in der JVM. Ein Objekt, das eins der letzten drei Rechte besitzt, kann Code in die JVM laden, der mehr Rechte besitzt, als ihm eigentlich zugedacht wurde. So kann nativer Programmcode die Sicherheitsmechanismen der JVM vollständig umgehen.

Die Zuordnung von `CodeSource` zu `Permissions` durch das *Policy*-Objekt wird schon *vor* dem Laden der Klassen, die in `CodeSource` enthalten sind, definiert. Es gibt allerdings eine Ausnahme: Die Instanziierung des *Policy*-Objekts kann solange verzögert werden, bis die erste Sicherheitsüberprüfung erforderlich ist. Die Java Architektur erlaubt, daß die Zuordnung der *Permissions* für eine Klasse zur Laufzeit geändert oder gelöscht werden kann, *nachdem* die Klasse bereits geladen wurde. Diese Änderungen müssen aber durch eine entsprechende *Permission* erlaubt werden: *'setPolicy'* in `java.security.SecurityPermission`. Bereits geladene Klassen behalten ihre zuvor zugeordneten Zugriffsrechte. Ob neu geladene Klassen, die neuen Rechte bekommen, hängt von der Implementierung des verwendeten Klassenladers ab. Die Standardimplementation der JVM von Sun macht dies nicht.

Zu beachten ist, daß die Zuordnung im *Policy*-Objekt für Klassen gilt und nicht für Objekte, d.h. es handelt sich um eine statische Zuordnung. Sie ist zudem addititiv: Wenn z.B. `/java/classes/` das Recht *'read'* hat und `/java/classes/X.class` das Recht *'write'*, dann hat die Klasse `X` effektiv die Rechte *'read'* und *'write'*.

Statt *Permissions* direkt an einzelne Klassen zu vergeben, können *Permissions* auch intermediär an *Protection Domains* gegeben werden. Eine **`ProtectionDomain`** ist die Menge von Objekten, die von einem Prinzipal (ein Benutzer oder ein anderer Softwaredienst) gerade

erreichbar sind. Alle Klassen und Objekte gehören jeweils zu genau einer `ProtectionDomain` und „erben“ indirekt die Rechte der Domain. Die Zuordnung von Code (Klassen und Objekte) zu einer `ProtectionDomain` geschieht durch die Java Laufzeitumgebung, bevor die jeweilige Klasse benutzbar ist und kann nicht mehr zur Laufzeit geändert werden. Die *Protection Domains* werden erzeugt, wenn die entsprechenden Klassen geladen werden. Klassen derselben Domain werden von demselben Klassenlader geladen.

Es gibt eine spezielle `ProtectionDomain`, die *System Domain*. In ihr befinden sich alle Systemklassen der JVM, die vom Ursprungsklassenlader geladen werden, und sie hat automatisch die *Permission* `AllPermission` zugeordnet. Die Interaktion zwischen verschiedenen *Protection Domains* kann verhindert werden, indem für jede Domain ein anderer Klassenlader verwendet wird. Denn jeder Klassenlader definiert seinen eigenen Namensraum. Eine Interaktion ist dann nur noch über Systemcode oder explizit erlaubte Schnittstellen möglich.

Ein Klassenlader fragt das *Policy*-Objekt, zu welcher `ProtectionDomain` eine zu ladende Klasse gehört und ruft `defineClass()` mit der Domain als Parameter auf, um die Klasse zu „definieren“. Wenn keine expliziten Angaben im *Policy*-Objekt gemacht wurden, wird eine standardmäßige `ProtectionDomain` verwendet.

Der Klassenlader

Ein wesentlicher Bestandteil der Architektur von Java ist das *sichere* Laden von Klassen, was Gong in [Gon98] beschreibt. Mit „sicher“ ist hier vor allem *typsicher* gemeint, d.h. daß nicht eine falsche Klasse geladen wird.

Das Laden von Klassen geschieht mit den Klassenladern (*class loader*). Der Vorgang beim Start einer Java Anwendung ist folgender:

1. Die *Java Virtual Machine* (JVM) erhält die Klassendatei und akzeptiert sie, wenn die Überprüfung des Bytecodes ein positives Ergebnis liefert.

2. Die JVM bestimmt den Ursprung[12] des Codes der Klasse und überprüft die Signatur, sofern der Code signiert wurde.

3. Die JVM überprüft, welche Sicherheitsregeln definiert wurden, und erstellt daraus einen Satz von Rechten für die Klasse (d.h. das *Policy*-Objekt wird erzeugt).

4. Die JVM lädt die Klasse, definiert ein Klassenobjekt und assoziiert die Klasse mit dem erstellten Satz von Rechten.

[12] z.B. die URL bei einem Applet oder der Dateipfad bei einer Application

5. Die JVM instanziiert Objekte der Klasse und führt die Methoden aus. (Dabei findet eine Typsicherheitsüberprüfung zur Laufzeit statt.)

6. Bei einem Zugriffsversuch einer Methode auf eine Ressource, überprüft die Zugriffskontrolle der JVM die Rechte der betreffenden Klasse. Wenn für den Zugriff die Rechte vorhanden sind, wird die Ausführung der Methode fortgesetzt, ansonsten wird ein *Security Exception* geworfen.

7. Wenn die Klasse und die instanziierten Objekte nicht mehr benötigt werden, werden sie vom *Garbage Collector* aus dem Speicher gelöscht.

In den Schritten 2 bis 4 ist der Klassenlader involviert. Mit diesem Konzept können neue Softwarekomponenten zur Laufzeit installiert werden. Klassenlader können auch vom Benutzer geschrieben werden, um z.B. eigene Sicherheitsattribute zu definieren.

Jeder Klassenlader hat einen eigenen Namensraum. Der *Typ* einer Klasse wird bestimmt durch den Klassennamen und den zugehörigen (definierenden) Klassenlader. Wenn mittels unterschiedlicher Klassenlader Klassen mit dem gleichen Namen geladen werden, werden sie von der JVM als unterschiedliche Typen interpretiert: Seien L_1 und L_2 zwei verschiedene Klassenlader und C ein Klassenname. Dann ist $\langle L_1, C\rangle$ der Typ der Klasse C, wenn sie mit Klassenlader L_1 geladen wurde und $\langle L_2, C\rangle$ der Typ der Klasse C, wenn sie mit Klassenlader L_2 geladen wurde. Es gilt: $\langle L_1, C\rangle \neq \langle L_2, C\rangle$.

Klassenlader werden als normale Java-Klassen implementiert. Folglich muß ein Klassenlader von einem anderen Klassenlader geladen werden. Dadurch ergibt sich eine **Klassenlader-Hierarchie** in der JVM, an deren Spitze der *Ursprungsklassenlader* steht, der nicht in Java, sondern nativ implementiert ist, und die Systemklassen der JVM lädt. Auf den Ursprungsklassenlader kann von Java-Code aus nicht zugegriffen werden.

Neben der Klassenlader-Hierarchie gibt es zwischen den Klassenladern in der JVM noch eine **Delegationsbeziehung**, die von der Hierarchie unabhängig ist: Ein Klassenlader kann eine Klasse selbst laden oder diese Aufgabe an einen anderen Klassenlader delegieren. Die Delegationsbeziehung wird als Vater-Sohn-Beziehung gebildet, wenn ein Klassenladerobjekt erzeugt wird. An der Spitze der Delegationsbeziehung steht auch hier wieder der Ursprungsklassenlader.

Dadurch ist zum einen eine Trennung gleicher Klassen in unterschiedliche Namensräume möglich. Zum anderen ergibt sich aus diesem Konzept das sichere Laden von Klassen, nämlich die Gewährleistung der Typsicherheit. Referenzen auf andere Klassen werden im Bytecode einer Klasse mit symbolischen Namen gespeichert. Als Beispiel sei die Klasse C wie folgt definiert:

```
public class C {
  public void m() {
    ...
    D item = new D();
    ...
  }
}
```

Wenn der Code ausgeführt werden soll, werden die symbolischen Referenzen durch einen Algorithmus in Zeiger auf tatsächliche Klassenobjekte aufgelöst:

- Zuerst wird in einem lokalen Cache des definierenden Klassenladers der Klasse C überprüft, ob die Klasse D schon geladen wurde.
- Wenn dies nicht der Fall ist, dann delegiert der definierende Klassenlader von C die Aufgabe an einen anderen Klassenlader – sofern eine Delegation definiert wurde. Ansonsten wird geprüft, ob es sich um eine Systemklasse handelt – in dem Fall wird der Ursprungsklassenlader der JVM verwendet.
- Wenn dies alles nicht zutrifft, versucht der Klassenlader selbst, die Klasse zu finden und zu laden. Durch eine eigene Implementation der Methode `findClass()` besteht die Möglichkeit, diesen Ladevorgang anzupassen. So kann z.B. erreicht werden, daß die Klassendatei der Klasse D über eine URL aus dem Internet geladen werden soll.

Der Typ der Klasse D ist dann $\langle L, D \rangle$, wobei L der Klassenlader ist, der D letztendlich definiert hat (dies muß nicht der Klassenlader von C sein!).

Wichtig für die Typsicherheit ist, daß ein und derselbe Klassenlader nicht die gleiche Klasse mehr als einmal laden darf. Es ist jedoch möglich, daß dieselbe Klasse von unterschiedlichen Klassenladern geladen wird, wobei jede geladene Klasse dann ein eigener Typ ist.

Weiterhin ist gemäß [Gon99] darauf zu achten, welche Klassen *Permissions* bezüglich Klassenladern erhalten sollen. Mit der Verwendung folgender Methoden kann sonst die Delegationsstruktur der Klassenladern unter Umständen umgangen werden:

- `ClassLoader.getSystemClassLoader()`
- `ClassLoader.getParent()`

Außerdem besteht die Möglichkeit, den Klassenlader einer Klasse zu finden und zu benutzen, um Objekte in einen fremden Namensraum zu plazieren. Dazu existieren folgende Methoden:

- `Class.getClassLoader()`
- `ClassLoader.findClass()`

Werden die Zugriffsrechte auf diese Methoden entsprechend eingeschränkt, sind die angesprochenen Probleme verhinderbar: Nur wer die `RuntimePermission("getClassLoader")` besitzt, kann eine Referenz auf einen Klassenlader bekommen. Klassenladerobjekte jedoch können immer den Klassenlader von anderen Objekten bekommen, wenn sie ein Delegationsvorfahre des betreffenden Klassenladers sind.

Der Standard-Klassenlader ist `java.net.URLClassLoader`. Alle Klassen im `CLASSPATH`[13] werden automatisch von Instanzen von `URLClassLoader` geladen. Der `URLClassLoader` gibt den zu ladenden Klassen zusätzlich zu den im *Policy*-Objekt definierten *Permissions* das Recht, auf die URLs, die dem Klassenlader übergeben wurden, lesend zuzugreifen.

Security Manager und Access Controller

Wenn ein Objekt eine Methode ausführt, um auf eine Ressource zuzugreifen, für die ein Zugriffsrecht nötig ist, dann wird der **`SecurityManager`** befragt, ob die Klasse des Objekts das Recht besitzt. Will man eine eigene Ressource mit einem Zugriffsrecht versehen, so muß man an geeigneter Stelle die Methode `checkPermission()` des `SecurityManager`s aufrufen. Folgendes Beispiel illustriert die Vorgehensweise:

```
public void doSomething() {
  SecurityManager sm = System.getSecurityManager();
  if (sm != null)
    sm.checkPermission(new MyPermission("doSomething"));
  ...
}
```

Die Methode `doSomething()` soll nur ausführbar sein, wenn man das Recht dafür hat. Dieses Recht wird durch die eigene *Permission*-Klasse `MyPermission` definiert. Der `SecurityManager` muß jedoch nicht notwendigerweise vorhanden sein, und er kann im laufenden Betrieb durch eine andere Implementierung ausgetauscht werden. Dafür sind die benannten *Permissions 'createSecurityManager'* und *'setSecurityManager'* in der Klasse `RuntimePermission` erforderlich.

[13] Systemklassen befinden sich seit JDK 1.2 in `Xbootclasspath`. Hier sollten sich auch nur die Klassen aus dem JDK befinden, denn Systemklassen haben immer `AllPermission`!

Der `SecurityManager` ist eigentlich nur aus Abwärtskompatibilität enthalten. Er delegiert alle Anfragen an den **`AccessController`**, der den Zugriffskontrollalgorithmus tatsächlich implementiert und die Sicherheitsüberprüfungen durchführt. Diese Klasse ist als `final` deklariert, kann also weder abgeleitet noch verändert werden, und es gibt keinen öffentlichen Konstruktor, sondern nur Klassenmethoden. Das hat zur Folge, daß der `AccessController` immer vorhanden ist und nicht zur Laufzeit ausgetauscht werden kann. Die Methode `checkPermission()` sollte demnach im `AccessController` direkt aufgerufen werden.

Die Methode `AccessController.checkPermission()` prüft, ob der gewünschte Zugriff, der durch die angegebene *Permission* bestimmt wird, im aktuellen Ausführungskontext erlaubt ist. Wenn dies nicht der Fall ist, wird eine `AccessControlException` geworfen. Der aktuelle Ausführungskontext wird durch die aktuelle Sequenz von Methodenaufrufen repräsentiert. Der Zugriffskontrollalgorithmus erlaubt nur dann den Zugriff, wenn jede *Protection Domain* im aktuellen Ausführungskontext die angegebene *Permission* besitzt.

Weil jede Methode zu einer Klasse und jede Klasse zu einer `ProtectionDomain` gehört, wird die gesamte aktuelle Aufrufkette aller Klassen der Anwendung überprüft. Dies ist wichtig, da sonst durch eine geschickte Aufrufkette versucht werden könnte, Zugriff auf eine Ressource zu erlangen, der für den aktuell ausgeführten Code nicht erlaubt ist. Im Fall von Methodenvererbung wird immer die *Protection Domain* derjenigen Klasse überprüft, die die entsprechende Methode tatsächlich implementiert.

Da der Zugriffskontrollalgorithmus in dieser Form in manchen Fällen zu restriktiv sein kann, gibt es eine Erweiterung: Code, der in einem `doPrivileged()`-Block ausgeführt wird, wird unabhängig von seinem Aufrufer ausgewertet. Der entsprechende Code ist selbst dafür verantwortlich, daß mit der zugewiesenen *Permission* sorgsam umgegangen wird. Folgendes Beispiel soll dies illustrieren: `A.x()` rufe `B.y()` auf. `B.y()` führe eine Aktion durch, die eine `YPermission` benötige. Normalerweise müßten sowohl die Klasse `A` als auch `B` die `YPermission` besitzen, damit die Aufrufkette erfolgreich durchgeführt werden kann. Wenn `B.y()` jedoch die entsprechende Aktion innerhalb von `doPrivileged()` durchführen würde, bräuchte nur `B` die `YPermission` zu besitzen. Es gilt zu beachten, daß `doPrivileged()`-Blöcke zwar das Überprüfen der Aufrufer verhindern, nicht jedoch der Klassen, die innerhalb des `doPrivileged()`-Blocks aufgerufen werden.

Die Anwendung von `doPrivileged()`-Blöcken ist *Thread*-bezogen und betrifft keine anderen Threads, die in der gleichen *Protection Domain* laufen, d.h. deren Zugriffskontrolle wird dadurch nicht beeinflußt. Ein Thread, der innerhalb eines `doPrivileged()`-Blocks neu gestartet wird, behält den privilegierten Status, auch wenn sein erzeugender Thread den `doPrivileged()`-Block wieder verlassen hat. Die Anwendung von `doPrivileged()` ist dementsprechend mit Vorsicht zu gebrauchen!

Untersuchungen der Architektur

Wallach und Felten [WF98] haben den „Stack Inspection“ Algorithmus, der den aktuellen Ausführungskontext überprüft, untersucht und in einer speziellen Logik modelliert. Die Entscheidung über den Zugriff korrespondiert mit der Konstruktion eines Beweises in der Logik. Die Autoren präsentieren eine Entscheidungsroutine, die effizient solch einen Beweis generieren kann und äquivalent mit dem „Stack Inspection“ Algorithmus ist. Sie zeigen, daß Zugriffe immer korrekt gewährt werden. Allerdings kann es sein, daß Zugriffe verboten werden, obwohl sie eigentlich erlaubt wären (*false positives*).

Qian, Goldberg und Coglio [QGC00] geben eine formale Spezifikation des Ladens von Klassen in der JVM und beweisen damit die Typsicherheit der JVM. Die Autoren geben formale Argumente, daß die Spezifikation der JVM und die Implementation von Sun[14] die Typsicherheit gewähren. In ihrem Beweis vereinfachen sie die Spezifikation der JVM. Dies soll aber das Ergebnis nicht verfälschen, da sich ihr Modell mit den fehlenden Aspekten durchaus erweitern lassen soll.

Goldberg [Gol98] hat auch den *bytecode verifier* der JVM näher spezifiziert, der vor dem Laden einer Klasse den Bytecode der Klassendefinition überprüft. Er gibt eine Datenflußanalyse an, die die Typkorrektheit der JVM auch auf dieser Ebene verifiziert.

Illiadis et al. [IGO98] betrachten die Architektur von JDK 1.2 insgesamt und vergleichen sie mit der von JDK 1.1. Besondere Beachtung wird dabei dem Aspekt, Java-Programme aus nicht vertrauenswürdigen Quellen zu laden, geschenkt. Wenn keine explizite *Policy* definiert wurde, wird das alte Sandbox-Modell benutzt, d.h. lokaler Code hat volle Zugriffsrechte auf alle Systemressourcen und jeder andere Code nur sehr eingeschränkte Zugriffsrechte. Mit der neuen Architektur sind aber feinere Regelungen mittels Zuordnung von *Permissions* möglich. Die Autoren haben die Konsequenzen für Threads untersucht: Die *Permission* eines Threads ist der Durchschnitt der *Permissions* aller seiner Aufrufer. Wenn ein Thread eine *Protection Domain* mit mehr Rechten betritt, dann behält er seine vorherigen Rechte. Wenn ein Thread eine *Protection Domain* mit weniger Rechten betritt, dann erhält er nur diese neuen Rechte.

Es wird auch als wichtig erachtet, daß ein Benutzer nicht zu viele sicherheitsrelevante Fragen bei der Konfiguration oder Installation eines Programms beantworten muß. Sonst würde er aufgrund von Langeweile oder Überforderung zu schnell überall auf „OK“ klicken. Besser wäre es, wenn man den Benutzer nur wenige Einstellungen machen lassen könnte. Dies wäre durch eine globale Sicherheitsrichtlinie erreichbar, die von einem Administrator definiert wird. Jeder Benutzer definiert lokal nur seine speziellen Einstellungen. In Java muß der Benutzer aber seine *Policy*-Datei vollständig selbst definieren.

[14] bezogen auf JDK 1.2.2

2.3.3. Die Anwendung und Entwicklung von Java

Die Sicherheit von Java basiert auf Datenkapselung, Trennung von Objekt-Namensräumen und Typsicherheit. Für die Entwicklung einer sicheren Applikation sind aber auch weitere Aspekte wichtig. So widmen sich Gong und Schemers [GS98] dem Schutz des internen Zustands von Objekten bezüglich Integrität und Vertraulichkeit. Es werden drei Konstrukte vorgestellt, die in der Java API vorhanden sind:

- `java.security.SignedObject`
- `java.security.GuardedObject`
- `javax.crypto.SealedObject`

`SignedObject` enthält ein Objekt[15] und die kryptographische Signatur des Objekts. Als Signieralgorithmus kommt standardmäßig SHA-1/DSA zur Anwendung, es können aber auch andere Algorithmen aus der *Java Cryptographic Extension* (JCE) verwendet werden. Eine Veränderung des so signierten Objekts würde auch eine Veränderung der Signatur bedeuten, was bei einer Verifizierung entdeckt werden würde. Im Gegensatz zum Code-Signieren von z.B. Applets wird hier nicht nur der statische Code der Klasse des Objekts signiert, sondern auch sein Zustand. `SealedObject` enthält dagegen eine verschlüsselte Form des Objekts[16]. Zur Verschlüsselung können die Algorithmen aus dem JCE verwendet werden, z.B. DES.

`GuardedObject` schützt den Zugriff auf ein anderes Objekt. Es enthält eine Referenz auf das zu schützende Objekt und einen `Guard`. Das zu schützende Objekt kann nur über eine `getObject()`-Methode erreicht werden, die den `Guard` fragt, ob der Zugriff erlaubt ist. `Guard` ist ein Interface, so daß jede beliebige Klasse diese Aufgabe übernehmen kann. Der Schutzmechanismus kann dahingehend angepaßt werden, daß jede Methode des Objektes unterschiedlich bewacht wird. Jede *Permission*-Klasse ist ein `Guard`.

Gong und Dodda beschreiben in [GD99] den Entwicklungsprozeß der *Java 2 Standard Edition*[17], bei der das einfache *Sandbox*-Modell von der neuen Sicherheitsarchitektur der *Protection Domains* abgelöst wurde. Es wurden zahlreiche Bemühungen zur Qualitätssicherung durchgeführt, wie Mitarbeiterschulungen, Programmierrichtlinien und eine automatische Code-Überprüfung auf bekannte Programmierfehler. Gegen Sabotage aus den eigenen Reihen haben die Entwickler keine Maßnahmen ergriffen. Die Architektur von *Java 2* basiert auf der Annahme, daß das zugrundeliegende Betriebssystem sicher ist und sich korrekt verhält.

Als Beispiele für schlechte Programmierpraktiken wird folgendes genannt:

[15] genauer: eine Kopie davon
[16] auch hier wieder eine Kopie
[17] Die *Java 2 Standard Edition* wurde erstmals im *Java Development Kit* (JDK) 1.2 implementiert.

- `get`-Methoden sollten keine Referenz auf ein privates Objektfeld zurückgeben, da sonst eine direkte Manipulation der Daten in diesem Feld möglich ist.
- `public` deklarierte Objektfelder sollten immer als `final` deklariert sein, da sonst auch hier direkte Änderungen möglich sind, die zudem zu Synchronisationsproblemen führen können.

Wenn man ein Tool hat, welches nach solchen Mustern regelmäßig im Quellcode sucht, kann man schlechte Implementationen frühzeitig erkennen und korrigieren.

Außerdem wurde die Qualitätssicherung von Java durch intensives Testen unterstützt. Es wurden automatische Testumgebungen benutzt, die so oft wie möglich (täglich) durchgeführt wurden, um Fehler frühzeitig zu erkennen. Es werden folgende Arten von Tests von den Autoren genannt und empfohlen:

- **Kompatibilitätstests**: Prüfung, ob die Design-Spezifikationen und Schnittstellen eingehalten werden.
- **Produkttests**: Überprüfung der korrekten Implementation von Features durch Funktionalitätstests, Konfigurationstests, Interoperabilitätsttest, Streßtests und Performanztests.
- **Penetrationstests**: Durchführung von direkten Angriffen auf das Programm („Tiger-Team“-Ansatz).
- **Regressionstests**: Überprüfung, ob nach einem Bugfix der behobene Fehler auch tatsächlich nicht mehr auftritt.
- **Fehlerbehandlung**: Wenn Fehler auftreten, dann werden sie einem *Trackingsystem* gemeldet, das den Status der Fehlerkorrektur überwacht.

Diese Sicherheitsanstrengungen lassen sich durchaus verallgemeinern und auf die Entwicklung eigener Software übertragen, um eine saubere Implementation zu erhalten.

2.3.4. Erweiterungen von Java

JSEF In [HKK00] wird JSEF, eine Erweiterung der *Java Security Architecture*, vorgestellt. Dabei geht es um die Erstellung von Sicherheitsprofilen für den Zugriff auf Ressourcen – wie Dateien und Netzwerk-Sockets, aber auch der Zugriff auf bestimmte Java APIs. Java an sich bietet nur die Möglichkeit, Zugriffsrechte zu *gewähren*. Hier wird ein Framework für Java vorgestellt, der auch die Möglichkeit bietet, Zugriffsrechte zu *verweigern*. Dadurch lassen sich

Sicherheitsregeln einfacher definieren. Es wird dabei zwischen globalen und benutzerspezifischen Sicherheitsregeln unterschieden, die dann bei einem Zugriffsversuch nach bestimmten Kriterien „gemischt“ werden. Mit diesem Framework lassen sich so hierarchische Sicherheitspolitiken zusammen mit einer Gruppenverwaltung implementieren.

Kava Welch und Stroud stellen in [WS99] Kava vor. Es handelt sich dabei um einen Aufsatz für die Java Laufzeitumgebung, der durch Bytecode-Transformationen beim Laden von Klassen ein nachträgliches Einfügen von Sicherheitsprüfungen ermöglicht. Damit stehen Methodenaufrufe, Initialisierungen, und Zustandsänderungen von Objekten unter der Kontrolle von Kava. Was kontrolliert werden soll, wird durch sogenannte Bindungen definiert. Mit Kava können Anwendungen entwickelt werden, die das Clark-Wilson-Modell unterstützen. Allerdings muß Kava zum vertrauenswürdigen Teil der JVM hinzugefügt werden. Damit ist bei Änderungen der JVM oder des Bytecode-Formats auch ein Update von Kava erforderlich. Zudem könnten die Bindungen manipuliert werden, da sie von Kava nicht geschützt werden. Kava bietet aber die Möglichkeit, das Sicherheitsmodell unabhängig von der Anwendungsentwicklung zu erstellen und noch nachträglich zu ändern. Eine nachträgliche Änderung des Sicherheitsmodells kann jedoch für bestimmte sicherheitskritische Anwendungen unerwünscht sein.

JAAS Seit JDK 1.3 befindet sich das Erweiterungspaket JAAS in Java, nämlich der *Java Authentication und Authorization Service* [LGK99]. JAAS erweitert das code-bezogene Sicherheitskonzept von Java um benutzerbezogene Zugriffskontrollen. Dazu werden zwei Begriffe definiert: Ein *Subjekt* ist ein Benutzer oder ein anderer Softwaredienst; ein *Prinzipal* ist der Name eines Subjekts. Ein Subjekt kann durchaus mehrere Namen haben. Durch eine erfolgreiche Authentifizierung wird ein Prinzipal mit einem Subjekt in der Java Laufzeitumgebung assoziiert. Die Informationen, die ein Subjekt zur Benutzung neuer Dienste authentifizieren, sind Zeugnisse (*credentials*), die aus einer öffentlichen und einer privaten Menge von Informationen bestehen. Mit JAAS kann eine Sicherheitspolitik definiert werden, die genau angibt, welche Ressourcen nur durch autorisierte Prinzipale benutzbar sind.

Dynamischer Klassenaustausch Malabarba et al. [MPG00] haben Java dahingehend untersucht, ob sich Klassen zur Laufzeit austauschen lassen. Sie sehen dafür nur zwei Möglichkeiten: die Neudefinition der Sprache Java oder die Modifikation der *Java Virtual Machine*. Malabarba et al. haben die JVM geändert und einen speziellen Klassenlader geschrieben, um Klassen dynamisch austauschen zu können. Auf diesem Artikel basierend kann also angenommen werden, daß es ohne größeren Aufwand (vor allem ohne Manipulation der JVM) nicht möglich ist, Java-Klassen zur Laufzeit auszutauschen.

2.4. Das Betriebssystem Windows 2000

Die Sicherheit einer Anwendung wird nicht nur von der Art und Weise, wie sie programmiert wurde, bestimmt, sondern auch von der Ausführungsumgebung, in der sie eingesetzt wird. Die Architektur und die grundlegenden Mechanismen des Betriebssystems definieren somit die Basis, auf der die Sicherheit des Gesamtsystems fußt. Deshalb wird im folgenden das Betriebssystem Windows 2000 von Microsoft näher betrachtet [SR00, MS00].

2.4.1. Systemarchitektur

Windows 2000 unterscheidet zwischen **Benutzermodus** und **Kernelmodus**. Anwendungen, die im Kernelmodus laufen, haben einen uneingeschränkten Zugriff auf die Hardware und auf Objekte im Systemspeicher. Benutzermodus-Anwendungen haben keinen direkten Zugriff auf die Hardware und den Systemspeicher. Sie besitzen einen eigenen privaten Adreßraum, auf den andere Benutzermodus-Anwendungen keinen Zugriff haben, und rufen Betriebssystemfunktionen nur indirekt über sogenannte **Teilsystem-DLLs** auf (Kernel32.dll, Advapi32.dll, User32.dll, Gdi32.dll). DLLs (Dynamic Link Libraries) sind Funktionsbibliotheken, die dynamisch geladen und von Programmen gemeinsam genutzt werden.

Windows 2000 besitzt drei **Umgebungsteilsysteme**, die für die Ausführung von Anwendungen verantwortlich sind: das *OS/2-Teilsystem* (Os2ss.exe), das *POSIX-Teilsystem* (Psxss.exe) und schließlich das *Win32-Teilsystem* (Csrss.exe). Die Umgebungsteilsysteme laufen alle im Benutzermodus. Nur das Win32-Teilsystem hat eine zusätzliche Komponente, die im Kernelmodus läuft (Win32k.sys) und vor allem für die graphische Benutzeroberfläche verantwortlich ist. Zudem existieren im Benutzermodus noch zwei weitere Klassen von speziellen Anwendungen: System-Unterstützungsprozesse und Dienste.

Dienste sind eigentlich normale Win32-Programme, die jedoch besondere Funktionen der Win32-API aufrufen, um mit dem Dienststeuerungsmanager zusammenzuarbeiten. Dienste laufen im Hintergrund und warten darauf, daß andere Anwendungen sie aufrufen.

Zu den **System-Unterstützungsprozessen** gehört der *Dienststeuerungsmanager* (Services.exe). Er ist verantwortlich für das Starten, Anhalten und Konfigurieren von Diensten. Der *Sitzungsmanager* (Smss.exe) ist als erster Benutzermodus-Prozeß verantwortlich für das Starten von Teilsystem-Prozessen und dem *Anmeldeprozeß* (Winlogon.exe). Der Anmeldeprozeß reagiert als einziger Prozeß auf die Tastenkombination Strg+Alt+Entf, um so eine neue Benutzeranmeldung einzuleiten. Nach erfolgreicher Benutzerauthentifizierung durch den *Lokalen Sicherheits-Authentifizierungsserver* (Lsass.exe) startet Winlogon einen Initialisierungsprozeß, der am Ende eine neue Shell (Standard: Explorer.exe) für den Benutzer startet.

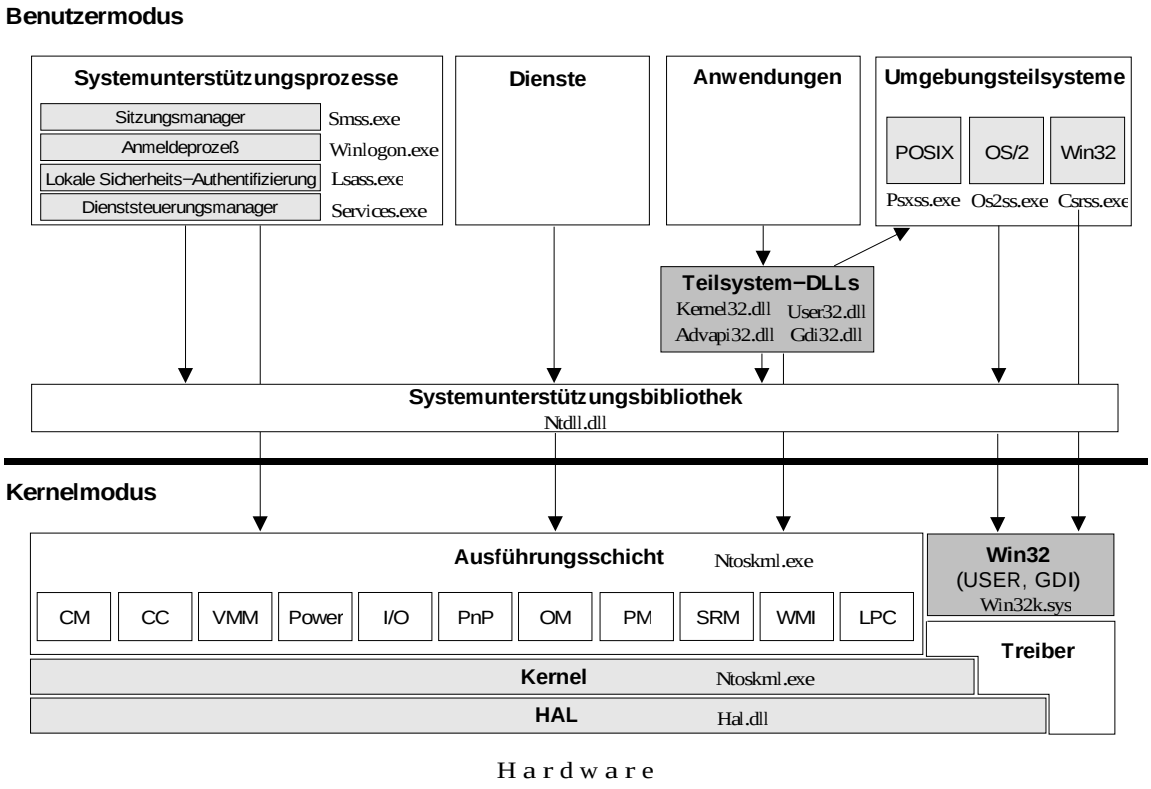

Abbildung 2.1.: Architektur von Windows 2000

Die **System-Unterstützungsbibliothek** (Ntdll.dll) dient als Schnittstelle zu den Systemdiensten im Kernelmodus. Sie läuft im Benutzermodus und wird vor allem von den Teilsystem-DLLs verwendet. Weiterhin bietet sie verschiedene Hilfsfunktionen, die auch von den Diensten, den Umgebungsteilsystemen und den System-Unterstützungsprozessen verwendet werden.

Im Kernelmodus von Windows 2000 läuft die **Ausführungsschicht** (Ntoskrnl.exe). Sie implementiert die verschiedenen Systemdienste wie den *Konfigurationsmanager* (*CM*) zur Verwaltung der *Registry*, den *Prozeß- und Threadmanager* (*PM*), den *Sicherheitsreferenzmonitor* (*SRM*), den *I/O-Manager*, den *Plug-and-Play-Manager* (*PnP*), die *Energieverwaltung* (*Power*), die *Windows Management Instrumentation* (*WMI*), den *Cachemanager* (*CC*), die *virtuelle Speicherverwaltung* (*VMM*), den *Objekt-Manager* (*OM*) und den *LPC-Mechanismus*. Unterhalb der Ausführungsschicht arbeitet der **Kernel** (auch Ntoskrnl.exe). Er ist für die Bereitstellung von hardwarenahen Mechanismen verantwortlich, z.B. Multiprozessor-Synchronisation, Interrupt- und Ausnahmenbehandlung, und für die Bereitstellung von Betriebssystemelementen, sogenannten Kernelobjekten. Damit werden die Ausführungsschicht und Gerätetreiber von der Hardwarearchitektur unabhängig gemacht. Auf unterster Ebene befindet sich die HAL, die **Hardware Abstraction Layer** (Hal.dll). Sie verbirgt spezifische und rechnerabhängige Unterschiede innerhalb einer Hardwarearchitektur.

Gerätetreiber (*.sys) arbeiten vorwiegend im Kernelmodus. Sie dienen zur Ansteuerung der Hardware, des Dateisystems oder eines Netzwerks. Sie greifen dabei nicht direkt auf die Hardware zu, sondern nutzen Funktionen der HAL oder des Kernels. Eine Ausnahme bilden die Grafiktreiber, sie greifen direkt auf die Hardware zu. Alle installierten Treiber sind in der *Registry* unter dem Schlüssel *HKEY_LOCAL_MACHINE\ SYSTEM\ CurrentControlSet\ Services* verzeichnet.

2.4.2. Systemmechanismen

Objekt-Manager

Der Objekt-Manager befindet sich in der Ausführungsschicht und ist verantwortlich für das Erstellen, Löschen und Überwachen von Objekten. Objekte gewähren Zugriff auf die internen Dienste der Ausführungsschicht, zum Beispiel ist ein Dateiobjekt für den Zugriff auf eine Datei nötig. Die Anforderung von Objekten über den Objekt-Manager stellt damit einen einheitlichen Mechanismus dar, um Systemressourcen aller Art verwenden zu können.

Es werden intern zwei Objektarten verwendet: *Objekte der Ausführungsschicht* repräsentieren die Systemressourcen, *Kernelobjekte* sind die Basiselemente, aus denen andere Objekte erstellt werden. Auf diese beiden Objektarten haben nur Prozesse im Kernelmodus direkten Zugriff. Die Umgebungsteilsysteme nutzen die Objekte der Ausführungsschicht, um eigene Objekte zu erstellen.

Der Objekt-Manager verwaltet im jeweiligen Objektheader eine Liste der Prozesse, die ein **Handle** auf das Objekt geöffnet haben. Ein Handle ist ein indirekter Zeiger auf das Objekt, d.h. kein direkter Zeiger auf die physikalische Adresse im Speicher, sondern eine Indexnummer eines Verzeichnisses, in dem der tatsächliche Zeiger steht. Dieses Verzeichnis wird im Kernelmodus verwaltet und kann auch nur von dort gelesen werden. Der Zugriff auf Objekte ist für Benutzermodus-Prozesse nur über Handles möglich. Auch der Aufruf von Operationen auf Objekte geschieht mit Handles. Nur Kernelmodus-Code kann über Zeiger direkt auf Objekte zugreifen.

Prozesse können Objekthandles auch von ihrem übergeordneten Prozeß erben, sofern er dies erlaubt, oder von anderen Prozessen als duplizierte Handles empfangen. Allerdings darf nur der Objekt-Manager Handles erstellen und Objekte suchen. Der Objekt-Manager ist somit in der Lage, jede ein Objekt betreffende Aktion im Benutzermodus zu überwachen. Dies ermöglicht die Kontrolle, ob der Aufrufer die Erlaubnis für eine gewünschte Operation auf ein Objekt hat.

Jeder Prozeß hat eine **Handletabelle**. Dort stehen alle offenen Handles des Prozesses. Wenn ein Prozeß ein Handle öffnen will, fragt der Objekt-Manager den Sicherheitsreferenzmonitor nach den Zugriffsrechten des Prozesses für das Objekt und speichert diese Information

in den Verzeichniseintrag, auf den das Handle zeigt. Wenn nun Operationen mit diesem Handle durchgeführt werden sollen, kann sofort geprüft werden, ob die erforderlichen Zugriffsrechte vorhanden sind.

Der **Namespace** von Objekten ist global in der Ausführungsschicht: Wenn ein Prozeß ein Objekt erstellt und einen Namen dafür angibt, können andere Prozesse über den Objekt-Manager mit diesem Namen ein Handle auf das Objekt öffnen. Wenn das nicht gewünscht ist, darf man bei der Objekterstellung keinen Namen angeben. Das Objekt bleibt dann privat für den Prozeß, der es erstellt hat.

LPC (Local Procedure Call)

LPCs stellen einen internen Kommunikationsmechanismus dar, um Nachrichten schnell zwischen verschiedenen Prozessen auszutauschen. Ein Prozeß muß dazu ein bestimmtes Objekt der Ausführungsschicht exportieren, das **Portobjekt**. Über dieses Portobjekt können dann Daten direkt ausgetauscht werden oder indirekt, indem Zeiger auf lesbare Speicherbereiche übergeben werden. *Winlogon*, *Lsass* und der Dienststeuerungsmanager verwenden beispielsweise LPCs.

Speicherverwaltung

Windows 2000 kann standardmäßig bis zu 4 GB physischen Speicher adressieren. Benutzerprozesse verfügen über einen gemeinsamen 4 GB großen virtuellen 32Bit-Adreßraum. Die Speicherverwaltung ist Teil der Ausführungsschicht. Sie umfaßt Systemdienste zum Zuweisen, Freigeben und Verwalten von virtuellem Speicher, eine Behandlungsroutine für ungültige Übertragungen oder Zugriffsfehler sowie weitere Schlüsselkomponenten.

Der **virtuelle Speicher** setzt sich zusammen aus dem physischen Arbeitsspeicher und einer Auslagerungsdatei auf dem Systemlaufwerk. In der *Registry* kann angegeben werden, ob Speicherseiten in der Auslagerungsdatei beim Systemende mit Nullen gefüllt werden sollen[18]. Dies stellt eine Sicherheitsfunktion dar, um zu verhindern, daß Daten ausgelesen werden können, die sich zuvor im Arbeitsspeicher befunden haben. Wenn Benutzermodus-Prozesse neue Speicherseiten anfordern, erhalten sie immer mit Nullen initialisierte Seiten.

Beim Aufruf von Speicherverwaltungsdiensten kann ein Handle übergeben werden, der den Prozeß bezeichnet, dessen virtueller Speicher manipuliert werden soll. Eine Manipulation erfolgt allerdings nur, wenn eine entsprechende Berechtigung gegeben ist. So können übergeordnete Prozesse standardmäßig immer den Arbeitsspeicher von ihnen erstellten Prozessen manipulieren, d.h. Lesen, Schreiben, Zuweisen oder Freigeben. Die Umgebungsteilsysteme machen

[18] Unter dem Schlüssel *HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management* muß dazu der Registrierungswert *ClearPageFileAtShutdown* auf 1 gesetzt werden.

z.B. von dieser Eigenschaft Gebrauch. Sie verwalten auf diese Weise ihre Clientprozesse. Das bedeutet, daß *Csrss* auf den Speicherbereich jedes Win32-Programms zugreifen kann.

Mit **Abschnittsobjekten** kann die Nutzung gemeinsamer Speicherbereiche für mehrere Prozesse implementiert werden. Dies erfolgt z.B. bei der gemeinsamen Nutzung einer DLL. Das Abschnittsobjekt definiert den Speicherbereich, auf den die Prozesse gemeinsam Zugriff haben.

Windows 2000 besitzt diverse **Speicherschutzmechanismen**, damit kein Benutzermodus-Prozeß Zugriff auf den Adreßraum eines anderen Prozesses erlangen kann: Erstens ist der Zugriff auf systemweite Datenstrukturen von Kernelmodus-Komponenten nur im Kernelmodus möglich, Benutzermodus-Prozesse können darauf nicht zugreifen. Zweitens hat jeder Prozeß seinen eigenen privaten Adreßraum, auf den andere Prozesse nicht zugreifen können (ausgenommen übergeordnete Prozesse und gemeinsame Speicherbereiche). Drittens werden auch hardwaregesteuerte Speicherschutzmechanismen der Prozessoren benutzt. Dies ist allerdings abhängig von der Architektur der Hardware. Viertens verfügen Abschnittsobjekte über Zugriffskontrollisten (ACLs). Versucht ein Prozeß, ein Abschnittsobjekt zu öffnen, wird überprüft, ob er das entsprechende Zugriffsrecht gemäß ACL besitzt. Ein Thread kann den Seitenschutz in einem Abschnittsobjekt ändern, wenn er die Berechtigung dazu in der ACL hat.

Registry

Die *Registry* ist die zentrale Datenbank von Windows 2000, in der alle wichtigen Informationen zu den Benutzern und zum System stehen. Sie ist in einer Baumstruktur aufgebaut, wobei die einzelnen Knoten benannte Schlüssel oder Werte sein können. Schlüssel können weitere Schlüssel enthalten und haben auch einen eigenen (unbenannten) Wert.

Die Sicherheitskontenverwaltung (*Security Account Management*) mit den Konto- und Gruppeninformationen inklusive der Kennwörter befindet sich unter dem Schlüssel *HKEY_LOCAL_MACHINE\SAM* (kurz: *HKLM\SAM*). Der Zugriff auf diese Daten ist stark beschränkt. Lediglich das Systemkonto, welches ein eigenständiges Benutzerkonto darstellt und das Computersystem repräsentiert, darf darauf zugreifen. In *HKEY_LOCAL_MACHINE\SECURITY* stehen die Sicherheitsrichtlinien und die Benutzerrechtszuordnungen. Dort wird z.B. definiert, wer Programme installieren oder das System herunterfahren darf.

Mit einem Trick kann man direkten Zugriff auf die Daten in *SAM* und *SECURITY* nehmen: Mit dem Befehl *at.exe* lassen sich zeitgesteuert Programme starten. Dies wird normalerweise für Backup-Zwecke verwendet. Jedoch läßt sich damit auch der *Registry*-Editor starten. Weil mit *at.exe* gestartete Programme im Systemkonto laufen, hat der auf diese Weise gestartete *Registry*-Editor Zugriff auf die Daten!

Die *Registry* wird aus sogenannten *Hives* zusammengesetzt. Ein *Hive* ist ein *Registry*-Frag-

ment, das in einer Datei auf dem Datenträger gespeichert ist. Die *Hives* von *SAM* und *SECURITY* befinden sich z.B. in den Verzeichnissen *%SystemRoot%\System32\Sam* bzw. *%SystemRoot%\System32\Security*[19].

Der Konfigurationsmanager definiert einen Typ für Schlüsselobjekte und übergibt dem Objekt-Manager ein Schlüsselobjekt `Registry`, in dem die *Registry* komplett enthalten ist. Wenn Anwendungen *Registry*-Schlüssel erstellen oder auf vorhandene Schlüssel zugreifen wollen, dann geschieht dies nur über die *Registry*-API-Funktionen mit Hilfe von Schlüsselobjekten. Die Schlüsselobjekte werden vom Objekt-Manager verwaltet. Auf diese Weise sind Zugriffskontrollen auch für *Registry*-Schlüssel möglich. Jeder Schlüssel verfügt über eine Sicherheitsbeschreibung, die den Zugriff auf ihn schützt.

Dienste

Dienste sind eigentlich normale Win32-Programme, die zusätzlich besondere Funktionen aufrufen, um mit dem Dienststeuerungsmanager zu kommunizieren. Sie sind Prozesse, die beim Systemstart gestartet werden und im Hintergrund laufen. Sie können optional auch erst bei Bedarf oder explizit durch Benutzer gestartet werden. Sie erfordern keinen interaktiven Benutzer, sondern warten im Hintergrund darauf, daß sie von anderen Anwendungen aufgerufen werden. Dienste implementieren häufig die Serverseite von Client/Server-Anwendungen. Zu den Komponenten eines Dienstes gehören die Dienstanwendung und ein Dienststeuerungsprogramm, das für das Starten, Anhalten und Konfigurieren des Dienstes verantwortlich ist.

Ein Dienst kann im Kontext eines bestimmten Benutzerkontos laufen. In der Regel ist dies das **lokale Systemkonto**. Dienste sind damit in der Lage, Zugriffstoken erstellen zu dürfen, haben uneingeschränkten Zugriff auf Dateien und *Registry*-Schlüssel und gehören zur Administratorengruppe. Wenn ein Dienst unter einem anderen Konto laufen soll, dann stehen die Anmeldedaten für den Dienst in der *Registry* im Unterschlüssel *Secrets* von *HKLM\SECURITY*.

Das Konto sowie weitere Einstellungen und Parameter des Dienstes stehen in der *Registry* unter *HKLM\SYSTEM\CurrentControlSet\Services*. Dort steht auch die Sicherheitsbeschreibung, die die Zugriffsrechte auf den Dienst regelt. Der Wert *ImagePath* unter diesem Schlüssel definiert den Pfad zur Programmdatei des Dienstes. Wenn er nicht gesetzt ist, sucht der Dienststeuerungsmanager im Verzeichnis *%SystemRoot%\System32* nach der Programmdatei. (Treiber werden unter demselben Schlüssel gelistet. Ihr Standardverzeichnis lautet allerdings *%SystemRoot%\System32\Drivers*.)

Der Dienststeuerungsmanager (SCM) startet Autostartdienste und auch Autostarttreiber. Er

[19] *%SystemRoot%* bezeichnet eine Systemvariable, die das Basisverzeichnis von Windows 2000 angibt, standardmäßig ist dies *C:\WINNT*.

beachtet dabei Reihenfolgeabhängigkeiten. Systemstarttreiber, d.h. Treiber, die grundlegend für das System sind, werden vom I/O-Manager geladen. Dienste meldet der SCM in anderen Konten als dem Systemkonto an, indem er die *Lsass*-Funktion `LsaLogonUser()` aufruft. *Lsass* vergleicht daraufhin das Kennwort, das unter dem Namen *_SC_<Dienstname>* im Unterschlüssel *HKLM\SECURITY\Policy\Secrets* gespeichert ist, mit den Kennwort des angegebenen Benutzerkontos.

Wenn ein Dienst fehlschlägt, z.B. aufgrund eines unerwarteten Ende des Dienstprozesses, dann kann der SCM bestimmte Aktionen ausführen, die in dem Wert *FailureActions* in der *Registry* stehen. So kann der SCM einen Neustart des Dienstes versuchen, ein anderes Programm ausführen oder gar den Neustart des Rechners bewirken.

Nicht jeder Dienst muß seinen eigenen Prozeß haben. Dienste können sich auch einen Prozeß teilen. Das führt zu einer geringeren Ressourcenauslastung, aber auch dazu, daß alle Dienste, die einen Prozeß gemeinsam verwenden, beendet werden, wenn ein Fehler bei einem einzelnen Dienst den Prozeß beendet. Der SCM selbst beherbergt mehrere Dienste, unter anderem *SecLogon*, der innerhalb der aktuellen Benutzersitzung eine andere Benutzersitzung starten kann, und den *Event Logger*, der bestimmte Ereignisse in einem Protokoll, dem *Event Log*, festhält. *Lsass* beherbergt die Dienste *Samss*, *Netlogon* und *PolicyAgent*.

Die Dienststeuerungsprogramme (SCPs) sind ebenfalls normale Win32-Programme, die zusätzlich bestimmte SCM-Funktionen verwenden. Sie müssen dazu zuerst einen Kommunikationskanal zum SCM öffnen. Dies geschieht mit Hilfe der Funktion `OpenSCManager()`. Der SCM erstellt bei seiner Initialisierung ein internes Objekt, das die SCM-Datenbank darstellt. Eine Sicherheitsbeschreibung schützt den Zugriff auf dieses Objekt. So darf zwar jeder Benutzer Dienste auflisten, aber nur der Administrator darf Dienste installieren oder löschen.

Die SCPs müssen bei der Funktion `OpenSCManager()` angeben, welche Art von Zugriff sie auf die SCM-Datenbank wünschen. Der Objekt-Manager prüft dann, ob der Zugriff gewährt werden kann. Wenn ein Dienst geöffnet werden soll, muß das SCP ebenfalls den Zugriffswunsch angeben: Abrufen, Konfigurieren, Anhalten oder Starten des Dienstes. So wird gewährleistet, daß nur berechtigte Benutzer Dienste verwenden oder verändern.

Windows File Protection

Windows 2000 enthält einen Mechanismus, um bestimmte Systemdateien davor zu schützen, durch inkorrekte Versionen ausgetauscht oder überschrieben zu werden: *Windows File Protection* (WFP)[20]. Geschützt werden alle DLL, EXE, OCX und SYS Dateien, die von Windows 2000 installiert wurden sowie einige TTF Schriftfonts. Wenn eine Anwendung versucht, eine

[20] In [SR00] finden sich keine Hinweise zu WFP. Die hier angegebenen Informationen stammen von [MS01h].

geschützte Systemdatei auszutauschen, stellt WFP sie wieder mit Hilfe eines Caches her. Im Verzeichnis `%SystemRoot%\System32\Dllcache` befinden sich Kopien der geschützten Systemdateien und in `%SystemRoot%\Driver Cache\i386\Driver.cab` zusätzliche Kopien von geschützten Treiberdateien. Das Verzeichnis und die Größe des Caches lassen sich in der Registry einstellen.

Wenn eine geänderte Datei nicht wiederhergestellt werden kann, weil sich keine Kopie im *Dllcache* befindet, wird der Administrator nach der Original-Installations-CD gefragt, um sie von dort wiederherzustellen. Ist kein Administrator eingeloggt, erscheint diese Meldung, sobald sich ein Administrator lokal anmeldet. Zusätzlich macht WFP bei einem Austauschversuch einen entsprechenden Eintrag im *Event Log*.

Um inkorrekte Versionen erkennen zu können, verwendet WFP Dateisignaturen[21], die in einer Katalogdatei gespeichert werden und mit den aktuellen Signaturen verglichen werden. Wenn die aktuell berechnete Signatur nicht mit der gespeicherten übereinstimmt, wird die entsprechende Datei wiederhergestellt. Die Katalogdatei selbst wird mit dem gleichen Mechanismus geschützt, d.h. es befindet sich eine Kopie im *Dllcache*. WFP kann allerdings nur erkennen, ob sie fehlt oder beschädigt ist. Beim Booten generiert WFP eine Liste der Dateien, die sich im *Dllcache* befinden, so daß das Hinzufügen oder Entfernen von Dateien im *Dllcache* im laufenden Betrieb erkannt werden kann.

Ein offizieller Austausch von geschützten Systemdateien ist nur durch spezielle Programme möglich, die bei der Installation eines *Service Packs* oder *Hotfixes* oder beim Upgrade des Betriebssystems von Microsoft mitgeliefert werden[22].

2.4.3. Systemstart

Beim Start eines PCs lädt das BIOS zunächst den MBR (Master Boot Record) des Systemlaufwerks und übergibt die Steuerung an ihm. Der MBR lädt den Startsektor der aktiven Partition auf dem Laufwerk und übergibt diesem wiederum die Steuerung. Der Startsektor schließlich lädt und startet *Ntldr*, das Ladeprogramm von Windows 2000. *Ntldr* liest die Datei Boot.ini zur Anzeige des Startmenüs und lädt Ntdetect.com, um das BIOS nach Geräten und der Konfiguration abzufragen. Danach wird der Kernel (Ntoskrnl.exe) und die HAL (Hal.dll) geladen, aber noch nicht ausgeführt. Erst werden noch die *Bootstart*-Treiber und der Dateisystemtreiber, der für die aktive Partition verantwortlich ist, geladen. In der *Registry* ist der Startwert für jeden Treiber definiert, aus dem ersichtlich ist, ob es sich um einen *Bootstart*-Treiber handelt oder

[21] Über das verwendete Verfahren ließen sich keine Informationen finden.

[22] Offensichtlich gibt es undokumentierte Funktionen, die einen Austausch von geschützten Systemdateien möglich machen.

nicht. Anschließend wird die Kontrolle an Ntoskrnl.exe übergeben.

Ntoskrnl.exe arbeitet in zwei Phasen. In der ersten Phase werden zunächst die Interrupts deaktiviert, damit die Systeminitialisierung vollständig ablaufen kann. Dabei werden die Datenstrukturen des Kernels initialisiert und dann die HAL. Als letztes werden die grundlegenden Datenstrukturen der Ausführungsschicht erstellt.

In der zweiten Phase werden die Interrupts wieder aktiviert. Dann erfolgt eine Initialisierung der Systemzeit, der Energieverwaltung und zusätzlicher Prozessoren, sofern vorhanden. Es werden Objekttypen für die Synchronisation erstellt (Semaphore, Mutexe, Zeitgeber und Ereignisse) und die Systemunterstützungsbibliothek Ntdll.dll in den Systemadreßraum geladen. Die restlichen Teile der Ausführungsschicht werden initialisiert. Wenn der I/O-Manager initialisiert wird, veranlaßt er das Laden der *Systemstart*-Treiber und aller Treiber für Geräte, die der PnP-Manager erkannt hat, und startet deren Initialisierungsroutinen[23]. Danach wird der Sitzungsmanager gestartet.

Der Sitzungsmanager startet bestimmte Programme, die beim Booten gestartet werden sollen. Standardmäßig ist dies die Datenträgerüberprüfung (Chkdsk.exe). Bekannte DLLs werden geöffnet, und mit Hilfe des Konfigurationsmanagers wird die *Registry* im Arbeitsspeicher aufgebaut. Der Sitzungsmanager startet das Win32-Teilsystem (Win32k.sys und Csrss.exe) und Winlogon. Die Kontrolle wird an Winlogon übergeben.

Winlogon erstellt die ersten Arbeitsstations- und Desktopobjekte, den Dienststeuerungsmanager-Prozeß und lädt die GINA-DLL (*Graphical Identification and Authentication Interface*) für die Benutzeranmeldung. Der Dienststeuerungsmanager lädt alle Dienste und Treiber mit dem Startwert *Autostart*. Winlogon erstellt zudem den Prozeß für *Lsass*. Da der Dienststeuerungsmanager unter Umständen auch *Lsass* benötigt, erfolgt eine gleichzeitige Initialisierung der beiden Prozesse. Ab hier wartet Winlogon darauf, daß sich ein Benutzer anmeldet.

Das Beenden des Systems (*Shutdown*) läuft in umgekehrter Reihenfolge: Nachdem ein Benutzer den Befehl zum Herunterfahren des Systems gegeben hat, wird Winlogon damit beauftragt, den *Shutdown* durchzuführen. Winlogon schickt an *Csrss* eine Nachricht, erst alle Benutzermodus-Prozesse (außer der System-Unterstützungsprozesse) zu beenden und dann alle Systemprozesse. Wenn dies geschehen ist, veranlaßt Winlogon das Herunterfahren der Treiber und der restlichen Teilsysteme der Ausführungsschicht. Am Ende bleibt nur der Prozeß der Energieverwaltung übrig, der den Computer ausschaltet oder neu bootet.

[23] Der Vorgang des Ladens von Treibern wird nochmal genauer in Abschnitt 2.4.6 beschrieben.

2.4.4. Prozesse und Threads

Ein Prozeß stellt unter Windows 2000 die Umgebung zur Ausführung eines Programms dar. Ein Thread ist der Code des Programms, der tatsächlich ausgeführt wird. Ein Prozeß kann mehrere Threads gleichzeitig besitzen, die dann parallel ausgeführt werden.

Prozesse

Zu jedem Prozeß existiert im Kernelmodus eine spezielle Datenstruktur namens EPROCESS, in der die Prozeß-ID, Informationen zur binären Programmdatei, ein Zeiger auf den *primären Zugriffstoken*, auf die Handletabelle und auf eine Modulliste verwaltet werden. In der Modulliste stehen alle benötigten DLLs des Prozesses.

Zur Prozeßerzeugung wird die Funktion `CreateProcess()` oder eine ihrer Varianten in Kernel32.dll aufgerufen. Diese Funktion überprüft zunächst, ob in der *Registry* unter *HKLM\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\ImageFileExecutionOptions* ein Eintrag mit dem Namen der Programmdatei vorhanden ist. Ist dies der Fall, dann wird statt dessen das Programm ausgeführt, das dort im Feld *Debugger* eingetragen ist.

Für jeden neu gestarteten Prozeß muß es einen übergeordneten Prozeß geben, einzige Ausnahme bilden Prozesse des Systemstarts. Dies ist notwendig, damit der neue Prozeß über einen Sicherheitskontext verfügen kann, denn er erbt den Zugriffstoken seines übergeordneten Prozesses. Alternativ kann beim Aufruf der Funktion `CreateProcessAsUser()` ein anderer Zugriffstoken übergeben werden.

Der Startthread des Prozesses wird zunächst im angehaltenen Zustand erstellt, die Ausführung beginnt erst, wenn der Prozeß vollständig initialisiert wurde. Denn zuvor wird das Win32-Teilsystem über den neuen Prozeß informiert, und alle für den Prozeß benötigten DLLs werden geladen. Danach wird das Programm schließlich im Benutzermodus ausgeführt.

Threads

Zu jedem Thread existiert im Kernelmodus eine Datenstruktur ETHREAD, in der Informationen zur Threadidentifizierung, Prozeßidentifizierung, zum Identitätswechsel, zu LPC-Nachrichten und zu anhängigen I/O-Anforderungen stehen. Außerdem ist ein Zeiger auf die EPROCESS-Datenstruktur des zugehörigen Prozesses und ein Zeiger auf einen Zugriffstoken enthalten.

Ein neuer Thread wird durch Aufruf der Funktion `CreateThread()` erzeugt. Diese Funktion erstellt im Adreßraum des zugehörigen Prozesses einen Stack für den Thread und initialisiert den Hardware-Kontext, d.h. Daten in bestimmten Prozessorregistern. Mit `CreateRemoteThread()` besteht alternativ die Möglichkeit, einen Thread in einem anderen Pro-

zeß zu starten – sofern man die Berechtigung dafür hat.

Das Win32-Teilsystem wird über den neuen Thread informiert. Daraufhin kann mit der Ausführung des Threads begonnen werden. Die restliche Threadinitialisierung findet im Kontext des neuen Threads statt, wozu das Benachrichtigen der geladenen DLLs über den neuen Thread gehört. Der Thread wird sodann im Benutzermodus ausgeführt.

Zeitplanung der Threadausführung Windows 2000 verwendet eine *prioritätsgesteuerte Zeitplanung* bei der Threadausführung. Die Zeitplanung erfolgt unabhängig davon, zu welchen Prozessen die einzelnen Threads gehören. Ein bereiter Thread mit höchster Priorität wird jeweils für die Dauer einer Zeitscheibe (Quantum) ausgeführt. Aufgrund der *präemptiven Ablaufsteuerung* kann es bei der Ausführung allerdings auch Unterbrechungen geben, bevor der Thread sein Quantum aufgebraucht hat, z.B. weil ein Thread mit einer höheren Priorität ausführungsbereit wird. Die Ablaufsteuerung ist im Kernel implementiert. Während sie aktiv ist, kann kein Thread ausgeführt werden, so daß die Informationen zur Ablaufsteuerung nicht geändert werden können.

Auftragsobjekte

Auftragsobjekte sind Kernelobjekte zur Steuerung mehrerer **Prozesse als Gruppe**. Ein Prozeß kann nur zu einem einzigen Auftrag gehören. Alle Nachkommen eines Prozesses aus einem Auftragsobjekt werden ebenfalls demselben Auftragsobjekt zugeordnet. Damit sind gemeinsame Einstellungen für alle zugehörigen Prozesse möglich, wie z.B. ein gemeinsamer Zugriffstoken. So erzeugt der Dienst *SecLogon* ein Auftragsobjekt, das alle von ihm erstellten Prozesse unter der angegebenen Benutzerkennung aufnimmt.

2.4.5. Sicherheitssystem

Komponenten des Sicherheitssystems

Zum Sicherheitssystem von Windows 2000 gehört der **Sicherheitsreferenzmonitor** (SRM). Er ist verantwortlich für die Berechtigungsprüfung bei Objektzugriffen, für die Überwachung von Benutzerrechten und für die Ausgabe von Überwachungsnachrichten.

Der **Lokale Sicherheits- und Authentifizierungsserver** (*Lsass*) verwaltet die lokalen Sicherheitsrichtlinien, ist für die Benutzerauthentifizierung verantwortlich und sendet Überwachungsnachrichten an das Ereignisprotokoll. Die *Lsass-Richtliniendatenbank* befindet sich in der *Registry* unter *HKLM\SECURITY*. *Lsass* verwendet ein *Authentifizierungspaket* in Form einer DLL, um zu überprüfen, ob ein Benutzername und ein Kennwort zusammengehören. Für

Anmeldungen über ein Netzwerk führt *Lsass* eine weitere DLL aus: den *Netzwerkanmeldedienst* (Netlogon.dll). Die *Sicherheitskontenverwaltung* (SAM) verwaltet die Daten über die lokalen Benutzer und Gruppen und deren Kennwörter. Sie ist in der Datei Samsrv.dll implementiert und wird von *Lsass* ausgeführt. Die *SAM-Datenbank* selber steht in der *Registry* unter *HKLM\SAM*.

Da *Lsass* im Benutzermodus ausgeführt wird, aber manche Kernelmodus-Programme mit *Lsass* kommunizieren müssen, gibt es den **Kernelsicherheitsgerätetreiber** (KSecDD). Er implementiert eine LPC-Schnittstelle zu *Lsass* für Kernelmodus-Programme.

Eine wichtige Komponente im Sicherheitssystem ist der **Anmeldeprozeß** (Winlogon), der die interaktive Anmeldesitzung verwaltet und den Shell-Prozeß des Benutzers startet. Zur Eingabe des Benutzernamens und Kennworts verwendet Winlogon die GINA-DLL, die standardmäßig in *%SystemRoot%\System32\Msgina.dll* implementiert ist.

Objektschutz

Windows 2000 implementiert eine **wahlfreie Zugriffskontrolle** für Objekte, d.h. der Besitzer eines Objekts legt fest, welche Benutzer Zugriff darauf haben. Zu den schützbaren Objekten gehören u.a. Dateien, Geräte, Auftragsobjekte, Prozesse, Threads und Ereignisse. Der Objekt-Manager ist für den Zugriffsschutz dieser Systemressourcen verantwortlich: Bei der Anforderung eines Objektes prüft er, ob der anfordernde Prozeß die Berechtigung für den Zugriff hat. Wenn der Zugriff nicht erlaubt ist, erhält der Prozeß kein Handle auf das Objekt.

Threads laufen immer im Kontext eines bestimmten Prozesses. Sie haben somit den gleichen Sicherheitskontext, d.h. nur die Berechtigungen, die auch der Prozeß hat. Threads können allerdings einen **Identitätswechsel** vornehmen. Sie laufen dann in einem anderen Sicherheitskontext. Die Zugriffskontrolle für Objekte erfolgt für diesen Thread anhand der neuen Identität. Der Thread hat aber nach wie vor auch noch Zugriff auf Objekte, die sein zugehöriger Prozeß angefordert und bekommen hat. Denn von Threads angeforderte Objekte werden alle gemeinsam in der Handletabelle ihres Prozesses eingetragen. Somit hat ein Thread Zugriff auf die Objekte, die ein anderer Thread des gleichen Prozesses angefordert hat.

Ein Identitätswechsel sorgt z.B. bei Client-/Serveranwendungen für eine transparente Sicherheitsüberprüfung: Der Server übernimmt zeitweise die Identität des Clients, der eine bestimmte Ressource anfordert. Der Objekt-Manager kann somit überprüfen, ob der Zugriff erlaubt ist oder nicht. Ein Identitätswechsel ist nur für einzelne Threads möglich, nicht für den gesamten Prozeß. Server können ohne Erlaubnis des Clients dessen Identität nicht annehmen.

Wenn Threads ein Objekt öffnen wollen, müssen sie im Vorfeld die gewünschte Art des Zugriffs angeben, z.B. *„nur Lesen“* oder *„Lesen und Schreiben“*. Während der Objekt-Manager

die Zugriffsprüfung durchführt, kann die Sicherheitsbeschreibung des betreffenden Objektes, in der die **Zugriffskontrollisten** (*Access Control Lists*, ACLs) stehen, nicht geändert werden, da eine Sperre darüber gelegt wird. Der Objekt-Manager ruft den SRM auf, der tatsächlich über den Zugriff entscheidet. Er entscheidet nach einer Sicherheitsgleichung, die aus dem Thread-Zugriffstoken, der gewünschten Zugriffsart und der Objekt-Sicherheitsbeschreibung besteht.

Die Zugriffsprüfung wird vom Objekt-Manager auch initiiert, wenn ein Prozeß schon über ein Handle auf das Objekt verfügt und darüber Bezug auf das Objekt nimmt. Denn alle Operationen auf Objekte sind im Benutzermodus nur über Handles möglich, und nur der Objekt-Manager kann Handles in echte Objektreferenzen auflösen.

Der Sicherheitskontext eines Prozesses oder Threads wird über einen **Zugriffstoken** beschrieben. Darin stehen Informationen über die zugehörigen Berechtigungen, Konten und Gruppen. Winlogon erstellt bei der Anmeldung eines Benutzers einen Zugriffstoken für dessen Shell-Prozeß. Alle Prozesse, die der Benutzer nun startet, erben diesen Token. Eine Ausnahme bilden die Prozesse, die mittels `CreateProcessAsUser()` gestartet wurden. Sie erhalten einen anderen Zugriffstoken. Wenn eine Anwendung allerdings diese Funktion verwenden will, muß sie die Anmeldedaten (Name und Kennwort) des betreffenden Benutzerkontos kennen, unter dem der neue Prozeß laufen soll.

Ein Zugriffstoken enthält zwei wichtige Komponenten: zum einen Felder mit den *SIDs* (*Security Identifiers*) des Benutzerkontos und der zugehörigen Gruppen, zum anderen *Privilegien*. Die SIDs benötigt der SRM, um damit Zugriffsberechtigungen auf Objekte zu prüfen. Die Privilegien beschreiben allgemein, was der Benutzer darf, z.B. den Rechner herunterfahren, Programme debuggen oder Treiber installieren. Es gibt drei Arten von Token: *Primärtoken* stellen den Sicherheitskontext eines Prozesses dar, *beschränkte Token* sind eine Kopie davon mit Einschränkungen, und *Identitätswechseltoken* dienen Threads, die ihren Sicherheitskontext wechseln wollen. *Lsass* ist normalerweise der einzige Prozeß, der Zugriffstoken erzeugt.

Die **Sicherheitsbeschreibung** eines Objektes enthält die SID des Besitzers und zwei ACLs: DACL und SACL. In der DACL stehen Einträge, die jeweils eine SID, eine Zugriffsmaske und Angaben für Vererbungseigenschaften enthalten. Es gibt zwei Typen von DACL-Einträgen: *access-allowed* und *access-denied*. In der Zugriffsmaske stehen die verschiedenen Arten des Zugriffs wie Lesen, Schreiben, Löschen. Die SID bezeichnet einen Benutzer oder eine Gruppe. Anhand dieser Informationen ist es dem SRM möglich zu erkennen, ob jemand z.B. das Recht hat, das betreffende Objekt zu lesen. Jeder Token enthält eine Standard-DACL, die jedes neu erstellte Objekt eines Prozesses bekommt, wenn keine spezifische DACL definiert wird.

Die SACL enthält Einträge zur Systemüberwachung und über das Systemüberwachungsobjekt. Damit kann festgelegt werden, welche Operationen des Benutzers überwacht werden sollen. Entsprechende Informationen werden dann ins Überwachungsprotokoll eingetragen.

Bei der Zugriffsüberprüfung wird die SID in einem ACL-Eintrag mit den SIDs im Zugriffstoken des Aufrufers verglichen. Bei Übereinstimmung kommt der ACL-Eintrag zur Anwendung, und die Entscheidung über die Zugriffsgewährung erfolgt anhand des Typs des Eintrags (*access-allowed* oder *access-denied*). Die Zugriffsgewährung ist von der relativen Reihenfolge der ACL-Einträge abhängig, d.h. der erste passende Eintrag kommt zur Anwendung. Wenn kein passender Eintrag gefunden wird, wird der Zugriff verweigert. Objekte können die ACL-Einträge von übergeordneten Objekten erben. Explizit gesetzte ACL-Einträge haben jedoch Vorrang vor geerbten Einträgen. Bei expliziten ACL-Einträgen stehen *access-denied*-Einträge immer vor *access-allowed*-Einträgen, so daß im Zweifelsfall der Zugriff verwehrt wird.

Die Zugriffsüberprüfung erfolgt immer beim Öffnen eines Handles auf ein Objekt. Wenn der Zugriff erlaubt wurde, dann bleibt diese Erlaubnis solange bestehen, wie das Handle offen ist. Erst wenn das Handle geschlossen und das Objekt erneut angefordert wird, wirkt sich eine zwischenzeitliche Änderung der DACL aus. Nur Besitzer eines Objekts können dessen DACL ändern. Sie haben somit praktisch immer Zugriff auf ihre Objekte. Der Administrator des Systems hat das *Inbesitznahmerecht*, d.h. er kann Besitzer jedes Objektes werden.

Für Programme, die im Kernelmodus ausgeführt werden, erfolgt keine Zugriffsüberprüfung. Sie können direkte Zeiger auf die Objekte verwenden und müssen keine Handles anfordern. Das bedeutet, daß z.B. Gerätetreiber uneingeschränkten Zugriff auf alle Systemressourcen haben!

Sicherheitsüberwachung

Der *Lsass*-Prozeß verwaltet die **Überwachungsrichtlinie**, in der steht, welche Ereignisse überwacht werden sollen. Alle Anwendungen und das Betriebssystem können *Überwachungsereignisse* erzeugen. Dies macht z.B. der Objekt-Manager, wenn ein Handle angefordert wird. Der Sicherheitsreferenzmonitor erstellt einen *Überwachungsdatensatz*, wenn ein Überwachungsereignis überwacht werden soll, und schickt diesen Datensatz über *Lsass* an den *Event Logger*, der ihn in das **Sicherheitsprotokoll** einträgt.

Benutzeranmeldung

Die Benutzeranmeldung erfolgt interaktiv über Winlogon. Andere Prozesse haben während der Anmeldung keinen Zugriff auf den Desktop. Winlogon ruft GINA auf, um Benutzername und Kennwort abzufragen, und anschließend *Lsass*, um sich diese Daten bestätigen zu lassen. *Lsass* verwendet dazu das Authentifizierungspaket und liest aus der Richtliniendatenbank, welche Privilegien der Benutzer hat. *Lsass* erstellt mit Hilfe der Ausführungsschicht einen Zugriffstoken, der dann an Winlogon übergeben wird. Winlogon ruft schließlich die Benutzer-Initialisierungsroutine auf, die den Shell-Prozeß startet.

2.4.6. I/O-System

I/O-Systemkomponenten

Zum I/O-System von Windows 2000 gehören mehrere Komponenten. Der *I/O-Manager* stellt das wesentliche Verbindungsglied zwischen Anwendungen und Geräten dar. *Gerätetreiber* bilden die Schnittstelle zwischen I/O-Manager und dem eigentlichen Gerät. Der *PnP-Manager* steuert die Zuweisung von Hardwareressourcen wie Interrupts und Speicheradressen an die Geräte und beauftragt den I/O-Manager mit dem Laden von Treibern, wenn neue Geräte erkannt wurden. Die *Energieverwaltung* ist für die Steuerung des Energiesparmodus von einzelnen Geräten verantwortlich. Mit Hilfe der *Windows Management Instrumentation* können Gerätetreiber Daten an Verwaltungsanwendungen wie den Systemmonitor liefern. Die *HAL* abstrahiert Unterschiede des Prozessors und des Interruptcontrollers vor Treibern und stellt dafür eine API zur Verfügung. Die Verwendung dieser API ist allerdings freiwillig, denn Treiber im Kernelmodus können auch direkt auf die Hardware zugreifen. In der *Registry* stehen die Informationen über die Hardware und die Einstellungen für die Treiber.

Es gibt unter Windows 2000 verschiedene Treibertypen:

- *Dateisystemtreiber* arbeiten I/O-Anforderungen an Dateien ab. Sie wandeln dateibezogene Adressen in datenträgerbezogene Adressen um.
- *Bildschirmtreiber* implementieren die Zeichenoperationen. Sie schreiben direkt in den Speicher der Grafikkarte oder senden Befehle an den Grafikchip.
- *WDM-Treiber* stellen die restlichen Treiber dar, die sich weiter untergliedern in:
  - *Bustreiber*, die die verschiedenen Geräte an einem Bus[24] verwalten.
  - *Funktionstreiber*, die die eigentliche Funktionalität eines Geräts implementieren.
  - *Filtertreiber*, die das Verhalten eines anderen Treibers verändern.

Treiber werden normalerweise im Kernelmodus ausgeführt. Es gibt allerdings auch Treiber, die im Benutzermodus ausgeführt werden: virtuelle Gerätetreiber für DOS-Programme und Druckertreiber.

Gerätetreiber sind keine Programme im herkömmlichen Sinne, sondern eine Kollektion von Routinen, die je nach Bedarf vom I/O-System aufgerufen werden. Beim Systemstart wird beispielsweise vom I/O-Manager die Initialisierungsroutine jedes geladenen Treibers aufgerufen. Treiber können im laufenden Betrieb vom I/O-Manager geladen oder entladen werden.

[24] Physische Geräte werden im PC an einem Bus verwaltet. So gibt es für Speicherlaufwerke den EIDE- oder SCSI-Bus, für Erweiterungskarten den ISA- oder PCI-Bus und für externe Geräte den USB-Bus.

I/O-Datenstrukturen

Alle I/O-Anforderungen von Anwendungen werden als virtuelle Dateioperationen ausgeführt. Der I/O-Manager leitet die Daten von der virtuellen Datei zum zuständigen Gerätetreiber und zurück. Für den Zugriff auf Dateien oder Geräte werden im Kernelmodus Datenstrukturen namens **Dateiobjekte** verwaltet. Sie repräsentieren die Ressourcen, enthalten die Daten der Ressource aber nicht selbst. Ein Dateiobjekt kann gleichzeitig von mehreren Benutzermodus-Prozessen genutzt werden. Wenn eine Anwendung eine Datei öffnet, liefert der Objekt-Manager ein Handle auf solch ein Dateiobjekt zurück. Der Zugriff wird, wie bei anderen Objekten auch, durch eine Sicherheitsbeschreibung geschützt.

Bei der Anforderung eines Handles auf ein Dateiobjekt muß der I/O-Manager nachprüfen, welcher Treiber für die Bearbeitung verantwortlich ist. **Treiberobjekte** repräsentieren hierzu einen einzelnen Treiber. Sie enthalten die Adressen der verschiedenen Routinen, die ein Treiber implementiert, und werden erstellt, wenn der entsprechende Treiber vom System geladen wird. Treiber können ihrerseits **Geräteobjekte** erstellen, die die Geräte oder Schnittstellen des Treibers repräsentieren. Das Treiberobjekt enthält eine Liste der Geräteobjekte, die der Treiber steuert. Wenn eine Datei geöffnet werden soll, kann anhand des Dateinamens das betreffende Geräteobjekt gefunden werden. So übersetzt der I/O-Manager beispielsweise die Anfrage nach `A:\datei.dat` in `\Device\Floppy0\datei.dat`. Das Geräteobjekt `Floppy0` ist somit für die angeforderte Datei zuständig. Da Geräteobjekte wiederum auf ihren Treiber verweisen, kann der I/O-Manager erkennen, welchen Treiber er aufrufen muß.

I/O-Anforderungspakete (*I/O Request Packets*, IRPs) repräsentieren die einzelnen I/O-Operationen, die Anwendungen anfordern. Der I/O-Manager erstellt bei einer Anforderung eine IRP und speichert dort einen Zeiger auf das betreffende Dateiobjekt. So ergibt sich eine Verweiskette vom IRP über das Dateiobjekt auf das Geräteobjekt und von da auf das Treiberobjekt und schließlich auf den Treiber. Im IRP wird auch gespeichert, welche konkrete Funktion des Treibers aufgerufen werden soll. Ein IRP wird in eine Warteschlange eingereiht. Wenn es zur Bearbeitung freigegeben wird, ruft der I/O-Manager die entsprechende Treiberroutine auf und übergibt das IRP dem Treiber. Nach Abschluß der Operation wird das IRP gelöscht.

I/O-Verarbeitung

Eine normale I/O-Operation läuft in der Form ab, daß eine Benutzermodus-Anwendung über die Teilsystem-DLLs eine I/O-Anforderung stellt. Diese wird an den I/O-Manager übergeben, der ein IRP erstellt und es an den betreffenden Gerätetreiber weiterreicht. Der Gerätetreiber greift schließlich entweder indirekt über die HAL oder direkt auf die Hardware zu und bearbeitet die I/O-Anforderung.

Treiber können zur Bearbeitung von I/O-Anforderungen aber auch andere Treiber aufrufen. Die Bearbeitung einer I/O-Anforderung ist für ein Gerät oft auf mehrere verschiedene Treiber verteilt. Es gibt *Geräteklassentreiber* (z.B. für Festplatten), *Anschlußtreiber* (z.B. für den SCSI-Bus) und *Miniport-Treiber* (z.B. für eine konkrete SCSI-Adapterkarte). Die gegenseitige Kommunikation der Treiber erfolgt über den I/O-Manager mittels IRPs.

Durch diese Zusammenarbeit von Treibern entsteht für jedes Gerät eine *Treiberhierarchie*. Zusätzliche Treiber können ohne Probleme in eine bestehende Hierarchie eingefügt werden. Operationen eines Treibers auf der unteren Stufe können von einem Treiber höherer Stufe gefiltert werden. So können z.B. virtuelle Laufwerke implementiert werden.

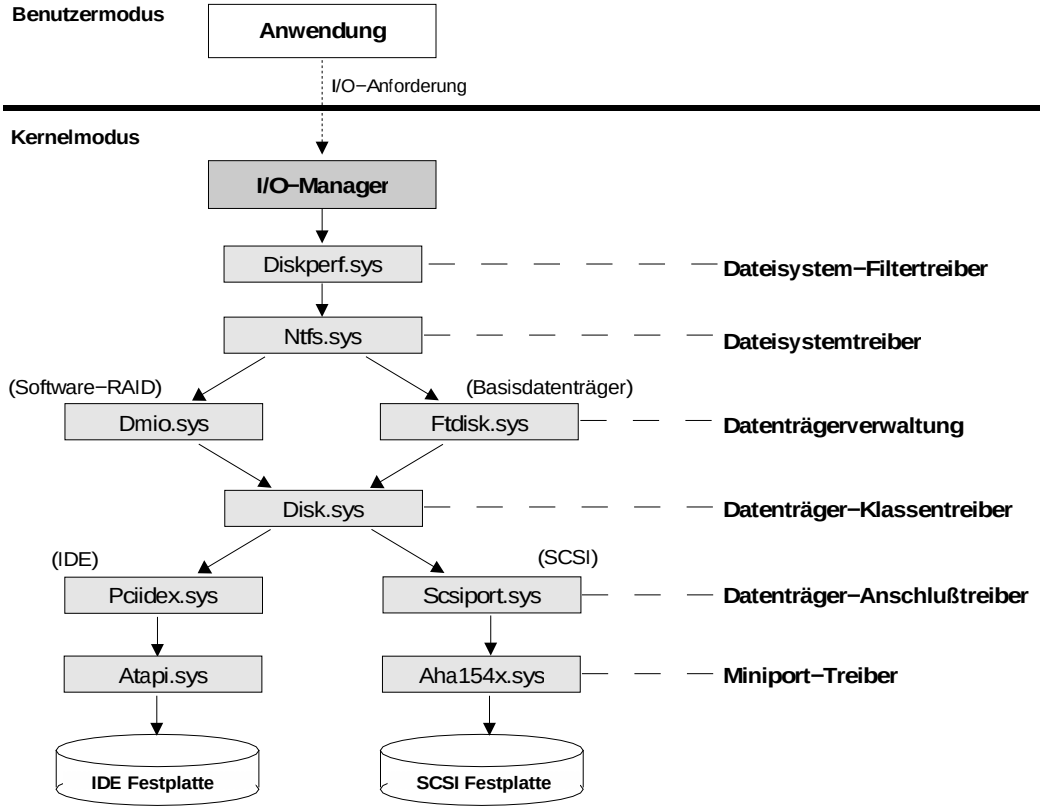

Abbildung 2.2.: Treiberhierarchie einer I/O-Anforderung in Windows 2000.

Die I/O-Operation wird in umgekehrter Reihenfolge abgeschlossen, d.h. das Gerät sendet Daten zurück an den untersten Treiber, der unterste Treiber gibt sie an den nächsthöheren Treiber usw., bis schließlich der I/O-Manager die Daten der anfordernden Anwendung übergibt.

Die I/O-Verarbeitung kann noch beschleunigt werden. Es gibt für Treiber die Anweisung „*Fast I/O*“, mit der eine I/O-Anforderung ausgeführt werden kann, ohne daß ein IRP erstellt wird. Mit einer speziellen Form der Abschnittsobjekte, den *Speicherzuordnungsdateien*, kann eine Abbildung einer Datei im Adreßraum eines Prozesses angelegt werden. Auf die Daten

der Datei kann auf diese Weise direkt zugegriffen werden, ohne jedesmal eine I/O-Operation ausführen zu müssen. I/O-Anforderungen können statt an einen Dateisystemtreiber auch direkt an einen Datenträgerverwaltungstreiber geschickt werden, um z.B. gelöschte Dateien wiederherzustellen. Andererseits arbeiten Dateisystemtreiber eng mit dem Cachemanager und der Speicherverwaltung zusammen, so daß bereits gelesene Daten evtl. aus dem Cache geladen werden können.

Laden und Installieren von Treibern

Alle installierten Treiber sind in der *Registry* unter *HKLM\SYSTEM\CurrentControlSet\Services* aufgelistet. Dort befinden sich die individuellen Einstellungen zum Ladevorgang der Treiber. Es gibt fünf mögliche Startwerte:

- SERVICE_BOOT_START: *Ntldr* lädt den Treiber direkt beim Booten.
- SERVICE_SYSTEM_START: Der Treiber wird nach den *Bootstart*-Treibern vom I/O-Manager geladen.
- SERVICE_AUTO_START: Der SCM veranlaßt das Laden des Treibers.
- SERVICE_DEMAND_START: Diese Einstellung ist für *Plug-and-Play*-Treiber, die nicht zwingend beim Systemstart benötigt werden.
- SERVICE_DISABLED: Der Treiber ist deaktiviert und wird nicht geladen.

Ein Treiber kann *explizit* geladen werden oder *auflistungsbasiert*. Beim expliziten Laden wird der Treiber beim Booten von *Ntldr* oder beim Systemstart von der Ausführungsschicht geladen. Die Ladereihenfolge der einzelnen Treiber ist in der *Registry* festlegbar. Beim auflistungsbasierten Laden wird der Treiber automatisch vom PnP-Manager geladen, wenn die entsprechenden Geräte am Bus aufgelistet werden. Der genaue Ladevorgang der Treiber sieht wie folgt aus:

1. ***Ntldr* lädt beim Booten alle *Bootstart*-Treiber.** Dazu gehören alle Bustreiber und ein Dateisystem-Erkennungstreiber. Wenn die Ausführungsschicht initialisiert wird, startet der I/O-Manager die Initialisierungsroutine der geladenen Treiber. Die Bustreiber melden dem PnP-Manager, welche Geräte an ihrem Bus angeschlossen sind, und der PnP-Manager erstellt daraufhin einen *Gerätebaum*, der die Geräte-Beziehungen repräsentiert.
2. **Der I/O-Manager ruft den PnP-Manager auf und startet das auflistungsbasierte Laden.** Der PnP-Manager lädt für alle aufgelisteten Geräte die entsprechenden Treiber. Dies geschieht rekursiv auch für untergeordnete Geräte und ungeachtet der in der *Registry* eingetragenen Startwerte für die Treiber (ausgenommen deaktivierte Treiber).

3. **Der PnP-Manager lädt alle *Systemstart*-Treiber**, sofern sie nicht schon geladen wurden. Wenn dadurch weitere Geräte gemeldet werden, lädt der PnP-Manager auch dafür die erforderlichen Treiber.

4. **Der SCM lädt alle *Autostart*-Treiber.**

Die Knoten des Gerätebaums, der vom PnP-Manager erstellt wird, werden *Geräteknoten* genannt. Sie enthalten Zeiger auf die entsprechenden Geräteobjekte. Über die Geräteknoten ist somit für den I/O-Manager ersichtlich, ob oberhalb eines Funktions- oder Bustreibers ein Filtertreiber installiert ist. Die gesamte Treiberhierarchie für ein Gerät wird innerhalb eines Geräteknotens abgebildet.

Die Bustreiber geben bei der Geräteauflistung dem PnP-Manager *Gerätekennungen* zurück. Mit Hilfe dieser Kennungen kann der PnP-Manager aus der *Registry* lesen, welche Funktions- und Filtertreiber für ein Gerät geladen werden müssen. Die entsprechenden Einträge finden sich unter den Schlüsseln *HKLM\SYSTEM\CurrentControlSet\Enum* und *HKLM\SYSTEM\CurrentControlSet\Control\Class*.

Dateisystemtreiber sind eine Besonderheit. Zum einen unterscheiden sie sich von anderen Treibern dadurch, daß sie sich beim I/O-Manager als Dateisystemtreiber explizit anmelden müssen. Sie werden nicht in Geräteknoten verwaltet, sondern in einer eigenen Liste des I/O-Managers. Zum anderen wird beim Systemstart der Dateisystem-Erkennungstreiber (`Fs_rec.sys`) als Ersatztreiber für alle Dateisysteme geladen. Dieser überprüft beim ersten Zugriff auf einen Datenträger dessen Dateisystem und lädt erst dann den zugehörigen Dateisystemtreiber.

Die Installation eines Treibers erfolgt automatisch, wenn vom System ein neues Gerät erkannt wird. Es wird dann nach einer passenden *INF-Datei* gesucht, in der die Funktionstreiber angegeben sind und die Einstellungen für die Schlüssel *Enum* und *Class* in der *Registry*. Die INF-Datei wird gewöhnlich zusammen mit den Treiberdateien vom Gerätehersteller geliefert. Sie kann zudem Anweisungen enthalten, daß bei der Treiberinstallation bestimmte DLLs geladen und bestimmte Routinen ausgeführt werden sollen. Dies ist gedacht, um z.B. Geräteeinstellungen vom Benutzer abzufragen. Vor der tatsächlichen Installation eines Treibers prüft der PnP-Manager in der *Registry* die *Systemrichtlinie zur Treibersignierung*. Dort ist festgelegt, ob nur signierte Treiber angenommen werden sollen oder auch unsignierte[25].

Treiber können auch manuell von Setup-Programmen des Herstellers installiert werden. Die Setup-Programme müssen dazu explizit aufgerufen werden und kopieren dann normalerweise die erforderlichen Dateien in die entsprechenden Verzeichnisse und tragen die Einstellungen in

[25] Hardware-Hersteller liefern häufig unsignierte Treiber mit, da die Signierung von der Firma Microsoft durchgeführt wird und Geld kostet. Somit werden in der Praxis wohl oft unsignierte Treiber installiert, so daß diese Systemrichtlinie kein Garant für Sicherheit sein kann.

der *Registry* ein. Bei dieser Art der Installation findet allerdings keine Prüfung von Treibersignaturen statt.

Standardmäßig können nur Administratoren Treiber installieren. Das Recht zum Laden von Treibern kann jedoch auch anderen Benutzern zugeordnet werden.

Dateisystem NTFS

Das Standard-Dateisystem von Windows 2000 ist NTFS. Nur mit diesem Dateisystem sind Zugriffsrechte und Zugriffsüberwachung auf Datei- und Verzeichnisebene möglich. Denn die Sicherheitsbeschreibungen werden im NTFS-Dateisystem auf dem Datenträger gespeichert, wo sie von Anwendungen nicht direkt erreichbar sind. Der NTFS-Dateisystemtreiber liefert sie dem SRM, wenn eine Zugriffsprüfung gemacht werden muß.

NTFS kann Anwendungen über Änderungen an Dateien informieren. Datei- und Verzeichnisänderungen können im *Änderungsjournal* aufgezeichnet werden. Das Änderungsjournal wird in einer internen Metadatendatei im NTFS gespeichert und ist somit vor direkten Zugriffen von Anwendungen geschützt. Die Journalgröße wird bei Bedarf dynamisch erweitert. Allerdings verkleinert NTFS das Journal wieder, wenn eine bestimmte Größe überschritten wird. Ältere Aufzeichnungen werden dann überschrieben.

Encrypting File System

NTFS bietet mit Hilfe des Dateisystemfiltertreibers `efs.sys` ein verschlüsseltes Dateisystem, das *Encrypting File System* (EFS), an, welches auf einem hybriden Verfahren aus symmetrischer und asymmetrischer Verschlüsselung beruht: Dateien werden symmetrisch mit dem Verfahren DESX verschlüsselt, der Schlüssel (*File Encryption Key*, FEK) wird zufällig gewählt und mit dem asymmetrischen Verfahren RSA verschlüsselt zusammen mit der Datei abgespeichert.

Für jeden Benutzer wird einmalig ein RSA-Schlüsselpaar generiert. Der private Schlüssel wird mit dem Kennwort des Benutzerkontos gesperrt. Die Entschlüsselung läuft für angemeldete Benutzer mit passenden privaten Schlüsseln transparent ab. Für jede verschlüsselte Datei wird allerdings auch der FEK mit dem öffentlichen Schlüssel des *Recovery Operators* (standardmäßig der Administrator) verschlüsselt abgespeichert. Dadurch hat der Administrator jederzeit Zugriff auf alle verschlüsselten Dateien im EFS.

3. Bedrohungsszenario

Nachdem eine sicherheitskritische Anwendung auf einem produktiven System installiert wurde, ist sie mehreren Bedrohungen ausgesetzt. Schädliche Programme können auf dem System aktiv sein, wie z.B. Trojanische Pferde oder Viren. Diese Bedrohungen sind besonders für kryptographische Anwendungen sehr gefährlich, wenn die Sicherheit allein auf den kryptographischen Algorithmen beruht. Auch die Architektur der Anwendung und die Integration ins Betriebssystem muß sicher sein.

Im folgenden wird davon ausgegangen, daß eine kryptographische Anwendung als Java-Programm auf einem Windows 2000 System eingesetzt werden soll. Für die weitere Betrachtung werden zunächst einige Annahmen gemacht.

3.1. Annahmen

Die erste Annahme dient dazu, Programmierfehler der Java Virtual Machine (JVM) oder deren Ausführungsumgebung ausschließen zu können, die das System angreifbar machen könnten, wie z.B. durch Buffer-Overflow-Angriffe.

Annahme 1: Das Betriebssystem und die JVM sind korrekt implementiert.

Die Annahme scheint dahingehend gerechtfertigt, daß es zu Programmfehlern nach einiger Zeit oft Bugfixes gibt, so daß betreffende Sicherheitslöcher beseitigt werden können. Zumal ist es die Aufgabe des Herstellers des Betriebssystems bzw. der JVM für deren Korrektheit zu sorgen. In dieser Arbeit geht es um die konzeptionellen Sicherheitsprobleme.

Annahme 2: Das Verhalten des Benutzers ist vertrauenswürdig.

Ein Mißbrauch des Programms durch den Benutzer selbst wird nicht behandelt. Es kann zudem davon ausgegangen werden, daß der Benutzer das Programm auf einem sicheren Wege erhält. Etwaige Manipulationen der Programmdateien, bevor der Benutzer diese installiert hat, werden ausgeschlossen.

Annahme 3: Die Programmdistribution der kryptographischen Anwendung ist sicher.

Die kryptographische Stärke der verwendeten Algorithmen der Anwendung sei sicher gegenüber Angriffen. Das heißt, bei geeigneter Wahl der Schlüssellänge kann ein Angreifer eine verschlüsselte Nachricht nicht in angemessener Zeit entschlüsseln, ohne den Schlüssel zu besitzen. Auch die Qualität der Schlüsselpaare bei Public-Key-Verfahren sei so beschaffen, daß sich der private Schlüssel nicht in angemessener Zeit durch einen Angreifer berechnen läßt.

Annahme 4: Die verwendeten Algorithmen sind kryptographisch sicher.

Zum Installationszeitpunkt der kryptographischen Anwendung sei ein „sauberes“ System vorliegend, d.h. alle Komponenten des Systems sind korrekt implementiert und enthalten keine Trojanischen Pferde.

Annahme 5: Die Umgebung ist vor und während der Installation vertrauenswürdig.

Diese Annahme gilt jedoch nicht nach der Installation. Es muß davon ausgegangen werden, daß sich später Trojanische Pferde auf dem System befinden können, möglicherweise aufgrund einer Installation von Programmen aus unseriösen Quellen.

3.2. Bedrohungen

Bedrohungen für eine kryptographische Anwendung ergeben sich zum einen aus Angriffen direkt auf die Anwendung selbst und zum anderen aus Angriffen auf ihre Ausführungsumgebung.

3.2.1. Angriffe auf die Anwendung

Durch eine Installation zusätzlicher, nicht vertrauenswürdiger Programme kann es dazu kommen, daß sich ein Trojaner auf dem System befindet. Dieser Trojaner könnte versuchen, einen direkten Angriff auf die kryptographische Anwendung zu fahren. Es gilt zu verhindern, daß der Trojaner durch geschicktes Ausnutzen von verschiedenen Funktionsaufrufen der kryptographischen Anwendung oder der JVM an die privaten Schlüssel der Programme gelangt.

Ein Trojaner kann wie ein normales Anwendungsprogramm agieren und das Betriebssystem oder die JVM und deren API normal benutzen. Das heißt, es kommt hier zunächst nur die intendierte Semantik vorhandener API-Methoden zur Anwendung.

Mögliche Angriffe wären beispielsweise die Überwachung des Login-Fensters, um das Paßwort des Benutzers im Verschlüsselungsprogramm auszuspähen. Ein Trojaner könnte sich daraufhin als der Benutzer beim Programm anmelden und dessen privaten Schlüssel „stehlen“ oder

in dessen Namen bestimmte Dateien signieren und per E-Mail verschicken, z.B. einen Kaufvertrag. Dies kann natürlich nicht im Sinne des Benutzers sein.

Wenn man dies verallgemeinert, so sind alle Angriffe denkbar, die versuchen, Informationen auszuspähen. Wenn z.B. in einem Java-Objekt der private Schlüssel gespeichert ist oder ein Paßwort oder der Startwert des Zufallszahlengenerators, dann könnte ein Trojaner versuchen, diese Daten zu erlangen, indem er entsprechende Methoden aufruft, die eigentlich für andere Objekte bestimmt sind. Wenn das Betriebssystem zudem keine geeigneten Speicherschutzmechanismen bietet, können native (nicht in der JVM ablaufende) Anwendungen versuchen, den Speicher der JVM auszulesen oder gar zu manipulieren.

Vorstellbar ist auch der Austausch von Programmkomponenten vor dem Programmstart, so daß sich „Trojaner-Klassen“ innerhalb der Anwendung befinden. Angriffe, mit denen man hier rechnen muß, sind z.B. das Übergeben einer falschen Datei, wenn ein Java-Programm eine Datei lesen will. Beim Schlüsselimport könnte ein anderer öffentlicher Schlüssel übergeben werden, als der Benutzer ursprünglich aufnehmen wollte. Dieser Schlüssel wird nun unter falschem Namen im Programm verwaltet. Wenn der Benutzer damit eine Nachricht verschlüsselt, kann sie der Adressat nicht mehr lesen. Damit ist unter Umständen ein *Man-in-the-Middle-Attack* beim Nachrichtenaustausch möglich, wenn z.B. auch der E-Mail-Versand von dem Trojaner überwacht werden kann. Eine analoge Bedrohung besteht beim Schlüsselexport.

Die Schlüsselerzeugung oder allgemein die Generierung von Zufallszahlen kann durch Lieferung falscher Daten ebenso angegriffen werden, beispielsweise wenn die Systemuhrzeit oder die Speichergröße vom Betriebssystem als Zufallswerte geliefert werden sollen. Die Verschlüsselung kann manipuliert werden, wenn die Verschlüsselungsroutine selbst durch einen Trojaner ersetzt wird, der die Verschlüsselung z.B. mit einem anderen Schlüssel als angegeben durchführt.

3.2.2. Angriffe auf die Ausführungsumgebung

Der direkte Angriff auf die Anwendung ist nicht die einzige Möglichkeit. Ein Angriff kann indirekt über das verwendete Betriebssystem erfolgen. Es wurde zwar davon ausgegangen, daß zur Installationszeit alle Komponenten des Systems korrekt sind. Es ist aber durchaus möglich, daß dieser Zustand nicht so bleibt. Ein Trojaner könnte wichtige Systemkomponenten austauschen, um andere Programme oder Module (Gerätetreiber, DLLs) durch eigene Versionen zu ersetzen. Betriebssystemaufrufe von der JVM würden sich auf diese Weise abfangen oder modifizieren lassen.

Wenn man davon ausgeht, daß einem Angreifer der Quellcode zu der kryptographischen Anwendung bekannt ist (z.B. über *Reverse Engineering* oder bei *Open Source*), dann ist auch

bekannt, aus welchen Daten Zufallszahlen für die Schlüsselgenerierung gewonnen werden. Sei die Quelle dieser Daten die Soundkarte, die an ihrem Mikrofoneingang das natürliche Rauschen der Umgebung aufnimmt. Um diese Daten zu bekommen, wird letztendlich der Soundkarten-Treiber aufgerufen, der die Rohdaten an die Anwendung liefert. Wird dieser Treiber durch einen Trojaner ausgetauscht, stehen einem Angreifer die Rohdaten und, da der Quellcode der Programme bekannt ist, auch das Verfahren zur Weiterverarbeitung der Rohdaten zur Verfügung. Der Angreifer würde damit die gleichen Informationen zur Generierung der Schlüssel besitzen wie die kryptographische Anwendung. Berechnet er mit diesen Informationen den privaten Schlüssel, kann er Nachrichten entschlüsseln und digitale Signaturen fälschen.

Bei einem Softwareupdate könnte ein Trojaner auch die JVM oder Teile davon durch eigene Versionen ersetzen. Damit wäre denkbar, daß ein Angreifer Wissen über die internen Zustände der kryptographischen Anwendung bekommt. Die privaten Schlüssel wären einfach auslesbar, oder der Ablauf der Java-Programme könnte manipuliert werden, da sie jetzt in einer Umgebung laufen, die vom Angreifer kontrolliert wird.

Als Beispiel einer kryptographischen Anwendung sei die Smartcard für die digitale Signatur betrachtet. Die geschilderten Angriffe haben skizziert, daß es möglich wäre, andere Daten an die Smartcard zur Bildung der digitalen Signatur zu schicken, als dem Benutzer angezeigt werden.

Ein Trojaner zwischen JVM und dem Betriebssystem könnte auch Lesezugriffe auf Dateien protokollieren und somit unter Umständen auf die Klartextdatei schließen, die verschlüsselt wird. Ein verschlüsselter Nachrichtenaustausch kann die Vertraulichkeit nicht mehr gewährleisten, wenn der Trojaner neben der verschlüsselten Datei die zugehörige Klartextdatei an eine andere Stelle sendet.

4. Integritätsbewahrende Softwareentwicklung in Java

In diesem Kapital wird zunächst davon ausgegangen, daß Betriebssystemkomponenten und Komponenten, die zur JVM gehören, nicht durch andere Programme, z.B. Trojanische Pferde, ausgetauscht werden können. Das bedeutet, zum Betriebssystem mitgelieferte Programme und die JVM selbst werden als vertrauenswürdig betrachtet.

4.1. Bewahrung der Schnittstellensemantik

4.1.1. Grundmodell einer Anwendung

Es wird nun ein Modell eines Java-Programms erstellt. Dabei symbolisiert eine Klasse `User` das Hauptprogramm. Ein `User`-Objekt benötigt direkt zwei weitere Klassen, `B` und `C`, um seine Funktion auszuführen. `B` stellt eine Methode zum Berechnen eines Wertes bereit und `C` hat nur eine Methode, um „Hallo“ auf den Bildschirm zu schreiben. Die Klassen `B` und `C` stellen eine Verallgemeinerung eines aus mehreren Komponenten bestehenden Programms mit unterschiedlich wichtigen Funktionen dar. Die Klasse `B` könnte z.B. für eine Verschlüsselungsroutine stehen und `C` für eine grafische Fortschrittsanzeige. Die Abbildung 4.1 zeigt ein Klassendiagramm des Modells.

Ein `B`-Objekt benötige zur Berechnung des Wertes noch einen geheimen Wert aus einem Objekt der Klasse `A`. Der geheime Wert könnte der private Schlüssel eines Benutzers sein. Die Intention des Programms sei nun, daß nur die Klasse `B` den geheimen Wert von Klasse `A` lesen darf. Dies ließe sich zwar dadurch realisieren, daß man den geheimen Wert direkt in `B` speichert, aber die Klassen `A`, `B`, `C` abstrahieren hier die verschiedenen Komponenten eines Programms. In einer realen Anwendung ist es nicht immer möglich alle Informationen nur dort zu speichern, wo sie benötigt werden. Komponenten können wiederum aus Teilkomponenten bestehen, und Daten werden oft mehrmals an verschiedenen Stellen benötigt. Den genauen

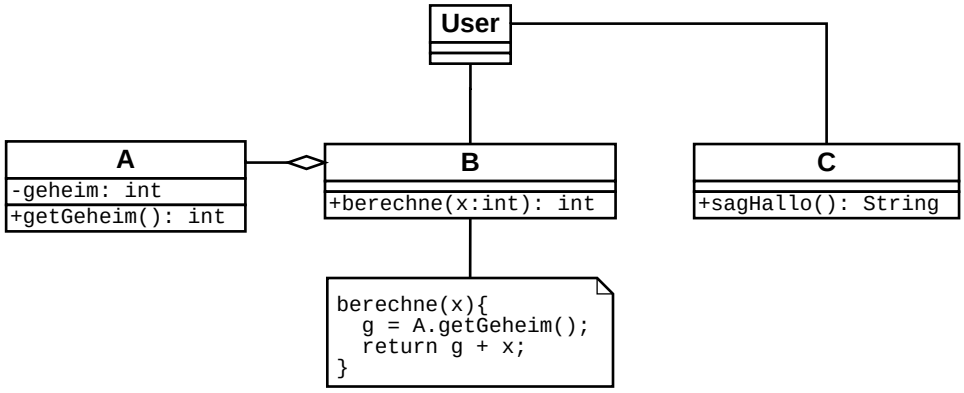

Abbildung 4.1.: Klassendiagramm einer Java-Anwendung

gedachten Programmablauf zeigt die Abbildung 4.2.

Die Frage ist, wie kann sichergestellt werden, daß nur Objekte der Klasse B den geheimen Wert von A abfragen und keine anderen Klassen oder gar externe Programme. Ebenso muß untersucht werden, was passiert, wenn die Klasse A durch ein Trojanisches Pferd ausgetauscht wird, welches die geheime Information einfach weiterverbreiten könnte. Da das Geheimnis der Klasse A durchaus einen kryptographischen privaten Schlüssel symbolisieren kann, gilt es, dies zu verhindern. Allgemein gilt es zu untersuchen, wie die Komponenten A und B sich quasi gegenseitig identifizieren können.

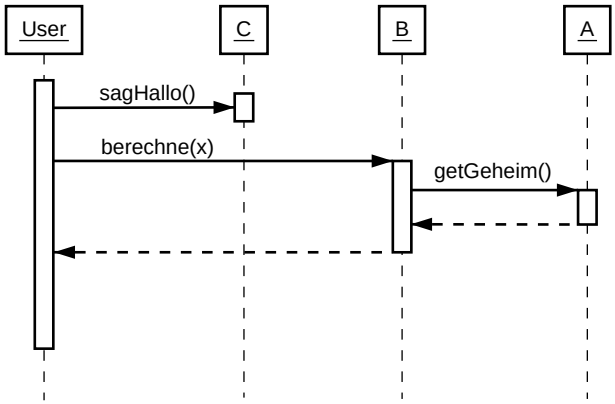

Abbildung 4.2.: Sequenzdiagramm der Java-Anwendung

4.1.2. Schutz durch JVM-Instanzen

Es sei angenommen, daß das Trojanische Pferd ein externes Programm ist, das nur die normale API der JVM und der Java-Systemklassen benutzt. Sollte der Trojaner eigene Instanzen der Klasse A erzeugen, erhalten diese die Initialwerte. Falls der geheime Wert in jedem Objekt

der Klasse `A` individuell ist, da er erst zur Laufzeit generiert wird, kann der Trojaner keinen Zugriff auf den geheimen Wert im laufenden `User`-Programm nehmen. Selbst wenn `A` eine *Singleton*-Klasse[1] ist, d.h. es gibt nur ein einziges Objekt dieser Klasse, ergibt sich die gleiche Situation für den Trojaner. Dies liegt daran, daß jedes Java-Programm eine eigene Instanz der JVM bekommt. Und eine Kommunikation zwischen verschiedenen JVMs ist standardmäßig nicht möglich (siehe [Sun02c, "System Goals"]).

Eine direkte Objekt-zu-Objekt-Kommunikation zwischen zwei verschiedenen JVM-Instanzen ist nur möglich bei der Verwendung von CORBA oder RMI. Bei CORBA meldet sich ein Objekt bei dem Object Request Broker (ORB) an und gibt eigene Methoden zum Aufruf frei. RMI benutzt dagegen Objekt-Kopien, die übertragen werden. Ein solches Objekt muß aber das Interface `java.rmi.Remote` implementieren und markiert somit alle seine Methoden aufrufbar von jeder JVM aus. Die sogenannten „Remote Objects“ müssen sich zudem in der „RMI Remote Object Registry“ registrieren, bevor ihre Methoden tatsächlich von außerhalb aufrufbar sind.

In beiden Fällen, sowohl bei CORBA als auch bei RMI, muß also explizit eine Art Anmeldung erfolgen, die den Aufruf von Methoden über JVM-Grenzen hinweg erlaubt. Hier geht es allerdings darum, daß dies verhindert wird. Durch Unterlassung dieser “Anmeldung”, d.h. durch Nichtbenutzung der CORBA- oder RMI-Technologie für die betreffenden Klassen, kann selbiges erreicht werden. Ein direkter Zugriff auf Objekte einer anderen JVM ist somit für Java-Programme nicht möglich[2]. Dies gilt allerdings nur, solange keine verteilte Architektur verwendet wird, die die Verwendung von CORBA oder RMI nötig macht.

4.1.3. Schutz durch Permissions

Sei nun angenommen, daß ein Trojaner sich im gleichen Adreßraum befindet wie das `User`-Programm, d.h. er läuft innerhalb derselben JVM und wurde von demselben Klassenlader geladen[3]. Im Versuch wurde dies dadurch realisiert, daß die Java-Bytecode-Datei einer Klasse durch eine Trojaner-Klasse gleichen Namens ausgetauscht wurde. Es wird zunächst ein Austausch der Klasse `C` im Grundmodell betrachtet.

[1] siehe “Singleton-Pattern” [GHJ95, S. 127 ff.]

[2] und auch für andere, native Programme (z.B. in C oder Assembler geschrieben), sofern sie nur die intendierte Semantik der Java- und JVM-API benutzen

[3] Dies kann z.B. dadurch geschehen, daß ein externes Trojaner-Programm zuvor Komponenten des `User`-Programms durch Trojaner-Komponenten ausgetauscht hat.

Austausch der Klasse C durch einen Trojaner

Dies bedeutet, daß eine Klasse ausgetauscht wird, die eigentlich keinen Zugriff auf die geheimen Daten nehmen darf. Die Trojaner-Klasse simuliert dabei das Verhalten der Klasse `C`, d.h. sie gibt "Hallo" aus, verfügt aber noch über ein zusätzliches Verhalten in Form einer privaten Methode, die auch bei der "Hallo"-Ausgabe ausgeführt wird. Diese private Methode stellt den Angriff dar.

Allgemein ist es nun möglich, daß der Trojaner direkt Methoden von anderen Objekten aufrufen kann, wenn er Zugriff auf die Objektreferenzen hat. Dies kann z.B. dadurch geschehen, daß Klassenmethoden solche Referenzen zurückgeben (wie das bei der *Singleton*-Klasse der Fall ist, die die Referenz auf das einzige Objekt der Klasse liefert). Der Trojaner kann auch die Reflection-API benutzen, um an Klassenobjekte innerhalb der JVM zu gelangen. Folgende Aufrufsequenz im Trojaner liefert den Klassenlader des `User`-Programms:

```
// Klassenlader des Trojaners:
ClassLoader cl = this.getClass().getClassLoader();
// Klassenlader der Applikation:
ClassLoader scl = cl.getSystemClassLoader();
```

Der nächste Schritt wäre, über den Systemklassenlader schon geladene Klassenobjekte finden zu lassen. Dazu gibt es die Methoden `findClass()` und `findLoadedClass()`. Beide sind allerdings `protected`, und somit nicht vom Trojaner aufrufbar. Es gibt allerdings die aufrufbare Methode `loadClass()`, die als Argument den zu ladenden Klassennamen verlangt. Wenn die benannte Klasse schon zuvor geladen wurde (von dem Klassenlader selbst oder von einem per Delegation übergeordneten Klassenlader), dann wird das vorhandene Klassenobjekt zurückgegeben. Ansonsten wird die Klasse neu geladen. Intern werden dabei die Methoden `findClass()` und `findLoadedClass()` benutzt. Somit ist es einem Trojaner möglich, an Klassenobjekte zu gelangen, deren Klassennamen er kennt.

Mit Hilfe dieses Klassenobjekts lassen sich neue Instanzen der betreffenden Klasse erzeugen. Dies stellt insofern kein Problem dar, da hier wieder die Initialwerte in den Objekten gespeichert sind. Allerdings ist es auch möglich, mittels `getMethod().invoke()` Klassenmethoden aufzurufen. Bei einer *Singleton*-Klasse kann der Trojaner eine Referenz auf ein schon geladenes Objekt erhalten.

Im Versuch gelang es dem Trojaner auf diese Weise den geheimen Wert auszulesen, da die Methode `getGeheim()` der Klasse `A` zwar nur für `B` „gedacht" war, aber als `public` deklariert wurde. Die Klasse `A` war als *Singleton*-Klasse implementiert. Der Angriff des Trojaners bestand neben der obigen Verwendung der Reflection-API in den folgenden Programmzeilen[4]:

[4] *Exception*-Behandlungen sind der Einfachheit halber nicht aufgeführt.

```
Class ca = scl.loadClass(”A”);
A objA = (A) ca.getMethod(”getInstance”).invoke();
objA.getGeheim();
```

Es zeigt sich, wie einfach es im Grunde genommen ist, Werte von Objekten innerhalb der JVM zur Laufzeit auszulesen, die eigentlich geheim bleiben sollten. Allerdings funktioniert dies nur bei *Singleton*-Klassen oder allgemein bei Klassen, die über Klassenmethoden Objektreferenzen zurückgeben. Ansonsten können trotz Reflection-API vorhandene Objektinstanzen nicht direkt erhalten werden. Nur Klassenmethoden und Klassenvariablen sind direkt zugänglich, wenn man keine Objektreferenzen hat! Es bleibt also für jede Anwendung im einzelnen zu untersuchen, ob nicht durch geschickte Aufrufsequenzen der API des eigenen Programms Referenzen auf Objekte zu erhalten sind.

Gegenmaßnahme Da die genaue Analyse des Quellcodes nach solchen Aufrufsequenzen im allgemeinen zu aufwendig sein dürfte, bietet sich ein Schutzmechanismus an, den Java selbst bietet: *Permissions*.

Man erzeugt eine eigene `Permission`-Klasse für den Zugriff auf die Methode `getGeheim()` der Klasse `A`. Dazu bieten sich *benannte Permissions* an, d.h. man leitet die eigene *Permission*-Klasse von `java.security.BasicPermission` ab. Dann definiert man eine benannte *Permission*, z.B. „getGeheimnis“, überprüft sie in der Methode `A.getGeheim()`, ob sie gesetzt ist und ordnet sie all den Klassen zu, die auf diese Methode zugreifen dürfen sollen, hier die Klasse `B`. Die Methode der Klasse `A` muß neben der einfachen Rückgabe des geheimen Wertes nun noch zusätzliche Überprüfungen durchführen:

```
public int getGeheim() {
  try {
     AccessController.checkPermission(
         new MyPermission(”getGeheimnis”));
  }
  catch (AccessControlException e) {
     // Ausgabe: Zugriff nicht erlaubt!!
     System.exit(1);
  }
  return geheim;
}
```

Zu beachten sind hier zweierlei Dinge. Zum einen sollte der `AccessController` verwendet werden statt des `SecurityManager`, da der `SecurityManager` nicht immer vorhanden

ist. Zum anderen sollte direkt hier die Exception aufgefangen und behandelt werden, falls der Aufrufer dieser Methode nicht die Erlaubnis zum Aufruf hat. Es wäre sonst möglich, daß ein Trojaner versucht, die Methode aufzurufen und alle anfallenden Exceptions einfach „stillschweigend“ auffängt. Der Trojaner kann zwar keinen Zugriff erlangen, aber der Benutzer bemerkt den unerlaubten Zugriffsversuch nicht und ist sich der Anwesenheit einer Trojaner-Komponente nicht bewußt.

Die Zuordnung von solchen Zugriffsrechten wird in einer *Policy*-Datei definiert. In dem hier verwendeten Modell wird die Methode `A.getGeheim()` von einem Objekt der Klasse `B` aufgerufen, das wiederum von einem `User`-Objekt dazu veranlaßt wurde. Aufgrund des Zugriffskontrollalgorithmus benötigen die Klassen `User`, `B` und `A` das Zugriffsrecht. Da man in der *Policy*-Datei *Permissions* nur ganzen Verzeichnissen oder einzelnen JAR-Dateien zuordnen kann, wäre es eine Möglichkeit, diese drei Klassen in einer JAR-Datei zusammenzufassen und ihr das entsprechende Recht zuzuordnen. Falls aber einer der Klassen noch weitere Rechte gewährt werden sollen, den anderen aber nicht, muß man für jede Klasse ein eigenes JAR-File erzeugen, um die *Permissions* genau zuordnen zu können. Im Beispiel würde die *Policy*-Datei `my.policy` folgende Einträge enthalten:

```
grant codeBase ”file:A.jar” {
   permission MyPermission ”getGeheimnis”;
};
grant codeBase ”file:B.jar” {
   permission MyPermission ”getGeheimnis”;
};
grant codeBase ”file:User.jar” {
   permission MyPermission ”getGeheimnis”;
};
```

Die *Policy*-Datei würde beim Programmstart mittels `java -Djava.security.policy=my.policy` der JVM übergeben werden. Da dies eine einfache Textdatei ist, muß zudem vom Benutzer sichergestellt werden, daß sie nicht (z.B. von einem Trojaner) modifiziert werden kann[5].

Für den Fall, daß die Aufrufkette `User-B-A` durch Einführung zusätzlicher Komponenten größer wird, erhöht sich das Risiko, daß sich ein Trojaner in diese Aufrufkette einfügt. Es ist also unter Umständen nicht wünschenswert, allen Klassen der Aufrufkette das Zugriffsrecht zu

[5]Dies kann dadurch realisiert werden, daß die *Policy*-Datei bei der Programminstallation auf dem Rechner des Benutzers in einen schreibgeschützten Bereich gespeichert wird, z.B. eine *read-only* Partition oder eine schreibgeschützte Diskette o.ä.

gewähren. In dem Fall muß man die `doPrivileged()`-Methode des `AccessController` benutzen. Der Aufruf der Methode `A.getGeheim()` in der Klasse `B` findet in einem `doPrivileged()`-Block statt. Lediglich `B` und `A` selbst benötigen dann die entsprechende *Permission*.

Befindet sich der Trojaner in der Klasse `C` und versucht er, die Methode `A.getGeheim()` aufzurufen, wirft der `AccessController` eine *Exception*, da die Klasse `C` nicht die notwendige *Permission* besitzt.

4.2. Bewahrung der Integrität der Komponenten

4.2.1. Schutz durch signierte JAR-Dateien

Im vorigen Abschnitt wurde gezeigt, wie man Java *Permissions* einsetzen kann, um die Schnittstellensemantik zu bewahren, obwohl ein Trojaner im gleichen Adreßraum der Anwendung ist. Wenn allerdings eine Komponente, die die entsprechende *Permission* besitzt, durch einen Trojaner ausgetauscht wird, dann genügt die Zugriffskontrolle der JVM nicht mehr, um die Integrität der Anwendung zu bewahren.

Austausch der Klasse A durch einen Trojaner

Es sei angenommen, daß ein Trojaner die Klasse `A` austauscht, die die geheimen Daten enthält bzw. von einer externen Quelle bekommt (z.B. von einer Datenbank). Es handelt sich folglich um eine Komponente, die die Zugriffsrechte auf die Daten besitzt. Der Trojaner könnte die Informationen an den Angreifer weiterleiten.

Im Beispiel wird bei der Ausführung von `getGeheim()` der manipulierten Klasse `A` noch eine private Methode des Trojaners aufgerufen, die das Geheimnis auf der Konsole ausgibt. Dadurch, daß die Klasse `A` die *Permission* „getGeheimnis“ selbst haben muß, um die Methode `getGeheim()` schließlich bei einem Aufruf ausführen zu können, besitzt der Trojaner ebenfalls die *Permission*, und der Angriff gelingt.

Gegenmaßnahme Der *Permission*-Mechanismus von Java allein reicht folglich nicht, um sich davor zu schützen, daß Komponenten des Programms durch modifizierte Versionen ausgetauscht werden. Java bietet aber einen Mechanismus zum Signieren (und Verifizieren von Signaturen) von JAR-Dateien.

In der *Policy*-Datei können *Permissions* nicht nur an Verzeichnisse oder JAR-Datei mittels der Angabe von `codeBase "..."` vergeben werden, sondern auch über das Schlüsselwort

`signedBy "..."` an signierten Code. Dazu müssen die JAR-Dateien mit dem JDK-Tool `jarsigner` entsprechend signiert sein. Dieser benutzt dabei Schlüssel aus einer bestimmten Datei (*Keystore*), die mit dem `keytool` aus dem JDK zuvor generiert wurden. Zum Signieren wird standardmäßig ein 1024 Bit großer Schlüssel für das Verfahren DSA-SHA1[6] erzeugt.

Die Idee ist folgende: Der Entwickler des Programms erzeugt für jede Komponente eine JAR-Datei, um *Permissions* genau zuordnen zu können. Dann signiert er alle diese JAR-Dateien mit seinem privaten Signierschlüssel. Der öffentliche Signierschlüssel wird in eine spezielle *Keystore*-Datei exportiert. In der *Policy*-Datei werden die *Permissions* entsprechend den Anforderungen der Anwendung mit der `codeBase`-Angabe definiert. Zusätzlich wird aber auch bei jeder Komponente noch die `signedBy`-Angabe gemacht, wobei der Aliasname anzugeben ist, unter dem der öffentliche Schlüssel in der speziellen *Keystore*-Datei gespeichert wurde. In der *Policy*-Datei muß zudem noch die Angabe gemacht werden, welche *Keystore*-Datei zu benutzen ist. Das fertige Programm wird nun distributiert, indem man die signierten JAR-Dateien, die *Keystore*-Datei und die *Policy*-Datei weitergibt.

Im Beispiel sieht die *Policy*-Datei folgendermaßen aus:

```
keystore "userkeystore";
grant signedBy "Marcel" codeBase "file:sA.jar" {
   permission MyPermission "getGeheimnis", signedBy "Mar-
cel";
};
grant signedBy "Marcel", codeBase "file:sB.jar" {
   permission MyPermission "getGeheimnis", signedBy "Mar-
cel";
};
```

Dabei befindet sich die Klasse `A` in `sA.jar` und `B` in `sB.jar`. Der Aliasname für den öffentlichen Schlüssel des Signierers lautet „Marcel“ und ist in der Datei `userkeystore` gespeichert.

Da die *Permissions* in der *Policy*-Datei vergeben werden und dort auch die *Keystore*-Datei bestimmt wird, müssen sowohl die *Policy*- als auch die *Keystore*-Datei in einem gesicherten Bereich gespeichert werden, damit sie nicht von einem Trojaner verändert werden können. Beim Programmstart sind dann der JVM die einzelnen JAR-Dateien und die *Policy*-Datei als Parameter zu übergeben.

Wenn die Klasse A durch einen Trojaner ausgetauscht werden soll, dann muß dazu die zugehörige JAR-Datei ausgetauscht werden. Da man davon ausgehen kann, daß der Entwickler

[6] Von den zu signierenden Daten wird ein Hashwert mittels SHA1 berechnet. Dieser Hashwert wird mit dem DSA-Verfahren signiert.

des Trojaners nicht den privaten Signierschlüssel des eigentlichen Programmentwicklers kennt, hat die ausgetauschte JAR-Datei zwangsläufig eine falsche Signatur, was von der JVM erkannt wird. Somit würde ein Angriffsversuch erfolgreich verhindert und zugleich aufgedeckt werden. Dies konnte im Versuch nachgewiesen werden.

Austausch der Permission-Klasse

Ein weiteres Angriffsszenario ist vorstellbar: Der Trojaner tauscht die *Permission*-Klasse aus. Über die `implies()`-Methode der *Permission*-Klasse könnten falsche Implikationen von Rechten zurückgegeben werden. Damit wäre die vom Entwickler intendierte Zuordnung von Zugriffsrechten veränderbar. Ein Trojaner könnte sich möglicherweise weitere Rechte verschaffen.

Diese Art von Angriff entspricht im Grunde aber dem vorherigen Fall, nämlich dem Austausch einer Komponente, die die entsprechenden Rechte schon besitzt. Da auch *Permission*-Klassen innerhalb von JAR-Dateien signiert werden können und bei jeder `permission`-Angabe in der *Policy*-Datei auch eine `signedBy`-Angabe für die *Permission*-Klasse möglich ist, greift hier die gleiche Schutzmaßnahme.

4.2.2. Dynamische Abwehrmaßnahmen?

Mittels *Permissions* kann geregelt werden, welche Zugriffe innerhalb einer Anwendung erlaubt bzw. nicht erlaubt sind. Signierte JAR-Dateien geben darüberhinaus eine gewisse Sicherheit, daß die Bytecode-Dateien nicht manipuliert oder ausgetauscht wurden. Dies ist allerdings eine statische Überprüfung der Klassen. Es stellt sich die Frage, ob nicht auch *dynamisch* geprüft werden kann, daß zwei Objekte, die miteinander kommunizieren, auch tatsächlich diejenigen sind, die sie vorgeben zu sein. Mit anderen Worten: Besteht die Möglichkeit, daß ein Objekt oder eine Klasse zur Laufzeit ausgetauscht wird, nachdem die Klasse (des Objekts) bereits geladen ist?

Java bietet zwar mit `SignedObject` die Möglichkeit, die Integrität des Zustands eines Objektes zu schützen. Dabei werden jedoch nur die Attribute des Objektes signiert, nicht der Code der Methodenimplementation. Es kann dadurch also nicht erkannt werden, ob sich das Objekt anders verhält.

Eine Idee wäre, von einem Objekt jeder Klasse einen sicheren Hashwert, z.B. mittels SHA-1, zu bilden und diesen Hashwert in andere Objekte fest einzucodieren, damit sie sozusagen ihr Gegenüber identifizieren können. Die Frage ist, wie man an geeignete Daten herankommt, die man zur Hashwertbildung heranziehen könnte, da die JVM es nicht ermöglicht, direkten Zugriff auf den Speicher und somit auf Objektcode zu nehmen.

Nichteignung des Serializable-Interfaces

Java bietet das Interface `Serializable` an, um Objekte in einen *Stream* schreiben und von dort wieder lesen zu können. Man könnte jetzt ein solches Objekt in einen `ByteArrayOutputStream` schreiben und die so erzeugte Byte-Folge als Eingabe für SHA-1 benutzen. In [Sun02a] findet sich der Hinweis, daß die Klasse, die Klassensignatur und Werte von allen nicht-`transient` und nicht-`static` Feldern der Klasse und allen Oberklassen abgespeichert werden. Dabei ergeben sich einige Probleme. Zum einen können sich Werte von Objektfeldern während der Laufzeit ändern. Dies würde es schwierig machen, einen geeigneten festen Hashwert zur Identifikation zu finden.

Zum anderen gibt es aber noch ein größeres Problem, daß diese Verfahrensweise als Schutz vor Trojanischen Pferden disqualifiziert. Wenn man eine private Methode bei einer Klasse hinzufügt und diese Methode zusätzlich in einer anderen bestehenden Methode der Klasse aufruft, dann ändert sich die serialisierte Byte-Folge nicht. Die Modifikation kann also nicht erkannt werden (siehe Abbildung 4.3).

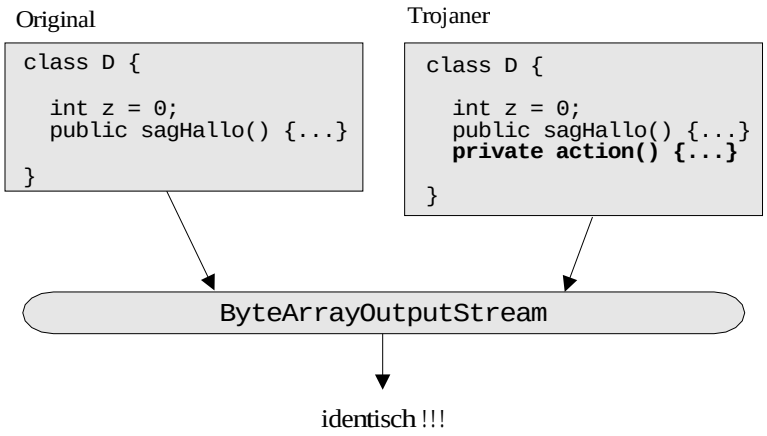

Abbildung 4.3.: Serialisierung manipulierter Klassen

Nichteignung der Reflection-API

Zu jedem Objekt gibt es die Methode `getClass()`, die das zugehörige Klassenobjekt liefert. Dies ist sozusagen der Einsprungspunkt der Reflection-API. Mit Hilfe der Klasse `java.lang.Class` und weiterer Klassen aus `java.lang.reflect` können zur Laufzeit Informationen über das fragliche Objekt gewonnen werden. So kann man abfragen, welche Methoden oder Felder es besitzt, welche *Interfaces* es implementiert oder zu welchem *Package* es gehört.

Die Frage ist, was kann ein Trojaner alles verändern und welche Möglichkeiten der Reflection-API gibt es dagegen. Zunächst einmal könnten Methoden hinzugefügt oder gelöscht, bestehende Methodenköpfe oder Methoden-*Modifier* verändert werden. Dies alles wäre über die Methode `getDeclaredMethods()` erkennbar, die Informationen über jede deklarierte Methode eines Objekts bzw. einer Klasse liefert. Eine Liste der verfügbaren Konstruktoren bietet `getDeclaredConstrutors()`, verfügbare Objektfelder samt Feld-*Modifier* zeigt `getDeclaredFields()`, und deklarierte innere Klassen gibt `getDeclaredClasses()` an. Weitere Informationen über die Klasse eines Objekts sind mit den Methoden `getName()`, `getPackage()`, `getModifiers()` und `getInterfaces()` erhältlich.

Modifikationen einer Klasse sind in dem Sinne erkennbar, daß man von den jeweils zurückgelieferten Werten der Relfection-API eine Repräsentation als `String` erhalten kann. Diese Strings könnte man konkatenieren und darüber einen SHA-1-Wert bilden. Das Problem ist aber, daß es keine Methoden in der Reflection-API gibt, die Informationen über die **Implementation** einer Methode oder eines Konstruktors liefern. Denn das, was man als String erhält, ist nur die Deklaration des Methodenkopfes mit zugehörigen *Modifier*, Parametern, Rückgabetypen und ggf. *Exceptions*. Andere Informationen sind nicht erhältlich. Ein Trojaner könnte exakt die gleiche Klasse nachbilden und müßte nur sein zusätzliches Verhalten innerhalb einer bestehenden Methode implementieren.

Nichteignung von Challenge-Response-Protokollen

Eine andere Idee wäre, Objekte und Objektreferenzen als Knoten und Verbindungen in einer Art „Mikronetzwerk“ zu interpretieren. In Netzwerken gibt es Challenge-Response-Protokolle, die gewährleisten, daß sich zwei Knoten gegenseitig identifizieren können, bevor sie miteinander vertrauliche Daten austauschen oder Operationen durchführen [MOV97, Kap.10].

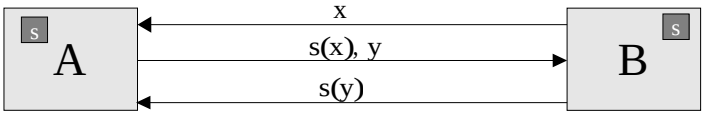

Abbildung 4.4.: Challenge-Response-Protokoll

Ein Challenge-Response-Protokoll läuft im Prinzip folgendermaßen ab (siehe Abbildung 4.4): Zwei Knoten A und B wollen sich gegenseitig identifizieren. B schickt zunächst an A einen zufälligen Wert x. A berechnet mit Hilfe der ihm bekannten Funktion s den Wert $s(x)$ und schickt $(s(x), y)$ an B, wobei y ein von A zufällig gewählter Wert ist. B berechnet dann mit der ihm ebenfalls bekannten Funktion s den Wert $s(y, x)$ und schickt diesen an A. Wenn

nur A und B die Funktion s kennen, dann haben sie sich auf diese Weise mit einer gewissen Wahrscheinlichkeit gegenseitig identifiziert, da die Werte für x und y zufällig gewählt wurden.

Ein derartiges Protokoll ist zur Identifikation von Objekten allerdings nicht geeignet. Angenommen ein Trojaner tauscht die Klasse A aus, und der Quellcode von A ist bekannt, dann ist zwangsläufig auch die Funktion s bekannt, und der Trojaner kann $s(x)$ berechnen. Wenn B vom Trojaner $s(x)$ und y bekommt, dann hält B den Trojaner für A, und die Identifikation ist unterlaufen.

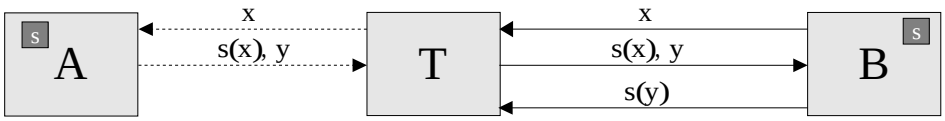

Abbildung 4.5.: Trojaner beim Challenge-Response-Protokoll

Wenn der Quellcode von A nicht bekannt ist, dann kann evtl. eine Analyse des Bytecodes der Klasse A (*Reverse Engineering*) zu dem Teil im Bytecode führen, der s implementiert. Ein Trojaner könnte dann diesen Bytecode-Teil ggf. in seinen Code einbauen. Eine einfachere Möglichkeit wäre allerdings, daß der Trojaner eine Kopie der Klasse A behält. Wenn die Anfrage von B an den Trojaner kommt, sich zu identifizieren, dann leitet dieser die Anfrage an die originale Klasse A weiter. Dessen Antwort schickt er zurück an B (siehe Abbildung 4.5). Dies entspricht quasi einem *Man-in-the-Middle-Attack*. Der Fall, daß der Trojaner B austauschen will, läuft analog.

Fazit

Mit den normalen Mitteln, die Java bietet, ist es nicht möglich, zur Laufzeit zu erkennen, ob sich die Implementation eines Objektes verändert hat. Man müßte mit nativ implementierten Methoden den Speicher der JVM diesbezüglich untersuchen. Das würde jedoch die Portabilität der Java-Anwendung erschweren.

Java bietet zwar keine Möglichkeit, Klassen zur Laufzeit auszutauschen, so daß sich die Implementation von Objekten auch nicht dynamisch ändern kann. Es gibt jedoch Bestrebungen, eine *Virtual Machine* mit dieser Möglichkeit zu entwickeln [MPG00]. Wenn die JVM in einer späteren Version diese Fähigkeit offiziell unterstützen sollte, dann müßten auch Möglichkeiten neu eingebracht werden, um die Veränderung der Implementation von Objekten zur Laufzeit erkennen zu können.

5. Robuste Integration in Windows 2000

Es wurde gezeigt, daß Java Schutzmöglichkeiten gegenüber direkten Angriffen auf eine Anwendung bietet. Es blieb jedoch ein Schwachpunkt: das sichere Speichern und Lesen der *Policy*- und der *Keystore*-Datei. Hierfür ist das Betriebssystem verantwortlich. Es gilt zu verhindern, daß Teile des Betriebssystems oder der JVM unbemerkt ausgetauscht oder manipuliert werden können. Java allein bietet keine Schutzmöglichkeiten vor derartigen Angriffen.

5.1. Integritätsprüfer für verwendete Module

In Windows 2000 kann der Zugriff auf Dateien über ACLs geregelt werden. Die Betriebssystem-Dateien und andere Programmdateien werden standardmäßig so geschützt, daß normale Benutzer nur lesend darauf Zugriff haben. Schreibzugriff haben nur Administratoren. Auf diese Weise können auch die *Policy*- und die *Keystore*-Datei vor Manipulationen geschützt werden.

Wenn jedoch zusätzliche Software auf dem System installiert werden soll, erfordert die Installation häufig administrative Rechte. Da auf Einzelplatzsystemen – vor allem im privaten Bereich – der Benutzer meistens gleichzeitig der Administrator ist, wird er sich zur Programminstallation als Administrator anmelden. Ein Trojaner in Form einer Installationsroutine, die vom Administrator ausgeführt wird, könnte dann andere Programm- oder Moduldateien modifizieren.

Windows File Protection schützt zwar durch Integritätsprüfungen die wichtigsten Betriebssystem- und Treiberdateien, ist jedoch nicht erweiterbar, um Manipulationen an anderen Anwendungen erkennen zu können. Zudem besteht die Gefahr, daß eine DLL gleichen Namens statt aus dem Originalverzeichnis aus einem lokalen Verzeichnis des Aufrufers geladen wird. Aus diesem Grund wird im folgenden ein selbst entwickelter Integritätsprüfer namens *Module-Stamper* vorgestellt.

Funktionsweise der Integritätsprüfung

Um die Integrität von Anwendungen erkennen zu können, muß zunächst bekannt sein, welche Komponenten zur Anwendung dazugehören. Neben ausführbaren Programmdateien (Exe-Dateien) kann die Anwendung noch verschiedene DLLs benötigen und auf bestimmte Treiber angewiesen sein.

Die Funktion `EnumDeviceDrivers()` aus der PSAPI [MS01f] listet alle aktuell geladenen Treiber auf, und mit der Funktion `GetDeviceDriverFileName()` erhält man die zugehörigen Namen der Treiberdateien (siehe auch Anhang A.1.3). Eine Liste der Module (DLLs), die eine Anwendung benötigt, erhält man von ihrem laufenden Prozeß. Die PSAPI bietet dazu die Funktion `EnumProcessModules()`, die ein Handle auf den Prozeß benötigt. Ein Prozeß-Handle kann man über die Prozeß-ID bekommen. Die genauen Dateinamen der Module liefert die Funktion `GetModuleFileNameEx()`. Der Begriff „Module“ bezeichne im folgenden sowohl DLLs als auch Treiber und ausführbare Programmdateien.

Die Integrität von Moduldateien kann dadurch überprüft werden, indem zu jeder Datei ein Hashwert über den Dateiinhalt gebildet wird und mit einem zuvor generierten Wert verglichen wird. Zur Bildung der Hashwerte wird die kryptographische Hashfunktion SHA-1 [NIS95] verwendet, so daß davon ausgegangen werden kann, daß die geringsten Änderungen an den Moduldateien eine Änderung der Hashwerte bewirken. Ein Abweichen der aktuellen Hashwerte mit den Referenzwerten wird somit als Integritätsverletzung interpretiert. Die Bildung eines Hashwertes für eine Moduldatei wird im folgenden „Stampen“ genannt, die Hashwerte selbst heißen „Stamps“.

Zu jeder zu überprüfenden Anwendung werden erstmalig bei der Installation der Anwendung die Stamps aller von ihr benötigten Module berechnet und in einer speziellen Datei, der „Stampdatei“, als Referenzwerte zusammen mit der Liste der Modulnamen gespeichert. Beim späteren Starten einer solchen Anwendung werden zuerst die benötigten Module erneut aufgelistet. Anschließend werden die Liste und die Stamps der Moduldateien mit den Referenzwerten verglichen. Dabei können folgende Situationen eintreten:

- Die Stamps einer Moduldatei stimmen überein.
- Die Stamps einer Moduldatei sind unterschiedlich.
- Eine Moduldatei fehlt in der aktuellen Liste.
- Eine Moduldatei ist neu hinzugekommen.

Wenn eine Moduldatei in der aktuellen Liste und der Referenzliste vorkommt und die Stamps übereinstimmen, bekommt der Anwender die Meldung „OK“. Stimmen sie nicht überein, wur-

de die Datei verändert, und der Anwender erhält eine Alarmmeldung. Ein Fehlen einer Moduldatei ist nicht problematisch. Bei einem neu hinzugekommenen Modul besteht jedoch die Gefahr, daß es sich um eine Trojaner-Komponente handelt. Wenn das Modul bei der Erstellung der Referenzwerte für diese Anwendung nicht geladen war, aber ein Referenz-Stamp existiert (z.B. weil es von einer anderen Anwendung benötigt wurde), dann kann überprüft werden, ob der aktuelle Stamp übereinstimmt. Ist dies nicht der Fall, wird eine Alarmmeldung gegeben. Die Integrität der betreffenden Anwendung ist nicht mehr gewährleistet.

Damit Veränderungen an Moduldateien korrekt erkannt werden können, muß die Stampdatei vor Manipulationen geschützt werden. Andernfalls könnte ein Angreifer nach einer Manipulation einer Moduldatei den zugehörigen Referenzwert in der Stampdatei durch den aktuellen Stamp ersetzen.

Implementation als Windows 2000 Dienst

Der *ModuleStamper* wurde als Windows 2000 Dienst implementiert. Dies befähigt ihn dazu, permanent im Hintergrund zu laufen und einige Privilegien des Systemkontos ausnutzen zu können. So wird der Dienst beim Systemstart automatisch gestartet und kann schon Überprüfungen durchführen, bevor sich der erste Benutzer am System anmelden kann. Beim Shutdown werden Dienste als letzte Prozesse beendet, so daß auch hier der *ModuleStamper* noch Überprüfungen durchführen kann, während alle anderen Benutzermodus-Prozesse schon beendet sind.

Um die Stampdatei zu schützen, bildet der Dienst beim Systemstart einen Stamp über die Stampdatei selbst und merkt ihn sich im Arbeitsspeicher. Vor jedem erneuten Zugriff auf die Stampdatei wird dieser Wert mit dem aktuellen verglichen. Zusätzlich bietet Windows 2000 einen *File Locking* Mechanismus: Der *ModuleStamper*-Dienst sperrt die Stampdatei, andere Prozesse können dann darauf weder lesend noch schreibend zugreifen. Da der Dienst im Systemkonto ausgeführt wird, kann der Zugriff auf die Stampdatei generell auf das Systemkonto beschränkt werden, so daß normale Benutzer überhaupt keinen Zugriff darauf haben.

Während der Dienst aktiv ist, wird seine eigene Programmdatei vom Betriebssystem für Schreibzugriffe gesperrt. Somit läßt sie sich nicht manipulieren. Um den Dienst nicht zwischenzeitlich beenden zu können, wurde er so implementiert, daß er die Dienst-Befehle *Stop* und *Pause/Continue* nicht akzeptiert, sondern nur *Run* und *Shutdown*.

Damit der Dienst beim Shutdown seine Überprüfungen zu Ende führen kann, bevor er selbst beendet wird, muß in der Registry unter *HKLM\SYSTEM\CurrentControlSet\Control* der Wert „WaitToKillServiceTimeout“ angepaßt werden. Die genaue Zeit ist vom eingesetzten Rechner abhängig, jedoch sollten 1-2 Minuten in den meisten Fällen völlig ausreichen.

In einem produktiven System kann der ModuleStamper als Autostart-Dienst mit dem Flag

SERVICE_ERROR_SEVERE installiert werden. Dies bewirkt, daß das System komplett heruntergefahren wird, wenn ein Fehler im Dienst auftritt oder er sich nicht starten läßt.

Der Dienst sollte nicht mit dem Flag SERVICE_INTERACTIVE_PROCESS installiert werden, da Benutzer den Dienst sonst beenden könnten, wenn er ein interaktives Fenster anzeigt. Statt dessen verwendet der ModuleStamper einen speziellen Befehl in der Funktion `MessageBox()`, um Meldungen anzuzeigen. Für die restliche Interaktion mit dem Benutzer, wie das Konfigurieren des Dienstes, werden spezielle Client-Programme verwendet.

Der *ModuleStamper* besteht aus drei Teilen: aus dem *ModuleStamper*-Dienst, dem Client und dem Konfigurationsprogramm (SCP). Mit dem Client können Programme gestartet werden, deren Module dann vom Dienst auf Integrität überprüft werden. Der Client kann den Dienst auch dazu veranlassen, eine Gesamtüberprüfung des Systems zu machen, d.h. alle wichtigen Betriebssystem-, DLL- und Treiberdateien werden überprüft. Mit dem SCP läßt sich der Dienst konfigurieren. Die Liste der zu überprüfenden Anwendungen kann definiert und geändert werden. Der Dienst besitzt jedoch eine unveränderbare, feste Standardliste, zu der alle wichtigen Betriebssystem-Programmdateien und alle installierten Treiber gehören.

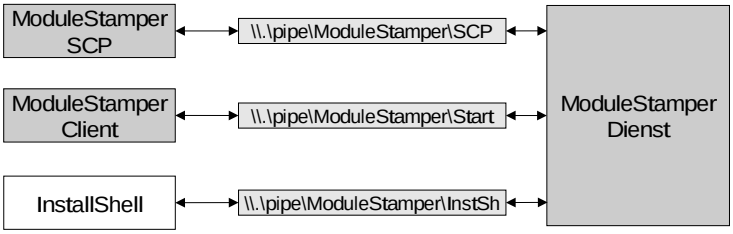

Abbildung 5.1.: Interprozeß-Kommunikation beim *ModuleStamper*

Damit der Client oder das SCP mit dem Dienst-Prozeß kommunizieren können, ist eine *Interprozeß-Kommunikation* nötig. Windows 2000 bietet dafür verschiedene Möglichkeiten. Die Interprozeß-Kommunikation im *ModuleStamper* geschieht mittels *Named Pipes*. Das sind virtuelle benannte Dateiobjekte, die vom Dienst-Prozeß angelegt werden, damit Client-Prozesse mit ihm kommunizieren können.

Es gibt drei *Named Pipes*: *„ModuleStamper\SCP“* ist für das Konfigurationsprogramm des Dienstes, *„ModuleStamper\Start“* ist für den Client, der Programme startet und über den *ModuleStamper*-Dienst die Module überprüfen läßt, und *„ModuleStamper\InstSh“* ist für spezielle Anfragen einer beschränkten Installationsshell (siehe Abschnitt 5.3.3). Da *Named Pipes* virtuelle Dateiobjekte sind, kann der Zugriff auf sie mit einer Sicherheitsbeschreibung geschützt werden. Die *Named Pipes „SCP“* und *„InstSh“* können nur vom Administratorkonto gelesen und geschrieben werden. Dies verhindert, daß normale Benutzer die Konfiguration des Dienstes verändern können.

Um eine Anwendung über den *ModuleStamper* zu starten, damit der Dienst die von der Anwendung benötigten Module und alle aktuell geladenen Treiber überprüfen kann, ruft man den Client wie folgt auf:

```
ModuleStamper -start X:\Ver\zeich\nix\Programm.exe
```

Der Client startet daraufhin das Programm und übermittelt über die *Named Pipe* dem Dienst die Prozeß-ID. Der Dienst überprüft alle Modul- und Treiberdateien, die geladen sind, und gibt seinen Bericht zurück an den Client. Falls eine Integritätsverletzung entdeckt wurde, zeigt der Dienst zusätzlich eine Warnmeldung an und macht einen entsprechenden Eintrag im *Event Log*.

Alle Konfigurationsdaten des *ModuleStampers* werden zusammen mit den Stamp-Referenzwerten in der Stampdatei gespeichert. Sie stellt somit die zentrale Datenbank des *ModuleStampers* dar. In der *Registry* werden ansonsten keine Daten gespeichert, bis auf die Einträge, die für das Funktionieren des Dienstes nötig sind.

Sowohl der *ModuleStamper*-Dienst als auch das SCP und der Client werden innerhalb einer ausführbaren Programmdatei *ModuleStamper.exe* implementiert. Durch die Sperrung der Datei vom Betriebssystem können das SCP und der Client ebenfalls nicht manipuliert werden, solange der Dienst aktiv ist.

Beim Start des Dienstes und beim Shutdown macht der *ModuleStamper* automatisch eine Überprüfung aller Dateien in der Standardliste. Zusätzlich überprüft er die *Registry*, ob neue Treiber oder Dienste installiert wurden und ob sein eigener Eintrag in der *Registry* manipuliert wurde. Wenn er beispielsweise von einem Trojaner als „deaktiviert“ markiert wurde, würde er beim nächsten Systemstart nicht mehr gestartet werden. In diesem Fall trägt er sich selbst wieder ein und gibt eine Warnmeldung aus. Der *ModuleStamper*-Dienst läßt sich nur über sein eigenes Konfigurationsprogramm deinstallieren.

5.2. Manipulationsfestigkeit von Diensten

5.2.1. Angriffe durch Treiber

Fälschung von Daten

Jedes Hardware-Gerät benötigt einen Treiber, der es ansteuert. Wie in Abschnitt 2.4.6 beschrieben, ist dabei jedem Gerät eine Treiberhierarchie zugeordnet, an die die I/O-Anforderungen für das Gerät weitergeleitet werden. Auf jeder Stufe der Hierarchie kann die Anfrage vom entsprechenden Treiber selbst vollständig beantwortet werden, ohne daß sie an die unteren Stufen weitergereicht werden muß.

Ein Dateisystemtreiber oder ein darüber liegender Filtertreiber könnte die Anfrage für eine bestimmte Datei, wie die Stampdatei, abfangen und statt dessen falsche Daten an den Aufrufer schicken. Ein Trojaner in Form eines solchen Treibers könnte das Konzept des *ModuleStampers* aushebeln. Die Modifikation von Modulen der JVM oder des Betriebssystems ist durch den *ModuleStamper* nicht mehr erkennbar, wenn der Trojaner-Treiber die neuen Hashwerte der modifizierten Module liefert anstatt der Werte, die in der Stampdatei stehen.

Beispielsweise seien auf einem System die beiden Dateien `ntfs.sys` und `jvm.dll` in der Stampdatei mit den Hashwerten „111111“ bzw. „222222“ vom *ModuleStamper* eingetragen worden[1]. Eine manipulierte Variante von `ntfs.sys` habe nach gleichem Verfahren den Hashwert „1A1A1A“ und eine Trojaner-Variante von `jvm.dll` den Wert „F2F2F2“. Normalerweise würde der *ModuleStamper* bei einer Überprüfung erkennen, wenn die beiden Dateien durch ihre manipulierten Varianten ausgetauscht würden. Bei der Überprüfung der modifizierten Datei `jvm.dll` würde der Hashalgorithmus den Wert „F2F2F2“ liefern, während in der Stampdatei der Wert „222222“ für diese Datei steht (siehe Abbildung 5.2). Der *ModuleStamper* würde folglich Alarm schlagen.

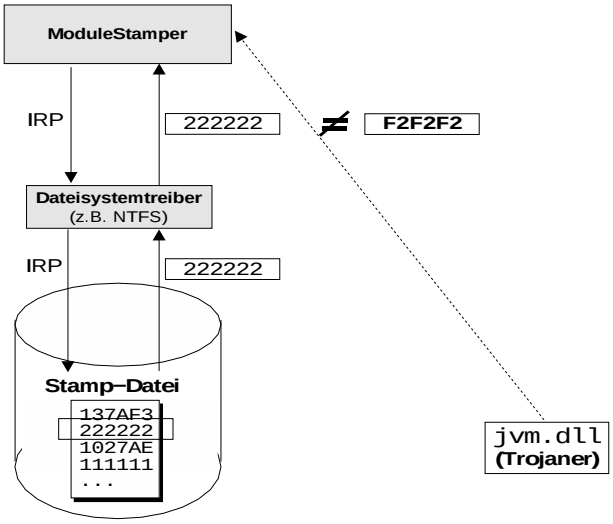

Abbildung 5.2.: Normale Funktionsweise des *ModuleStampers*

Wenn aber die manipulierte Variante des Dateisystemtreibers `ntfs.sys` geladen ist, und dieser einen Aufruf der Stampdatei abfängt und geschickt gefälschte Daten zurückliefert, kann der *ModuleStamper* die Veränderung nicht mehr erkennen. Dies ist für den Treiber nicht sonderlich schwierig, wenn er die Vorgehensweise des *ModuleStampers* und dessen verwendete Algorithmen kennt[2]. Im Beispiel berechnet der *ModuleStamper* für die modifizierte Datei `jvm.dll`

[1] Diese Werte stellen hier natürlich nur eine Vereinfachung zur Illustration dar. In der implementierten Fassung werden SHA-1 Hashwerte von den Dateien gebildet.

[2] Aufgrund von *Reverse Engineering* oder *Open Source* kann man davon ausgehen, daß der Entwickler des Tro-

den Hashwert „F2F2F2“. Der manipulierte Dateisystemtreiber braucht nur bei der Anfrage nach dem Referenzwert in der Stampdatei ebenfalls diesen Wert anstelle des Originals zu liefern, und der *ModuleStamper* hält die Datei für unverändert. Abbildung 5.3 veranschaulicht dies.

Beim Austausch einer Betriebssystem-Komponente, wie des Dateisystemtreibers, muß die neue Version die gleiche Funktionalität bieten wie das Original, da sonst möglicherweise Programmfehler oder Systemabstürze auftreten können, die auf die ausgetauschte Komponente aufmerksam machen würden. Dies stellt allerdings keine allzu große Hürde dar. Schon geringe Manipulationen am Original, wie ein Umlenken eines Funktionsaufrufs auf eine neue, hinzugefügte Routine, können die zusätzliche Funktionalität eines Trojaners einbringen: Der Code der neuen Routine muß an die Originaldatei angefügt werden, und die dort eingetragenen Sprungadressen für die Funktionsaufrufe müssen geändert werden.

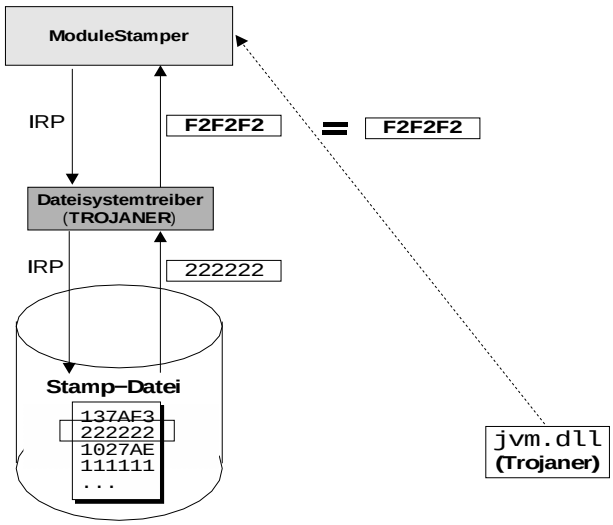

Abbildung 5.3.: *ModuleStamper* mit manipuliertem Dateisystemtreiber

Der Dateisystemtreiber muß zudem nicht selbst ausgetauscht werden. Ein zusätzlicher Filtertreiber, der oberhalb des Dateisystemtreibers operiert, kann auch Anfragen für die Stampdatei abfangen. Dieser ist aber über die *Registry* zu erkennen: Eine Liste der installierten Treiber läßt sich immer aus der *Registry* entnehmen[3], und in den *Registry*-Schlüsseln *Enum*[4] und *Class*[5] für Geräte steht, ob ein Treiber als Filtertreiber eingetragen ist. Es bleibt jedoch das Problem, daß die Bedeutung von Treibern nicht immer automatisch zu erkennen ist. Es kann schließlich durchaus gewollt sein, daß ein Filtertreiber installiert wird – z.B. für ein virtuelles Laufwerk.

Mit einer alternativen Methode kann der Austausch eines Treibers nicht nur vor dem hier

janers darüber Kenntnis besitzt und seinen Trojaner entsprechend anpassen kann.

[3] Alle installierten Treiber sind im Schlüssel *HKLM\SYSTEM\CurrentControlSet\Services* aufgelistet.

[4] *HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum*

[5] *HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Class*

vorgestellten *ModuleStamper* verborgen werden, sondern vor allen Einbruchsmeldern, die die Dateiinhalte untersuchen. Ähnlich der *Stealth-Technik* bei Computerviren[6] werden Anfragen für die Datei des Treibers umgelenkt auf die Originaldatei, die irgendwo versteckt gespeichert wird. Sie kann sich an einem anderen Platz auf dem Datenträger oder direkt in der Trojaner-Treiberdatei selbst befinden (als konstanter Datenblock). Beim Laden des Treibers während des Systemstarts wird die Trojaner-Datei gelesen, danach werden jedoch alle Leseversuche auf diese Datei vom Trojaner-Treiber abgefangen und auf die Originaldaten umgelenkt. Dies ist möglich und führt zu keinen Konflikten, da eine Treiberdatei von Windows 2000 nicht mehr benötigt wird, nachdem sie in den Systemspeicher geladen wurde. Mit dieser Technik ist es für den Trojaner-Treiber einfach, die ursprüngliche Funktionalität des ausgetauschten Treibers zu gewährleisten: Je nach Bedarf werden Anfragen an den Originalcode weitergeleitet oder von eigenen Routinen bearbeitet.

Bei der Verwendung einer derartigen „Stealth-Technik“ kann grundsätzlich *jeder* Treiber die Manipulation seiner Treiberdatei verbergen, es muß sich nicht notwendigerweise um einen Dateisystemtreiber handeln. Treiber laufen normalerweise im Kernelmodus und haben somit uneingeschränkten Zugriff auf Hardware und Systemspeicher. Selbst ein Trojaner in Form eines Joystick-Treibers könnte Anfragen für seine eigene Treiberdatei manipulieren, da sämtliche internen Datenstrukturen des Betriebssystems einem Treiber zugänglich sind. Beispielsweise könnte solch ein Joystick-Treiber durch geschickte Manipulation des I/O-Anforderungspakets für seine Treiberdatei die Anfrage auf eine andere Datei umlenken. Eine andere Möglichkeit wäre, zur Laufzeit die Treiberhierarchie für ein Gerät zu ändern, so daß Dateianfragen auch immer den Joystick-Treiber „passieren“ müssen. Dazu muß der entsprechende Geräteknoten im Gerätebaum des I/O-Managers manipuliert werden. Bei derartigen Manipulationen müssen jedoch die genauen Speicheradressen der internen Datenstrukturen herausgefunden werden, da ein Joystick-Treiber keinen offiziellen Zugriff auf Geräteknoten oder nicht für ihn bestimmte I/O-Anforderungspakete hat.

Solche eigentlich unerlaubten aber dennoch möglichen Zugriffe auf interne Datenstrukturen und deren Manipulationen können unter Umständen zu Inkonsistenzen oder Systemabstürzen führen, da z.B. die Synchronisation des Objektzugriffs nicht beachtet wird. So kann ein (inoffizieller) Schreibvorgang auf eine interne Datenstruktur durch den Trojaner mit einem offiziellen Schreibvorgang eines anderen Programms kollidieren. Windows 2000 bietet zwar mit *Verteilerobjekten* einen Synchronisationsmechanismus, er ist jedoch für die Ausführungsschicht und den Benutzermodus gedacht. Treiber im Kernelmodus können diesen Mechanismus umgehen. Die Entwicklung eines Trojaner-Treibers, der Manipulationen an internen Datenstrukturen vorneh-

[6] siehe [Spa94]

men kann, ohne Inkonsistenzen und Systemabstürze hervorzurufen, ist folglich eine ungleich schwierigerer Aufgabe als die Entwicklung eines betriebssystemkonformen Dateisystemtreibers mit zusätzlichem Verhalten.

Zu den betriebssystemkonformen Treibern, die auf die vorgestellte Weise ihr Vorhandensein gegenüber Integritätsprüfprogrammen wie den *ModuleStamper* verbergen können, gehören alle Treiber, die in der Hierarchie zur Abarbeitung von I/O-Anforderungen auf Datenträgern stehen: **Miniport-Treiber**, **Datenträger-Anschlußtreiber**, **Datenträger-Klassentreiber**, **Datenträgerverwaltungstreiber**, **Dateisystemtreiber** sowie diverse **Filtertreiber** (siehe Abb. 2.2). Dabei muß es sich um die Hierarchie desjenigen Datenträgers handeln, der die Stampdatei des Prüfprogramms oder (bei der Stealth-Technik) die Treiberdatei des Trojaners enthält. Denn nur diese Hierarchie bekommt vom I/O-Manager die Anfrage weitergereicht, wenn die entsprechende Datei gelesen werden soll.

Manipulation von Prozessen

Da ein Treiber, der im Kernelmodus läuft, uneingeschränkten Zugriff auf den Systemspeicher hat, kann er auch auf die Datenstrukturen zugreifen, die die laufenden Prozesse verwalten. Über diesen Zugriff kann er den Arbeitsspeicher bestimmter Prozesse lesen und sogar verändern. Somit sind Prozesse durch Treiber im Kernelmodus direkt manipulierbar.

Es ist folglich denkbar, daß ein Trojaner-Treiber einen bestimmten Dienst direkt angreifen kann. Er könnte dessen Programmverhalten ändern, indem er z.B. Sprungadressen oder Datenstrukturen im Prozeßspeicher des Dienstes ändert. Auf diese Weise könnte der Trojaner verhindern, daß der *ModuleStamper* Alarm schlägt, wenn ein verändertes Modul gefunden wurde, oder daß überhaupt eine Überprüfung der Module durchführt wird.

Durch den Zugriff auf den Systemspeicher könnte ein Trojaner-Treiber auch direkt die kryptographische Anwendung oder ein beliebig anderes laufendes Programm manipulieren. So könnte auch dort das Programmverhalten geändert oder interne Zustände und Daten ausgelesen werden.

Die Entwicklung eines solchen Treibers ist jedoch sehr aufwendig, da genaueste Kenntnis der Programme vorhanden sein muß, deren Prozeßspeicher manipuliert werden soll. Dennoch ist es technisch möglich und sollte folglich als potentieller Angriffsweg nicht ausgeschlossen werden.

Austausch vorhandener Treiber

Grundsätzlich kann ein Treiber im laufenden Betrieb ausgetauscht werden, wenn er sich entladen und wieder laden läßt und wenn man entsprechende Zugriffsrechte auf die Treiberdatei hat.

Ist ein Treiber schon geladen, wird seine Treiberdatei nicht mehr benötigt. Änderungen an der Datei machen sich erst bemerkbar, wenn der Treiber erneut geladen wird.

Da Treiber letztendlich vom I/O-Manager durch Geräteknoten verwaltet werden, kann ein Treiber entladen werden, indem der Geräteknoten, dem er zugeordnet ist, deaktiviert wird. Verweisen keine weiteren Geräteknoten auf den Treiber, wird er nicht mehr benötigt und vom I/O-Manager entladen. Ein erneutes Aktivieren des Geräteknotens bewirkt, daß alle zugehörigen Treiber, die nicht mehr im Speicher sind, wieder geladen werden. Ein Angreifer könnte vor dem Aktivieren des Geräteknotens die Treiberdatei austauschen, so daß sie anstelle des Originals geladen wird.

Geräteknoten lassen sich z.B. mit dem Programm *Disabler* [MS01a] für folgende Windows-Geräteklassen deaktivieren und aktivieren: PORTS (Kommunikationsschnittstellen), USB, HDC (Festplatten-Controller), NET (Netzwerkkarten), MEDIA (Soundkarten). Über den Gerätemanager von Windows 2000 lassen sich zudem auch Geräteknoten der Klassen CDROM, FLOPPYDISK, FDC (Floppylaufwerk-Controller) und MONITOR de- und reaktivieren. Die Reaktivierung erfolgt, wenn der Vorgang *„Nach geänderter Hardware suchen“* im Gerätemanager gestartet wird. Die zur Deaktivierung bzw. Reaktivierung erforderlichen Funktionen `CM_Disable_DevNode()` und `CM_Enable_DevNode()` finden sich in der *Configuration Manager* API [MS01i] und könnten von einem Angreifer verwendet werden. Anhang A.1.1 zeigt beispielhaft die Deaktivierung von Geräteknoten, um einen Filtertreiber im laufenden Betrieb auszutauschen.

Falls sich ein Treiber im laufenden Betrieb nicht entladen und wieder laden läßt, bleibt nur die Möglichkeit, die Treiberdatei auszutauschen und auf einen Neustart des Systems zu warten. Dann werden alle Treiber neu geladen, die für das System erforderlich sind. Ein Neustart läßt sich zudem programmatisch erzwingen. Ein Trojaner könnte davon Gebrauch machen, um nach einem Austausch das System sofort neu zu starten. Da nach einer Installation ein anschließender Neustart für viele Programme unter Windows typisch ist, wäre dieses Verhalten auch nicht weiter auffällig.

Die Treiberdateien, die mit Windows 2000 ausgeliefert werden, werden zwar von *Windows File Protection* (WFP) geschützt, so daß ein Austausch rückgängig gemacht und ein Eintrag im *Event Log* vorgenommen wird. Jedoch gelingt die Wiederherstellung nur, wenn eine entsprechende Kopie im *Dllcache* von WFP vorhanden ist. Ein Angreifer könnte vor dem Austausch der Treiberdatei die Kopie im *Dllcache* (und in der Datei *Drivers.cab*) löschen. WFP könnte die Datei dann nicht wiederherstellen. Es würde nur ein Eintrag im *Event Log* gemacht und der Administrator nach der Installations-CD gefragt werden.

Wenn der ausgetauschte Treiber sofort im laufenden Betrieb geladen werden kann, befindet sich der Angreifer im Kernelmodus. Von dort hat er vollen Zugriff auf den Systemspeicher und

könnte den *Event Logger*, der selbst ein Dienst ist, manipulieren. Da WFP nur bestimmte und nicht alle Treiberdateien schützt, könnte in einem zweistufigen Angriff ein unbemerkter Austausch einer geschützten Datei gelingen: Zuerst wird ein ungeschützter Treiber ausgetauscht und neu geladen. Dieser manipuliert den *Event Logger*, so daß in einem zweiten Schritt der Austausch einer geschützten Treiberdatei nicht protokolliert wird. Der zweite Schritt kann später erfolgen, wenn kein Administrator mehr angemeldet ist, so daß die Frage nach der Installations-CD nicht sofort angezeigt wird.

Bereits geladene **Dateisystemtreiber** lassen sich nicht im laufenden Betrieb austauschen. Diese Feststellung läßt sich aus den Informationen in [SR00] herleiten. Wie in Abschnitt 2.4.6 beschrieben, werden Dateisystemtreiber nicht in Geräteknoten, sondern in einer eigenen Liste vom I/O-Manager verwaltet. Sie müssen sich zudem explizit als Dateisystemtreiber beim I/O-Manager anmelden, und nur beim Laden eines Treibers wird dessen Initialisierungsroutine aufgerufen. In der Liste stehen nur bereits geladene Treiber, *„da alle Dateisystemtreiber während ihrer Initialisierung beim E/A-Manager*[7] *registriert werden“* [SR00, S. 545]. Bei einem Zugriffsversuch auf einen Datenträger *„fragt der E/A-Manager über eine Bereitstellungsanforderung jeden registrierten Dateisystemtreiber, ob der Treiber das Datenträgerformat als sein Format erkennt“* [SR00, S. 546]. Daraus folgt, daß der Dateisystemtreiber schon geladen sein muß, wenn der I/O-Manager ihn über das Datenträgerformat befragt.

Es werden nicht alle vorhandenen Dateisystemtreiber gleich zu Beginn geladen. Windows 2000 verwendet einen Ersatztreiber, den *File System Recognizer* (`Fs_rec.sys`), der alle unterstützten Dateisysteme erkennen kann. Allerdings *„wenn der E/A-Manager ihn während eines Systembereitstellungsvorgangs für einen neuen Datenträger aufruft, lädt File System Recognizer den passenden Dateisystemtreiber, falls der Startdatensatz einem Treiber zugeordnet ist, der nicht geladen ist“* [SR00, S. 547]. Der Dateisystemtreiber steht ab diesem Zeitpunkt in der Liste des I/O-Managers.

Ein Treiber wird nur entladen, wenn er nicht mehr benötigt wird. Dies ist der Fall, wenn alle Geräteknoten im Gerätebaum deaktiviert oder gelöscht wurden, die auf diesen Treiber verweisen. Da Dateisystemtreiber in einer speziellen Liste und nicht in Geräteknoten verwaltet werden, läßt sich daraus schließen, daß ein einmal geladener Dateisystemtreiber im laufenden Betrieb nicht wieder entladen wird. Dies konnte auch in der Praxis bestätigt werden (siehe Anhang A.1.2). Für den Austausch eines geladenen Dateisystemtreibers ist es jedoch erforderlich, daß er entladen und wieder geladen wird. Daraus folgt, daß sich bereits geladene Dateisystemtreiber im laufenden Betrieb nicht austauschen lassen. Die Änderungen an den Treiberdateien würden erst nach einem Neustart des Betriebssystems aktiv werden.

[7] Mit E/A-Manager ist der I/O-Manager gemeint.

Dateisystemtreiber können folglich nur ohne einen Neustart ausgetauscht werden, wenn sie noch nicht geladen sind. Beim Systemstart von Windows 2000 wird allerdings eine Überprüfung aller Datenträger gemacht, denen ein Laufwerksbuchstabe zugeordnet ist (siehe [SR00, S. 548]). Das bedeutet, daß die wichtigsten Dateisystemtreiber schon von Beginn an geladen sind. Im praktischen Test (siehe Anhang A.1.3) waren dies u.a. die Dateisystemtreiber NTFS (`ntfs.sys`) und FAT (`fastfat.sys`) für Festplatten und Disketten sowie CDFS (`cdfs.sys`) für CD-Laufwerke.

Für **Datenträger-Klassentreiber** läßt sich keine allgemeine Aussage treffen. Hier kommt es darauf an, ob sich alle Geräteknoten im laufenden Betrieb deaktivieren lassen, die den betreffenden Treiber verwenden. So ließen sich im Versuch (siehe Anhang A.1.2) alle Geräteknoten für CDROM-Laufwerke deaktivieren, was zur Folge hatte, daß der Datenträger-Klassentreiber `cdrom.sys` im laufenden Betrieb entladen wurde. Der Klassentreiber `disk.sys` für Festplatten konnte jedoch nicht entladen werden, da sich eine Deaktivierung aller Festplatten-Geräteknoten nicht im laufenden Betrieb durchführen ließ. Dies dürfte darauf zurückzuführen sein, daß sich das Betriebssystem selbst auf einer Festplatte befindet und die entsprechenden Treiber laufend benötigt.

Prinzipiell lassen sich **Filtertreiber** ebenfalls im laufenden Betrieb austauschen. Wenn sich der zugehörige Geräteknoten nicht im laufenden Betrieb deaktivieren und reaktivieren läßt, wirkt sich ein Austausch der Treiberdatei ebenfalls erst nach einem Neustart aus. Da sich die Festplatten-Geräteknoten offensichtlich nicht ohne Neustart deaktivieren lassen, können auch die Filtertreiber, die zu den entsprechenden Datenträgern gehören, nicht im laufenden Betrieb ausgetauscht werden. Welche das im einzelnen sind, hängt von der jeweiligen Konfiguration des Systems ab.

Installation neuer Treiber

Wenn sich ein vorhandener Treiber nicht austauschen läßt, so besteht noch die Möglichkeit einen neuen Treiber zu installieren. Der neue Treiber könnte dann die erforderlichen Manipulationen durchführen. Es gibt zwei Wege, wie ein neuer Treiber in das System gelangen kann: die automatische und die manuelle Treiberinstallation.

Wie in Abschnitt 2.4.6 beschrieben, versucht der PnP-Manager automatisch einen Treiber zu installieren, wenn ein neues Gerät an einem Bus gemeldet wird. Wenn er keinen geeigneten Treiber finden kann, wird der Benutzer gefragt, einen anzugeben. Wenn ein passender Treiber gefunden wurde, wird er automatisch installiert und sofort geladen, sofern der aktuell angemeldete Benutzer das Recht zum Laden von Treibern besitzt. Ein Trojaner-Treiber könnte somit in das System gelangen, wenn er zusammen mit einem neuen Gerät, z.B. einem Joystick, geliefert

und installiert wird.

Die manuelle Treiberinstallation geschieht normalerweise mit sogenannten Setup-Programmen, die die erforderlichen Eintragungen in der *Registry* machen. Durch Deaktivieren und anschließendes Reaktivieren des Geräteknotens, dem der Treiber zugeordnet ist, wird der Treiber auch im laufenden Betrieb geladen – sofern der Aufrufer das Recht zum Laden von Treibern besitzt. Grundsätzlich kann jedes beliebige Programm Treiber auf diese Weise installieren, wenn der Aufrufer die entsprechenden Rechte besitzt.

Dateisystemfiltertreiber können im laufenden Betrieb auch ohne Deaktivierung von Geräteknoten hinzugeladen werden. So wird beim Start des Programms *Filemon.exe* [SR00, S. 531] ein neuer Dateisystemfiltertreiber *Filem.sys* installiert und sofort geladen, wenn man das Programm als Administrator startet. Dies konnte mit dem selbst entwickelten Programm *listdrv.exe* (siehe Anhang A.1.3) überprüft werden.

Statt Treiber aus einer Treiberhierarchie auszutauschen, könnte ein Trojaner auch die Hierarchie verändern: Filtertreiber können, wie gezeigt, einfach zu einer bestehenden Treiberhierarchie hinzugefügt werden. Sie müssen jedoch in der *Registry* explizit als Filtertreiber eingetragen sein.

Gegenmaßnahme

Für den Austausch von Treiberdateien oder die Installation neuer Treiber sind grundsätzlich entsprechende Zugriffsrechte auf die Dateien und die *Registry* erforderlich. Diese Rechte haben normalerweise nur Administratoren. Das beschriebene Angriffsszenario trifft somit nicht auf normale Benutzer zu. Es bleibt jedoch die Gefahr, daß der Administrator durch die Installation zusätzlicher Programme, unwissentlich einen Trojaner-Treiber in das System bringt.

Um neue Treiber zu erkennen, könnte man nach einer Programminstallation zusätzlich die *Registry* nach Veränderungen der Treibereinträge überprüfen lassen. Insbesondere könnte nach neuen Filtertreibern gesucht werden, da sie als solche in der *Registry* gekennzeichnet sind. Um dies durchführen zu können, muß man den Zustand der *Registry* vor einer Programminstallation kennen. Der *ModuleStamper*-Dienst könnte diese Aufgabe übernehmen: Alle relevanten *Registry*-Einträge[8] werden ebenfalls wie die Treiberdateien „gestampt“, d.h. es wird ein Hashwert zu jedem Eintrag gebildet und mit einem Referenzwert verglichen, der bei der Installation des *ModuleStampers* berechnet und abgespeichert wurde. Bei einem Abweichen der Werte gibt der Dienst Alarm.

[8] Konkret sind dies alle Unterschlüssel von *HKLM\SYSTEM\CurrentControlSet\Services* für die Treiberdateien und alle Werte „UpperFilters“ und „LowerFilters“ unterhalb von *HKLM\SYSTEM\CurrentControlSet\Control\Class* und *HKLM\SYSTEM\CurrentControlSet\Enum* für Filtertreiber.

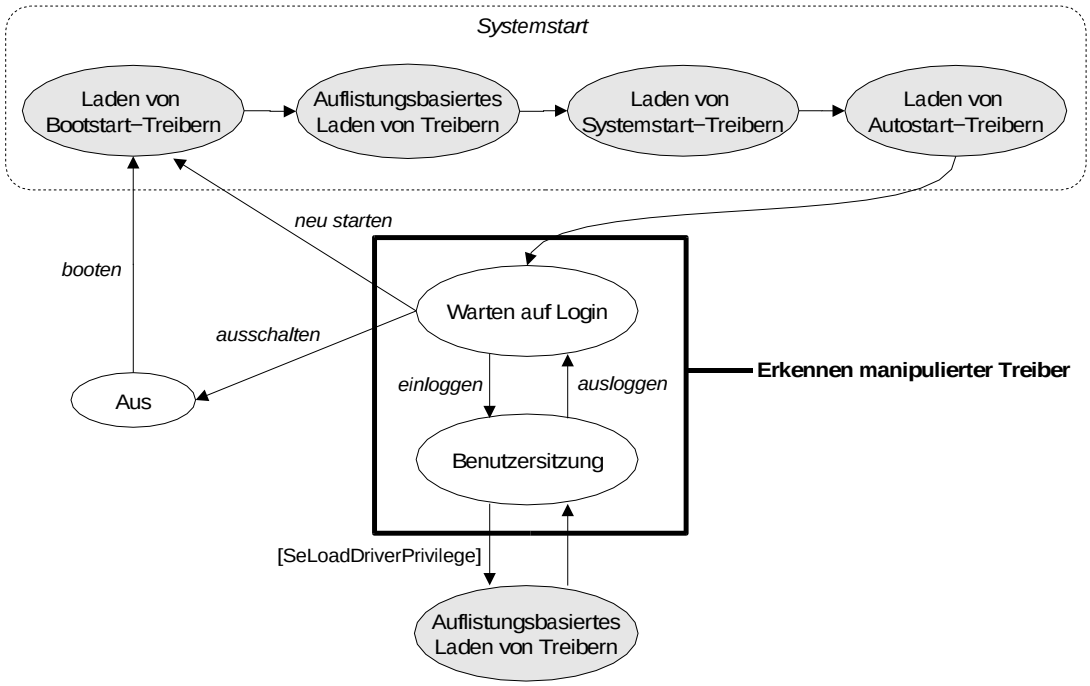

Abbildung 5.4.: Zustandsmodell von Windows 2000

Die Erkennung manipulierter Treiberdateien oder veränderter *Registry*-Einträge kann nicht mehr gewährleistet werden, wenn ein Trojaner-Treiber bereits geladen wurde. Er könnte den *ModuleStamper*-Prozeß manipulieren oder die Stealth-Technik anwenden. Bevor ein Treiber geladen wird, müssen also die vorher genannten Überprüfungen gemacht werden. Wenn man die Programmabläufe von Windows 2000 nur auf das Laden von Treibern abstrahiert, ergibt sich ein Zustandsmodell, wie in Abbildung 5.4 gezeigt wird.

Wird das System gebootet, werden zuerst die *Bootstart*-Treiber geladen. Dann folgt das auflistungsbasierte Laden, dem sich das Laden der *Systemstart*-Treiber und der *Autostart*-Treiber anschließt. Danach wird auf ein Login eines Benutzers gewartet. Wenn eine Benutzersitzung aktiv ist, kann erneut das auflistungsbasierte Laden von Treibern aktiviert werden, z.B. durch Deaktivierung und Reaktivierung von Geräteknoten wie oben beschrieben. Dafür benötigt der angemeldete Benutzer jedoch das Recht zum Laden von Treibern (*SeLoadDriverPrivilege*). Besitzt er das Recht nicht, werden Treiber erst wieder beim nächsten Systemstart oder nach Anmeldung eines entsprechend privilegierten Benutzers geladen. Ein Erkennen von ausgetauschten oder neuen Treibern ist nur möglich, wenn das System innerhalb der Zustände „Warten auf Login“ und „Benutzersitzung“[9] bleibt.

Für den *ModuleStamper* bedeutet dies, daß er die Überprüfungen der Treiberdateien und

[9]Der Zustand „Benutzersitzung“ stimmt hier mit der *Benutzersitzung* von Windows 2000 überein, solange keine Treiber geladen werden.

der *Registry* direkt nach einer Programminstallation durchführen muß, und bevor das System neu gestartet wird. Dienste bekommen vom SCM immer eine Meldung, wenn ein *Shutdown* durchgeführt wird, d.h. wenn das System herunterfährt. Nach Erhalt der Meldung können die Dienste lediglich noch Aufräum- oder Abschlußarbeiten durchführen. So überprüft der *ModuleStamper* die Stamps, wenn er das *Shutdown*-Signal erhält.

Damit Trojaner-Treiber nicht bereits im laufenden Betrieb geladen werden, muß den entsprechenden Installationsprogrammen das Recht *SeLoadDriverPrivilege* entzogen werden. Dies verhindert, daß man von der aktiven Benutzer-Session zum auflistungsbasierten Laden von Treibern gelangt.

5.2.2. Angriffe durch Debug-Prozesse

Solange ein Dienst aktiv ist, kann seine ausführbare Programmdatei (Exe-Datei) nicht gelöscht oder verändert werden. Selbst dem Administrator oder dem Systemkonto wird der Zugriff verweigert[10]. Der Grund liegt im Ressourcenschutz durch den Objekt-Manager. Kein Prozeß im Benutzermodus kann auf Dateien zugreifen, die von einem anderen Prozeß verwendet werden, und ein aktiver Dienst-Prozeß hält seine eigene Exe-Datei automatisch in Verwendung.

Allerdings könnte der Dienst-Prozeß selbst manipuliert werden. Unter Windows 2000 ermöglicht die Benutzerberechtigung *SeDebugPrivilege* den Zugriff auf das Prozeßhandle einer beliebigen laufenden Anwendung. Dieses Recht ist eigentlich dafür gedacht, um Programme damit zu *debuggen*, und ist standardmäßig nur den Administratoren zugeordnet.

Besitzt ein Prozeß das Debug-Privileg, kann er es jederzeit aktivieren[11]. Mit der Funktion `DebugActiveProcess()` kann er sich an einen beliebig anderen Prozeß „anheften“. Er kann dann den aktiven Thread des anderen Prozesses anhalten, beenden oder seinen Kontext ändern. So kann er z.B. den Instruktionszeiger neu setzen, so daß im anderen Prozeß als nächstes eine andere Routine ausgeführt wird, als ursprünglich definiert.

Ein Prozeß mit Debug-Privileg kann außerdem das Handle eines anderen Prozesses mit dem Zugriff PROCESS_ALL_ACCESS öffnen. Dies ermöglicht ihm, folgende Funktionen mit dem Handle aufzurufen:

- `TerminateProcess()`, um den anderen Prozeß zu beenden,

[10] Dies läßt sich mit dem „at“-Trick einfach nachprüfen: Als Administrator startet man z.B. mit dem Befehl „`at 15:00 /interactive cmd`“ um 15:00 Uhr eine interaktive Kommandoshell. Die Shell wird im Systemkonto ausgeführt. Wenn man nun versucht, die Exe-Datei eines laufenden Dienstes zu löschen, erhält man nur die Meldung *„Zugriff verweigert“*.

[11] In Windows 2000 können Privilegien in einem Zugriffstoken deaktiviert sein, obwohl sie dem Prozeß zugeordnet sind. Durch Aufruf der Funktion `AdjustTokenPrivileges()` können vorhandene Benutzerberechtigungen an- oder abgeschaltet werden. Jedoch läßt sich die Menge der in einem Token enthaltenen Rechte niemals erweitern (siehe [MS01g]).

- `CreateRemoteThread()`, um einen neuen Thread im anderen Prozeß zu starten,
- `ReadProcessMemory()`, um den privaten Arbeitsspeicher des anderen Prozesses auszulesen,
- `WriteProcessMemory()`, um den privaten Arbeitsspeicher des anderen Prozesses zu verändern.

Im Versuch konnte auf diese Weise der *ModuleStamper*-Dienst mit dem selbst entwickelten Programm *SilentKill.exe* beendet werden. Es zeigte sich, daß vom SCM im Systemereignisprotokoll des *Event Loggers* ein Eintrag über das unerwartete Beenden des *ModuleStamper*-Dienstes gemacht wurde.

Ein Angreifer könnte jedoch mit dem Debug-Privileg einen zusätzlichen Thread im Dienst-Prozeß erzeugen und starten, der alle übrigen Threads des Dienstes beendet und somit die Kontrolle übernimmt. Der SCM würde davon nichts merken, da der Dienst als Prozeß immer noch laufen würde. Weitere Manipulationen, wie das Austauschen von Komponenten, wären von diesem Zeitpunkt an unbemerkt möglich, da der *ModuleStamper* vom Angreifer kontrolliert wird.

Das Debug-Privileg ist nicht die einzige Möglichkeit auf einen anderen Prozeß derartigen Zugriff zu bekommen. Mit dem Inbesitznahmerecht (*SeTakeOwnershipPrivilege*), das standardmäßig nur der Administrator besitzt, kann man die Zugriffskontrolliste (ACL) jedes Objektes ändern. Ein Angreifer mit *SeTakeOwnershipPrivilege* könnte das Prozeßobjekt eines Dienstes in seinen Besitz nehmen und dann die ACL so abändern, daß er ein Handle des Prozesses mit PROCESS_ALL_ACCESS öffnen darf. Damit sind analoge Angriffe wie mit *SeDebugPrivilege* möglich.

Gegenmaßnahme Normale Benutzer sollten die Rechte *SeDebugPrivilege* und *SeTakeOwnershipPrivilege* grundsätzlich nicht besitzen – was standardmäßig der Fall ist –, da sonst Dienste oder andere Programme manipuliert werden können. Wenn der Administrator zusätzliche Software installiert oder nichtvertrauenswürdige Programme ausführt, müssen diese Privilegien zumindest temporär entzogen werden.

5.2.3. Angriffe durch neue Konten

Winlogon erstellt beim Anmeldevorgang eines Benutzers einen Starttoken, *„der den sich anmeldenden Benutzer repräsentiert, und verknüpft diesen Token mit dem Anmelde-Shellprozeß des Benutzers. Alle Programme, die der Benutzer ausführt, erben eine Kopie dieses Starttokens“* [SR00, S. 424].

Wenn man davon ausgeht, daß der Administrator zusätzliche Software installiert und auf diese Weise ein Trojaner-Programm gestartet wird, dann bekommt auch der Trojaner den Starttoken des Administrators. Mit der Funktion *PrivilegeCheck()* kann er überprüfen, welche Privilegien in dem Token enthalten sind, und sieht, ob die erforderlichen Benutzerrechte für einen Angriff fehlen, wie z.B. *SeLoadDriverPrivilege* oder *SeDebugPrivilege*. Da Administratoren die Benutzerrechte für alle Benutzer (und damit auch für sich selbst) ändern können, könnte man annehmen, daß ein Angreifer als Administrator sich die erforderlichen Rechte wiedergibt.

Obiger zitierter Sachverhalt bedeutet jedoch, daß neu hinzugenommene Benutzerrechte nicht in der aktuellen Benutzersitzung aktiv werden können. Ein eingeloggter Benutzer verwendet immer den Token, den er beim Anmeldevorgang von Winlogon erhalten hat. Auch wenn er Prozesse neu startet, nachdem ihm weitere Privilegien eingeräumt wurden, verwenden diese noch den alten Token mit den alten Privilegien[12].

Um einen neuen Token mit den neuen Privilegien zu bekommen, muß man sich neu anmelden, oder man startet mit dem Befehl „runas“ eine neue Sitzung innerhalb der aktuellen Benutzersitzung. Sowohl Winlogon als auch *runas* erfordern die Eingabe des Benutzernamens und des Kennworts. Letzteres dürfte einem Angreifer nicht bekannt sein, so daß er keinen neuen Token bekommen kann.

Administratoren haben die Möglichkeit, neue Benutzerkonten anlegen zu können. Ein Angreifer im Administratorkonto könnte nun ein neues Konto erstellen und diesem die für seinen Angriff erforderlichen Privilegien geben. Da der Angreifer für das neue Konto das Paßwort selbst definieren darf, kann er auch einen Token für das neue Konto bekommen und einen Prozeß damit starten. Folgende Vorgehensweise ermöglicht einen solchen Angriff:

1. Mit der Funktion `NetUserAdd()` wird ein neues Konto inkl. Paßwort angelegt.
2. Mit `LsaAddAccountRights()` werden dem neuen Konto Privilegien zugeordnet.
3. Mit `LogonUser()` kann man einen neuen Token für das neue Konto erstellen lassen, wenn man Kontoname und Paßwort als Argumente übergibt.
4. Mit `CreateProcessAsUser()` läßt sich ein neuer Prozeß mit diesem Token starten.

Der neue Prozeß läuft mit dem Token des neu angelegten Kontos und besitzt die zugeordneten Privilegien. Wurde dem Konto z.B. das Debug-Privileg zugeordnet, kann der neue Prozeß andere Prozesse manipulieren. Prinzipiell könnte der neue Prozeß damit alle Windows 2000 Privilegien bekommen, was ihm weitreichende Manipulationsmöglichkeiten bietet – mehr als der Administrator standardmäßig besitzt!

[12] siehe hierzu auch [MS01g] und [Kim98]

Um diese Vorgehensweise durchführen zu können, benötigt der Angreifer jedoch noch einige Rechte. Um `LsaAddAccountRights()` aufrufen zu können, benötigt man ein Handle auf das sogenannte *Policy*-Objekt[13] mit den Zugriffsrechten POLICY_LOOKUP_NAMES und POLICY_CREATE_ACCOUNT. Administratoren haben diese Zugriffsrechte.

Um `LogonUser()` aufrufen zu können, ist das Recht *SeTcbPrivilege* notwendig[14]. Dieses Benutzerrecht erlaubt Prozessen, als Teil der vertrauenswürdigen Computerumgebung (*trusted computer base*) ausgeführt zu werden. Für die Erstellung eines primären Zugriffstoken wird das Recht *SeCreateTokenPrivilege* benötigt. Normalerweise besitzt nur das Systemkonto diese beiden Privilegen. In bestimmten Fällen ist darüber hinaus das Recht *SeChangeNotifyPrivilege* erforderlich, Administratoren oder das Systemkonto benötigen es nicht.

Für die Funktion `CreateProcessAsUser()` sind schließlich noch in diesem Zusammenhang die Rechte *SeAssignPrimaryTokenPrivilege* und *SeIncreaseQuotaPrivilege* erforderlich. Administratoren besitzen standardmäßig letzteres, nur das Systemkonto besitzt beide.

Gegenmaßnahme Um zu verhindern, daß ein neues Konto angelegt wird, könnte man die Zugriffskontrolliste des *Policy*-Objekts von Windows 2000 dahingehend ändern, daß der Administrator keinen Zugriff mehr darauf hat. Jedoch dürfte dies nur temporär für die Dauer der Ausführung der nichtvertrauenswürdigen Programme sein, da der Administrator sonst keine Benutzerkonten mehr verwalten kann. Das Anlegen neuer Konten bei einer Programminstallation könnte aber auch ein legaler Vorgang sein, beispielsweise das Erstellen eines eigenen Verwaltungskontos für eine Datenbankanwendung.

Es scheint sinnvoll zu sein, nur die letzten beiden Schritte, das Erzeugen eines Tokens und das Starten eines Prozesses im neuen Konto, zu unterbinden. Das Erstellen eines Kontos an sich stellt noch keine Gefahr dar und kann sogar als Hinweis für einen Angriff gesehen werden.

Damit kein primärer Token erstellt werden kann, dürfen dem Angreifer die Rechte *SeTcbPrivilege* und *SeCreateTokenPrivilege* nicht zur Verfügung stehen. Zusätzlich dürfen nichtvertrauenswürdige Programme nicht das Recht *SeAssignPrimaryTokenPrivilege* besitzen, damit keine Prozesse mit neuen Token erstellt werden können. Standardmäßig haben weder Administratoren noch normale Benutzer diese drei Rechte.

Die anderen Privilegien – *SeIncreaseQuotaPrivilege* und *SeChangeNotifyPrivilege* – sollten in ihrer standardmäßigen Zuordnung nicht verändert werden, da sie auch für andere Aufgaben notwendig sind (siehe [MS01g, "Windows NT Privileges"]).

[13] nicht das *Policy*-Objekt von Java!
[14] Unter Windows XP ist dieses Privileg nicht mehr erforderlich! (siehe [MS01g])

5.2.4. Angriffe durch andere Dienste

Ein Programm, das im Administratorkonto ausgeführt wird, kann durch den Entzug der entsprechenden Privilegien daran gehindert werden, daß es Treiber lädt, Prozesse debuggt oder Prozesse in anderen Konten startet. Es kann aber neue Dienste installieren und starten oder bestehende Dienste benutzen. Dadurch kann einem Angreifer ein Zugang zum Systemkonto ermöglicht werden, wie im folgenden gezeigt wird. Das Systemkonto hat per Definition die meisten Privilegien zugeordnet[15], darunter auch die bisher genannten, so daß die geschilderten Angriffstechniken wieder zur Verfügung stehen würden.

Ein Angreifer im Administratorkonto könnte einen neuen Dienst installieren, der ein Trojanisches Pferd ist. Dienste laufen standardmäßig im Systemkonto und lassen sich derart installieren, daß sie im laufenden Betrieb sofort gestartet werden können. Der Prozeß des Trojaner-Dienstes kann also vom Angreifer sofort gestartet werden und hat damit alle Privilegien des Systemkontos, insbesondere *SeLoadDriverPrivilege*, *SeDebugPrivilege* und *SeTakeOwnershipPrivilege*.

Prozesse im Systemkonto haben standardmäßig Zugriff auf alle Objekte, die mit einer Standard-DACL erstellt wurden, d.h. auf alle Objekte, die keine explizite ACL erhalten haben. Aufgrund des Inbesitznahmerechtes können Systemkonto-Prozesse zusätzlich jedes beliebige Objekt im Benutzermodus in ihren Besitz nehmen und die ACL ändern. Allerdings kann das Objekt nicht dem ursprünglichen Besitzer zurückgegeben werden. Der ursprüngliche Besitzer muß es explizit wieder in seinen Besitz nehmen. Eine Manipulation der ACL könnte auf diese Weise bemerkt werden.

Ein Trojaner-Dienst kann sofort im laufenden Betrieb Treiber laden, die im Kernelmodus operieren und weitere Manipulationen durchführen können, oder er kann andere Prozesse, insbesondere andere Dienst-Prozesse, direkt manipulieren oder beenden. Er kann durch die weitreichenden Privilegien des Systemkontos somit den Benutzermodus kontrollieren und durch das Laden eines Trojaner-Treibers auch den Kernelmodus, d.h. er kann Zugriff auf *alle* Objekte des Systems haben. Ein Dienst im Systemkonto bietet einem Angreifer damit prinzipiell die Möglichkeit, sämtliche Betriebssystembestandteile und Anwendungen unbemerkt zu manipulieren.

Mit Hilfe von *at.exe* gelangt man ebenfalls vom Administratorkonto ins Systemkonto. Wie schon zuvor erwähnt, lassen sich mit *at.exe* zeitgesteuert beliebige Programme starten, die im Systemkonto ausgeführt werden. *At.exe* benutzt dazu den *Schedule*-Dienst (auch Taskplaner-Dienst genannt, in *MsTask.exe* implementiert), der im Systemkonto läuft. In der graphischen Benutzerschnittstelle „Systemsteuerung / Geplante Tasks“ des *Schedule*-Dienstes lassen sich für die zu startenden Programme explizit Benutzerkonten angeben – wenn das Paßwort bekannt

[15] Für eine genaue Liste der zugeordneten Privilegien siehe [MS01d].

ist. *At.exe* bietet diese Möglichkeit nicht und meldet alle zu startenden Programme beim *Schedule*-Dienst ohne explizites Benutzerkonto an. Das hat zur Folge, daß der *Schedule*-Dienst die von *at.exe* angegebenen Programme im Systemkonto startet. Ein Angreifer könnte auf diese Weise zu einer bestimmten Zeit ein Trojaner-Programm starten, das somit alle Privilegien des Systemkontos besitzt.

Das Programm *at.exe* ruft für seine Eintragungen beim *Schedule*-Dienst die Funktion `NetScheduleJobAdd()`[16] auf. Diese Funktion kann nur von Mitgliedern der lokalen Administratorengruppe aufgerufen werden. In praktischen Versuchen stellte sich heraus, daß diese Funktion und das Programm *at.exe* mit einer Fehlermeldung abbrechen, wenn der *Schedule*-Dienst deaktiviert ist; es werden keine Programme zur zeitgesteuerten Ausführung eingetragen.

Statt die Funktionen `LogonUser()` und `CreateProcessAsUser()` zu verwenden, um einen Angriff durch ein neues Konto durchzuführen, kann ein Angreifer auch die Funktion `CreateProcessWithLogonW()`[17] aufrufen, die kein *SeTcbPrivilege* benötigt und die beiden anderen Funktionen ersetzt. Die Funktion kann von jedem beliebigen Benutzer aufgerufen werden und startet einen neuen Prozeß in einem anderen Konto – wenn das korrekte Paßwort angegeben wird.

Im Gegensatz zu `LogonUser()`, welches einen Token von *Lsass* direkt erstellen läßt, beauftragt `CreateProcessWithLogonW()` den *SecLogon*-Dienst, von *Lsass* einen Token anzufordern. Der Befehl *runas* verwendet ebenfalls den *SecLogon*-Dienst. Ist dieser Dienst deaktiviert, brechen sowohl `CreateProcessWithLogonW()` als auch *runas* mit einer Fehlermeldung ab und können keine Prozesse starten, was im praktischen Versuch nachgewiesen werden konnte.

Gegenmaßnahme Das Systemkonto hat seine zugeordneten Privilegien per Definition, d.h. sie sind fest eincodiert und lassen sich nicht ändern. Einem Angreifer muß folglich der Zugang zum Systemkonto unmöglich gemacht werden.

Das Löschen von *at.exe* würde keinen Erfolg bringen, da die API-Funktion `NetScheduleJobAdd()` einem Angreifer weiterhin zur Verfügung stehen würde. Der *Schedule*-Dienst läßt sich in keinem anderen Konto als dem Systemkonto ausführen[18]. Mit `NetScheduleJobEnum()` kann man zwar alle Aufträge des *Schedule*-Dienstes auflisten, bereits gestartete Aufträge werden jedoch nicht aufgeführt. Ein Trojaner-Programm könnte bereits ausgeführt worden sein, wenn eine derartige Überprüfung durchgeführt werden soll.

[16] siehe [MS01e]
[17] siehe [MS01c]
[18] Ein Versuch, den *Schedule*-Dienst in einem Benutzerkonto oder im Administratorkonto laufen zu lassen, brachte nur eine Fehlermeldung hervor.

Als Ausweg bleibt, den *Schedule*-Dienst während der Ausführung nichtvertrauenswürdiger Programme zu deaktivieren. Ebenso muß der *SecLogon*-Dienst deaktiviert werden, damit die Funktion `CreateProcessWithLogonW()` nicht dazu benutzt werden kann, einen Angriff durch neue Konten durchzuführen. Es ist dabei darauf zu achten, daß der Angreifer die heruntergefahrenen Dienste nicht wieder aktivieren kann. Die Win32-API bietet dazu die Möglichkeit, die Datenbank des Dienststeuerungsmanagers zu sperren, so daß keine Dienste mehr gestartet werden können, außer von dem einzigen Prozeß, der die Sperre hält.

5.3. Beschränkte Installationsumgebung

Um die potentiellen Angriffe abzuwehren, die bei einer Installation nichtvertrauenswürdiger Programme und ihrer Ausführung auftreten können, lege es am nächsten, ein eigenes Benutzerkonto mit beschränkten Rechten dafür einzurichten. Unter Windows 2000 ist dies jedoch praktisch nicht durchführbar.

So wurde im Versuch ein Installationskonto eingerichtet, daß zwar nicht zur Gruppe der Administratoren gehörte, jedoch *alle* Privilegien besaß und Vollzugriff auf alle Dateien und Verzeichnisse hatte. Trotz dieser weitreichenden Rechte konnte beispielsweise weder das *Windows 2000 Service Pack 2* noch das *Internet Explorer 5 Service Pack 2* unter diesem Konto installiert werden. Die Installation wurde mit der Meldung „Sie besitzen nicht das Recht, Windows 2000 zu aktualisieren“ bzw. „Sie haben auf diesem Computer keine Administratorberechtigung“ abgebrochen. Die Rechte des Installationskontos zu beschränken würde erst recht keinen Erfolg bringen.

Der Grund dürfte darin zu finden sein, daß diese Installationsprogramme zwar administrative Privilegien benötigen aber nicht die vorhanden Privilegien abfragen, sondern nur prüfen, ob der Aufrufer ein Administrator ist. Dies kann jedes Programm einfach dadurch erreichen, indem es die Benutzer-SID seines Tokens untersucht. Der Administrator hat auf jedem System immer die konstante Endung „-500“ in seiner SID. Mit der Funktion `CheckTokenMembership()` können Programme die Mitgliedschaft ihres Aufrufers in der Administratorengruppe überprüfen.

Ein Angreifer könnte ebenfalls diese Überprüfung machen und seine Installation verweigern, wenn er nicht von einem Administratorkonto gestartet wird. Es ist anzunehmen, daß in der Praxis viele Programme diese Überprüfung machen, um festzustellen, ob die erforderlichen Rechte vorhanden sind – obwohl dies vielleicht gar nicht der Fall ist[19]. Aus Gewohnheit würde

[19] Man könnte dem Administrator sämtliche Privilegien entziehen und die ACLs aller Dateien und Verzeichnis so ändern, daß er keinen Schreibzugriff mehr darauf hat. Diese Programme würden bei der Installation dennoch nach dem Administratorkonto verlangen.

der Anwender die Programme im Administratorkonto installieren, wenn sie darauf bestehen. Einem Trojaner würde dies die Möglichkeit für einen erfolgreichen Angriff bieten.

Es erscheint aber auch nicht sinnvoll, dem Administrator Privilegien zu entziehen, da er sonst seine Verwaltungsaufgaben nicht vollständig durchführen könnte. Im folgenden wird deshalb eine beschränkte Ausführungsumgebung vorgestellt, die im Rahmen dieser Diplomarbeit entwickelt wurde und die die Installation und Ausführung nichtvertrauenswürdiger Programme im Administratorkonto ermöglicht, ohne daß die Privilegien des Administrators geändert werden und ohne daß die zuvor beschriebenen Angriffe durchgeführt werden können.

5.3.1. Konzept der Installationsshell

Meldet sich ein Benutzer in Windows 2000 an, wird ein Zugriffstoken für ihn erstellt, der seine Konto-SID, seine Gruppen-SIDs und seine Privilegien enthält. Mit diesem Token wird ein Shell-Prozeß gestartet, normalerweise *Explorer.exe*. Alle Programme, die der Benutzer startet, werden aus dieser Shell gestartet und erben den Zugriffstoken.

Es gibt jedoch noch die Möglichkeit, Prozesse mit einem *beschränkten Token* zu starten. Ein beschränkter Token wird aus dem Zugriffstoken abgeleitet, indem eine oder mehrere SIDs oder Privilegien entfernt werden. Die Win32-API bietet dazu die Funktion `CreateRestricted-Token()`. Übergibt man einen solchen Token der Funktion `CreateProcessAsUser()`, wird der neue Prozeß mit den beschränkten Rechten gestartet.

Die Idee ist, innerhalb der normalen Shell des Administrators, die Prozesse mit dem Zugriffstoken startet, eine zweite Shell einzusetzen, die Prozesse mit einem beschränkten Token startet. Diese zweite Shell wird im folgenden *Installationsshell* genannt. Nur in dieser Installationsshell darf der Administrator nichtvertrauenswürdige Programme installieren und ausführen.

Die Installationsshell erzeugt aus dem Zugriffstoken des aktuell angemeldeten Benutzers, in dem Fall des Administrators, einen beschränkten Token. Die SIDs werden im beschränkten Token wie im normalen Token belassen, da Programme unter Umständen diese Information und die zugehörigen Zugriffsrechte benötigen. Es werden jedoch einige Privilegien entfernt, die für einen Angriff ausgenutzt werden könnten. So sind die folgenden Privilegien im beschränkten Token nicht enthalten:

- *SeLoadDriverPrivilege*
- *SeDebugPrivilege*
- *SeTakeOwnershipPrivilege*
- *SeTcbPrivilege*

- *SeCreateTokenPrivilege*
- *SeAssignPrimaryTokenPrivilege*

Um zu verhindern, daß Treiber im laufenden Betrieb in den Kernelmodus geladen werden, muß das Recht *SeLoadDriverPrivilege* entfernt werden. Damit ein Angreifer keine anderen laufenden Prozesse manipulieren kann, darf er die Rechte *SeDebugPrivilege* und *SeTakeOwnershipPrivilege* nicht besitzen. Um einen Angriff durch neue Benutzerkonten auszuschließen, müssen die letzten drei Rechte entzogen werden. Ohne das Recht *SeTcbPrivilege* kann ein Angreifer die Funktion `LogonUser()` nicht aufrufen, um einen neuen Token erstellen zu lassen, und ohne das Recht *SeCreateTokenPrivilege* kann er auch selbst keinen neuen Token erzeugen. Das Fehlen von *SeAssignPrimaryTokenPrivilege* schließlich verhindert, daß die Funktion `CreateProcessAsUser()` einen Prozeß mit einem neuen Token starten kann. Sie kann höchstens noch einen Prozeß mit einem beschränkten Token des Aufrufers starten.

Die Installationsshell selbst besitzt den normalen Zugriffstoken. Aber alle Prozesse, die mit der Installationsshell gestartet werden, bekommen den beschränkten Token zugewiesen. Damit ist gewährleistet, daß sie die oben genannten Privilegien nicht besitzen und dennoch im Administratorkonto laufen. Eine Überprüfung der Mitgliedschaft in der Administratorengruppe würde demnach erfolgreich verlaufen, jedoch nicht die Durchführung von Operationen, die die entzogenen Privilegien benötigen. Auch wenn die Prozesse, die in der Installationsshell gestartet wurden, weitere Prozesse starten, erben diese den beschränkten Token und können die entzogenen Privilegien nicht wiedererlangen.

Ein beschränkter Token reicht allein nicht aus, um Angriffe zu verhindern. Da die mit der Installationsshell gestarteten Prozesse noch die SID eines Administrators haben (und evtl. auch benötigen), können sie trotz beschränktem Token Dienste installieren und starten oder vorhandene Dienste benutzen. Um auszuschließen, daß ein Angreifer die Funktion `CreateProcessWithLogonW()` benutzen kann, darf der *SecLogon*-Dienst nicht aktiv sein. Damit kein Angreifer mittels *at.exe* einen Prozeß im Systemkonto startet, darf der *Schedule*-Dienst nicht aktiv sein. Würde es einem Angreifer gelingen, einen Prozeß im Systemkonto zu starten, hätte er alle Privilegien zur Verfügung, um Manipulationen am System unbemerkt durchführen zu können.

Mit dem Konzept der Installationsshell bietet sich aber eine Möglichkeit, einem Angreifer den Weg ins Systemkonto zu versperren. Die Installationsshell kann bei ihrem Start die Dienste herunterfahren, die für einen Angriff mißbraucht werden könnten. Wird die Shell im Administratorkonto ausgeführt, hat sie die Berechtigung dazu. Zusätzlich muß sie dafür sorgen, daß ein potentieller Angreifer weder die heruntergefahrenen Dienste noch einen eigenen Dienst starten

kann. Windows 2000 bietet dafür die Funktion `LockServiceDatabase()`[20], mit der die Datenbank des Dienststeuerungsmanagers (SCM) gesperrt wird.

Das Sperren der SCM-Datenbank bewirkt, daß kein Dienst mehr gestartet werden kann, solange die Sperre nicht wieder mit der Funktion `UnlockServiceDatabase()` freigegeben wird. Zu jeder Zeit kann jeweils nur ein einziger Prozeß die Sperre der SCM-Datenbank besitzen. Kein anderer Prozeß kann die Sperre aufheben.

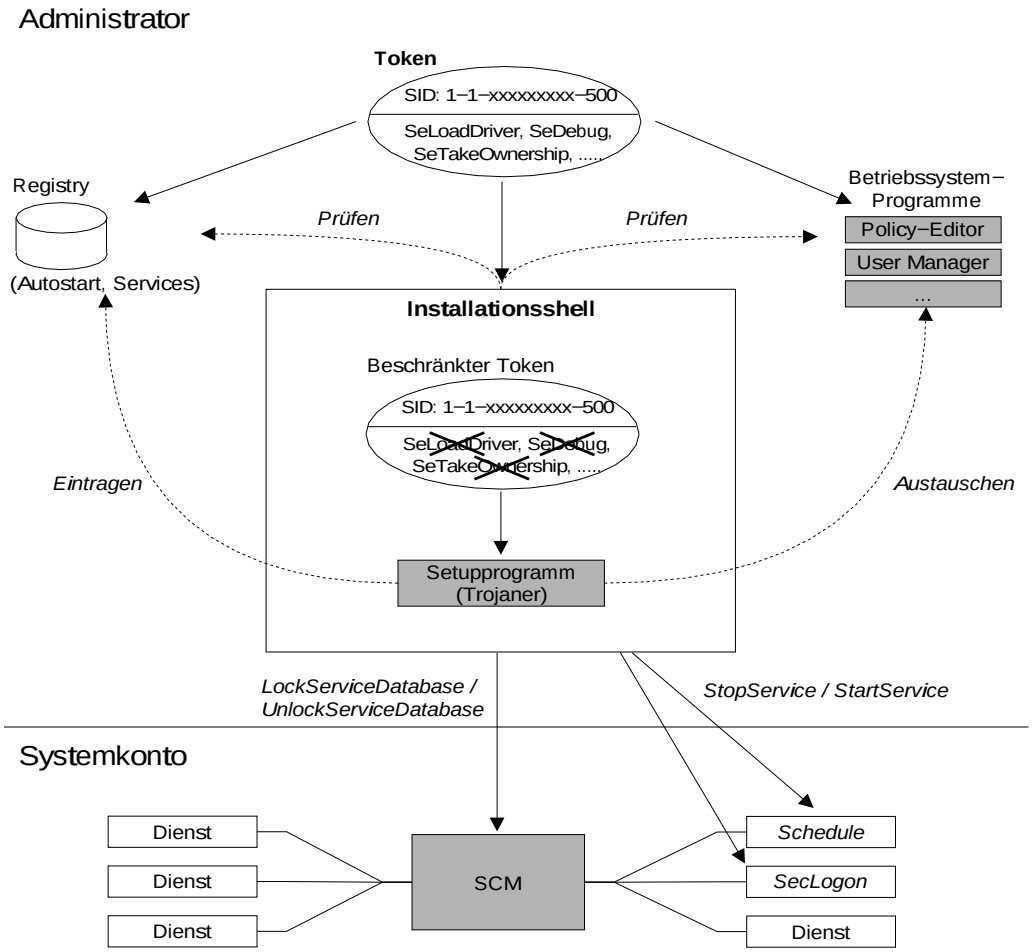

Abbildung 5.5.: Architektur der Installationsshell

Die Installationsshell führt nun bei ihrem Start folgende Aktionen aus:

1. Herunterfahren des *SecLogon*-Dienstes
2. Herunterfahren des *Schedule*-Dienstes
3. Sperren der SCM-Datenbank

[20] siehe [MS01c] und [Ric98]

Damit besitzt der Prozeß der Installationsshell die Sperre der SCM-Datenbank. Ein Angreifer kann lediglich Dienste installieren, aber starten lassen sie sich nicht. Ebenso können die beiden Dienste *SecLogon* und *Schedule* nicht gestartet werden, so daß `CreateProcessWithLogonW()` und *at.exe* nicht verwendet werden können. Ein Angreifer, der aus der Installationsshell gestartet wird, hat keine Möglichkeit mehr, ins Systemkonto zu gelangen. Zusammen mit dem beschränkten Token sind auch die anderen beschriebenen Angriffsmöglichkeiten ausgeschlossen. Abbildung 5.5 zeigt die Architektur der Installationsshell.

Wenn die Installationsshell beendet wird, gibt sie die SCM-Datenbank wieder frei und startet die Dienste *SecLogon* und *Schedule*. Bevor die Installationsshell beendet wird, ist darauf zu achten, daß keine Prozesse mehr laufen, die aus ihr gestartet wurden, und auch keine Kind-Prozesse, die von diesen Prozessen gestartet wurden. Zwar besitzen diese Prozesse noch den beschränkten Token, aber sie könnten nach Beenden der Shell die beiden genannten Dienste benutzen oder eigene starten und somit Prozesse mit den Privilegien des Systemkontos erzeugen.

5.3.2. Angriffe auf die Installationsshell und ihre Abwehr

Da Prozesse, die aus der Installationsshell gestartet werden, die gleiche SID wie die Installationsshell haben, besitzen sie damit das Recht, die Installationsshell zu terminieren. Da beim Terminieren die Sperre der SCM-Datenbank automatisch vom SCM aufgehoben wird, muß dies immer als Angriff gewertet werden.

Einem Angreifer kann diese Möglichkeit genommen werden, wenn man die Installationsshell selbst als Dienst im Systemkonto implementiert. Um den korrekten Token des Benutzers zu erhalten, um daraus einen beschränkten Token zu erstellen, müßte der Benutzer sich neben der normalen Anmeldung beim Betriebssystem noch ein zweites Mal bei der Installationsshell anmelden. Die Installationsshell selbst würde mit dem Token des Systemkontos laufen, welches eine eigene SID besitzt, und mit der Funktion `LogonUser()` vom Betriebssystem einen Token des bei ihr angemeldeten Benutzers bekommen. Auf diese Weise hätten die Installationsshell und von ihr gestartete Prozesse eine unterschiedliche Benutzer-SID. Diese Prozesse könnten die Shell nicht mehr terminieren. Da das plötzliche Beenden der Shell durch Terminieren im allgemeinen jedoch für den Benutzer erkennbar ist, wurde aus Einfachheit die Installationsshell als normale Anwendung entwickelt.

Ein Angriffsversuch muß nicht sofort während der Programminstallation erfolgen. So bietet Windows 2000 die Möglichkeit, in der *Registry* und im Startmenü „Autostart-Programme“ einzutragen, die bei der nächsten Anmeldung des Benutzers in seiner *Explorer*-Shell ausgeführt werden. Ein Angreifer kann zudem einen Dienst oder einen Treiber in der *Registry*[21] eintragen,

[21] Dienste und Treiber stehen unter *HKLM\SYSTEM\CurrentControlSet\Services*.

der automatisch beim nächsten Neustart des Betriebssystems gestartet wird. Beim Beenden der Installationsshell und noch vor dem Freigeben der SCM-Datenbank muß deshalb die *Registry* und das Startmenü nach entsprechenden Einträgen überprüft werden.

Ferner hat ein Angreifer, der innerhalb der Installationsshell ausgeführt wird, alle Zugriffsrechte, die ihm seine Benutzer- und Gruppen-SIDs ermöglichen. Im Falle des Administrators könnte er somit Dateien des Betriebssystems oder von Anwendungen modifizieren. Alle Dateien, die zu vertrauenswürdigen Komponenten gehören, müssen folglich beim Beenden der Installationsshell auch auf Veränderungen hin überprüft werden. Für Dateien, die gerade von anderen Prozessen in Benutzung sind, kann in der *Registry*[22] eingetragen werden, daß sie beim nächsten Neustart des Betriebssystems ausgetauscht werden. Auch dieser Eintrag muß geprüft werden.

Damit die Installationsshell nicht umgangen werden kann, darf die Eigenschaft „Autorun“ von CD-ROM-Laufwerken in der *Registry*[23] nicht aktiviert sein, da sonst beim Einlegen einer CD-ROM von der *Explorer*-Shell automatisch ein Programm gestartet wird.

5.3.3. Implementation

Da der *ModuleStamper*-Dienst schon Dateien auf Veränderungen hin untersucht, wurde die Überprüfung der *Registry* auch dort auf die gleiche Weise implementiert, d.h. für die relevanten *Registry*-Einträge werden „Stamps“ gebildet und bei einer Überprüfung die aktuellen Werte mit den Referenzwerten verglichen. Die Referenzwerte der *Registry*-Einträge werden zusammen mit der Bezeichnung ihrer Schlüssel in der Stampdatei gespeichert.

Die Installationsshell ruft beim Beenden und vor Freigeben der SCM-Datenbank automatisch den *ModuleStamper*-Dienst über eine eigens von ihm für sie kreierte *Named Pipe* auf und läßt sich einen Bericht geben. Falls Dateien manipuliert oder oben genannte Eintragungen in der *Registry* oder im Startmenü verändert oder hinzugefügt wurden, ergeben sich unterschiedliche Stamp-Werte und dies wird dem Benutzer sofort angezeigt. Die Überprüfung kann zusätzlich jederzeit manuell von der Installationsshell aus gestartet werden.

Während die Installationsshell aktiv ist, läßt sich der *ModuleStamper*-Dienst nicht konfigurieren. Andernfalls könnte ein Trojaner, der zwar aus der Installationsshell gestartet wurde aber dennoch eine Administrator-SID besitzt, die Konfiguration des Dienstes ändern, beispielsweise die Liste der zu überprüfenden Anwendungen. Der *ModuleStamper*-Dienst prüft deshalb bei

[22] Der Eintrag befindet sich im Wert „PendingFileRenameOperations“ im Schlüssel *HKLM\SYSTEM\ CurrentControlSet\Control\Session Manager* (siehe [MS01j]).

[23] Unter *HKLM\SYSTEM\CurrentControlSet\Services\CDRom* kann das Autorun-Verhalten mittels „Autorun=0“ global abgeschaltet werden und unter *HKCU\Software\Microsoft\Windows\CurrentVersion\ Policies\Explorer* mittels „NoDriveTypeAutoRun=0x00000095“ individuell für jeden Benutzer (siehe [MS01k]).

Anfragen zur Änderung seiner Konfiguration die Liste der aktiven Prozesse. Wenn sich darunter die Installationsshell befindet, verweigert er die Anfrage.

Um zu verhindern, daß die Installationsshell beendet wird, während noch von ihr gestartete Prozesse laufen, werden alle gestarteten Prozesse in einem *Auftragsobjekt* verwaltet. Kind-Prozesse, die wiederum von diesen Prozessen gestartet werden, gehören somit automatisch zu dem Auftragsobjekt. Nur wenn keine Prozesse mehr im Auftragsobjekt gelistet werden, läßt sich die Installationsshell ordnungsgemäß beenden. Andernfalls gibt sie eine Warnmeldung aus und verweigert ihr Beenden[24].

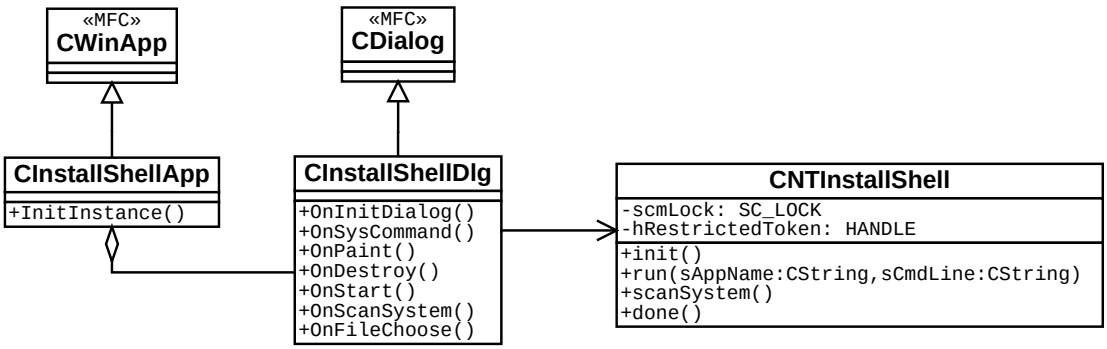

Abbildung 5.6.: Klassendiagramm der Installationsshell

Die Installationsshell *InstallShell.exe* wurde in C++ implementiert, da die Systemfunktionen von Windows 2000 in C vorliegen und auf diese Weise optimal eingebunden werden konnten. Dazu wurde eine graphische Benutzerschnittstelle entwickelt, die die *Microsoft Foundation Classes* (MFC) verwendet, um ein Dialogfeld anzuzeigen, in dem man die auszuführende Programmdatei wählen und ihr Startargumente übergeben kann.

Die Klasse `CInstallShellApp` repräsentiert die Hauptanwendung, die von Windows gestartet wird. Sie besitzt ein Dialogfeld, das in der Klasse `CInstallShellDlg` implementiert ist. Das Dialogfeld verwendet ein Objekt der Klasse `CNTInstallShell`, die die eigentliche Funktionalität der Installationsshell implementiert.

[24] Dies verhindert jedoch nicht ihr gewaltsames Terminieren!

6. Anwendungen

In diesem Kapitel werden zwei kryptographische Anwendungen vorgestellt, die im Rahmen dieser Diplomarbeit entwickelt wurden. Abschnitt 6.1 beschreibt die Architektur und das Design einer vollständigen Implementation des Kryptographie-Standards IEEE P1363 und wie eine darauf beruhende Anwendung zur Verschlüsselung und Signierung von Daten robust in Windows 2000 integriert werden kann. In Abschnitt 6.2 wird ein Konzept präsentiert, wie unter Windows 2000 eine exklusive Kommunikation von Java-Programmen unter Verwendung der kryptographischen Module gemäß P1363 realisiert werden kann.

6.1. Architektur einer Java-Implementation von IEEE P1363

Der Standard IEEE P1363 [IEE99] spezifiziert Funktionen und Schemata zur Durchführung kryptographischer Operationen, wie das Verschlüsseln oder Signieren von Nachrichten, basierend auf Public-Key-Kryptographie. Die dazu nötigen Schlüssel und Schlüsselparameter werden ebenfalls spezifiziert.

Der Standard spezifiziert mathematische *Primitives* basierend auf dem Diskreten Logarithmus (DL) Problem, dem Problem der Faktorisierung großer ganzer Zahlen (*integer factorization*, IF) und dem Problem des Diskreten Logarithmus in Elliptischen Kurven (EC). Diese *Primitives* können Basisoperationen durchführen, wie erstens das Ver- und Entschlüsseln einer Zahl und zweitens das Bilden und die Verifikation einer Signatur von einer Zahl. Drittens können sie aus den Schlüsseln zweier Parteien einen gemeinsamen geheimen Wert ableiten, aus dem ein geheimer Schlüssel für eine symmetrische Verschlüsselung gebildet werden kann. Es werden Hilfsfunktionen spezifiziert, die verschiedene Datentypen konvertieren, eine Nachricht in eine Zahl codieren und umgekehrt sowie Nachrichten maskieren oder Hashwerte bilden.

Basierend auf den *Primitives* und Hilfsfunktionen werden Schemata für digitale Signaturen, für die Verschlüsselung und für die Vereinbarung gemeinsamer geheimer Schlüssel spezifiziert.

Die Schemata definieren den genauen Ablauf der einzelnen Operationen. Sie benötigen dafür kryptographische Schlüssel, je nach Operation den privaten oder öffentlichen Schlüssel eines Schlüsselpaars. Es gibt drei verschiedene Schlüssel-Typen: DL-, IF- und EC-Schlüssel.

Die im folgenden beschriebene Java-Implementierung beruht auf der letzten Entwurfsversion (D13) des Standards P1363 und wurde zusammen mit Uwe Saliger entwickelt. Bestehende Teilimplementierungen aus einem Programmierpraktikum[1] und der Diplomarbeit von Martin Schulze[2] wurden übernommen, angepaßt und überarbeitet, so daß eine einheitliche, konsistente und vollständige Implementierung des Standards P1363 in Java zur Verfügung steht.

6.1.1. Architektur und Klassen-Design

Der Standard P1363 wurde derart implementiert, daß er als Klassen-Bibliothek auch in andere Anwendungen eingebunden werden kann. Die Gliederung des Standards in *Primitives*, Hilfsfunktionen und Schemata wurde in Form von Java-Paketen (*packages*) nachgebildet. Die Abbildung 6.1 zeigt die Architektur der P1363-Bibliothek. Die Pfeile zeigen die Abhängigkeiten der einzelnen Unterpakete und verstehen sich im Sinne von „verwendet“[3].

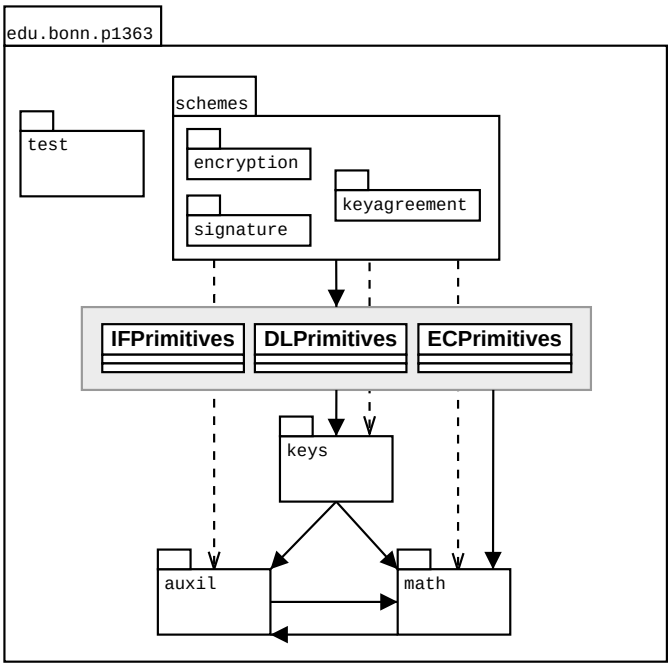

Abbildung 6.1.: Architektur der P1363-Bibliothek

[1] „Moderne kryptographische Verfahren in Java“, Praktikum bei Dr. A. Spalka, Universität Bonn, WS 2000/2001.
[2] Martin Schulze, „Public Key Kryptographie mittels Elliptischer Kurven – eine Java-Implementierung des IEEE P1363“, Diplomarbeit an der Universität Bonn, 2001.
[3] Aus Gründen der Übersichtlichkeit wurden die Pfeile ausgehend von `test` nicht eingezeichnet, ebenso wurden die Pfeile vom Unterpaket `schemes` zu den tiefer liegenden Unterpaketen gestrichelt dargestellt.

So finden sich im Paket `edu.bonn.p1363` die Unterpakete `auxil` und `math` mit Klassen, die die kryptographischen und mathematischen Hilfsfunktionen beinhalten. Basierend auf diesen beiden Paketen wurden alle Schlüsseltypen und zugehörige Schlüsselgeneratoren im Unterpaket `keys` implementiert. Alle *Primitives* wurden ihrem jeweiligen zugrunde liegenden Problem (IF, DL oder EC) entsprechenden Klassen zugeordnet, die sich direkt im Paket `p1363` befinden und auf die Unterpakete `keys` und `math` zugreifen. Alle Unterpakete und die *Primitives* werden von den Schemata verwendet, die sich in `schemes` befinden. Schließlich gibt es das Unterpaket `test`, in dem zahlreiche Testklassen enthalten sind, die die Klassen und Methoden der anderen Unterpakete auf Funktionalität und Geschwindigkeit testen können.

Die einzelnen Unterpakete werden in den folgenden Abschnitten genauer beschrieben. Zugehörige Klassendiagramme finden sich in Anhang B.

Hilfsfunktionen und Schlüsselklassen

Zu den Hilfsfunktionen gehört eine kryptographische Hashfunktion. Der Standard P1363 sieht dafür SHA-1 oder RIPEMD-160 vor. Hier wurde nur SHA-1[4] implementiert. Alle anderen Hilfsfunktionen – Codierungsfunktionen (EME1, EMSA1, EMSA2), Maskierungsfunktion (MGF1), Funktion zum Ableiten von gemeinsamen Schlüsseln (KDF1) und Funktionen zum Konvertieren von Datentypen – wurden vollständig implementiert und befinden sich im Unterpaket `auxil`.

Im Unterpaket `math` befinden sich Klassen zur Implementation der Arithmetik in endlichen Primkörpern und auf elliptischen Kurven, die für DL- bzw. EC-Schlüssel benötigt wird. Die Klassen `FiniteFieldElement` und `ECPoint` wurden aus bestehenden Implementierungen so angepaßt, daß sie „*immutable*“ sind, d.h. der Zustand eines Objektes kann sich nicht ändern. Wenn beispielsweise zwei Körperelemente addiert werden, wird ein neues `FiniteFieldElement`-Objekt erzeugt und als Ergebnis der Rechenoperation zurückgegeben. Dies ist wichtig, damit Schlüssel-Objekte, die auf diesen Arithmetik-Objekten beruhen, nicht im laufenden Betrieb (versehentlich oder absichtlich) geändert werden können.

Die Schlüssel-Klassen enthalten alle ihrem jeweiligen Typ entsprechende `get`-Methoden, die die einzelnen Bestandteile eines Schlüssels, z.B. Körperelemente, liefern, damit die *Primitives* mit diesen Werten rechnen können. Referenzen auf interne Objekte eines Schlüssels werden nur übergeben, wenn diese Objekte „immutable“ sind. Denn solche Objekte können durch die normale API von Java und der JVM nicht geändert werden, so daß ihre Integrität zur Laufzeit gewahrt bleibt. Objekte der Klasse `java.math.BigInteger` sind auch „immutable“ und werden innerhalb der IF-Schlüssel verwendet.

[4] siehe [NIS95]

Um einzelne Schlüssel oder ganze Schlüsselpaare samt ihrer Bereichsparameter[5] in allen Methoden in einer einheitlichen Weise verwenden zu können, arbeiten alle Schemata und *Primitives* mit Objekten der Klasse `KryptoKey` aus dem Unterpaket `keys`. Ein `KryptoKey` besitzt jeweils eine Referenz auf ein Objekt vom Typ `PrivateKey`, `PublicKey` und `DomainParameter`. Falls ein Wert nicht vorhanden ist, ist die Referenz `null`. So besitzt man im allgemeinen von anderen Parteien nur den öffentlichen Schlüssel (ggf. mit Bereichsparametern). `KryptoKey`-Objekte sind ebenfalls „immutable“.

Für jeden Schlüssel-Typ existiert auch ein Schlüsselgenerator. IF-Schlüssel werden zufällig generiert, wobei die zu erhaltende Schlüssellänge in Bits angegeben werden muß. DL- und EC-Schlüssel werden zufällig aus ihren entsprechenden Bereichsparametern gewählt. Bereichsparameter für DL-Schlüssel lassen sich mit der Klassenmethoden `DLDomainParameter.generate()` zufällig generieren, wobei die zu erhaltende Körpergröße und die Größe der verwendeten Untergruppe in Bits angegeben werden müssen. Aus Geschwindigkeitsgründen werden die Bereichsparameter für EC-Schlüssel nicht zufällig neu generiert, sondern aus einem festen Satz ausgewählt. Eine entsprechende Generierungsmethode in der Klasse `ECDomainParameter` ist allerdings vorgesehen und könnte später implementiert werden.

Im Unterpaket `math` befinden sich ferner die beiden abstrakten Klassen `Primetest` für einen Primzahltest und `SecRandom` für einen Zufallszahlengenerator. Prim- und Zufallszahlen werden an vielen Stellen von P1363 benötigt, beispielsweise bei der Schlüsselerzeugung oder bei der Verschlüsselung von Nachrichten. Mehrere konkrete Implementierungen von Primzahltests und Zufallszahlengeneratoren wurden von Uwe Saliger im Rahmen seiner Diplomarbeit[6] entwickelt. Die konkreten Primzahltests befinden sich ebenfalls im Unterpaket `math`, während die Zufallszahlengeneratoren in einem eigenen Paket außerhalb von `p1363` sind.

Signaturschemata

Alle Signaturschemata sind im Unterpaket `schemes.signature` in einer Klassenhierarchie untergebracht, an deren Spitze das Interface `SignatureScheme` ist. Es bietet eine Methode zum Signieren von Daten und eine zum Verifizieren einer Signatur. Im Standard P1363 sind nur Signaturschemata mit Appendix spezifiziert, d.h. die Nachricht selbst kann aus der Signatur nicht zurückgerechnet werden, da die Signatur nur über einen Hashwert der Nachricht gebildet wird und zusammen mit der Nachricht verschickt werden muß.

[5] Die Bereichsparameter (*domain parameter*) eines EC-Schlüssels spezifizieren die zugrunde liegende elliptische Kurve, während die Bereichsparameter von DL-Schlüsseln den endlichen Primkörper definieren. IF-Schlüssel haben keine expliziten Bereichsparameter, da sie auf den natürlichen Zahlen beruhen.

[6] Uwe Saliger, „Sichere Implementierung und Integration kryptographischer Softwarekomponenten am Beispiel der Zufallszahlengenerierung“, Diplomarbeit an der Universität Bonn, 2002.

Für jeden Problem-Typ (IF, DL, EC) gibt es eine eigene abstrakte Klasse, die jeweilige Gemeinsamkeiten innerhalb eines Problem-Typs für die Signaturschemata kapselt. Die konkreten Signaturschemata sind von diesen Klassen abgeleitet und entsprechen genau den Signaturschemata in P1363.

Verschlüsselungsschemata

Die Schemata zur Verschlüsselung sind in einer analogen Klassenhierarchie angeordnet, obwohl in P1363 nur ein einziges Schema spezifiziert wird. Für eine einfachere Wartbarkeit im Falle einer späteren Erweiterung der Bibliothek wurde aber auch hier eine abstrakte Klasse für Verschlüsselung beruhend auf IF definiert. Die Klasse implementiert das Interface `EncryptionScheme`, das Methoden spezifiziert zum Ver- und Entschlüsseln sowie zur Angabe der maximalen Blockgröße einer Nachricht, die auf einmal verarbeitet werden kann. Längere Nachrichten müssen in eine entsprechende Anzahl von Blöcken aufgeteilt werden, und jeder Block wird einzeln mit dem Schema-Objekt verschlüsselt bzw. entschlüsselt. Auf diese Weise können Nachrichten beliebiger Länge bearbeitet werden. Es ist Aufgabe der Anwendung dieser Bibliothek, die Nachricht entsprechend einzuteilen und die Blöcke zu verwalten.

Schemata zur Vereinbarung gemeinsamer geheimer Schlüssel

Im Unterpaket `schemes.keyagreement` befinden sich die Schemata zur Vereinbarung gemeinsamer geheimer Schlüssel. Aus den eigenen privaten Schlüsseln und den öffentlichen Schlüsseln einer anderen Partei lassen sich gemeinsame geheime Werte (*shared secret values*) ableiten. Die andere Partei kann dies ebenfalls, indem sie ihre privaten Schlüssel und die öffentlichen Schlüssel der ersten Partei verwendet. Beide Parteien erhalten den gleichen Wert. Auf diese Weise lassen sich gemeinsame geheime Schlüssel vereinbaren, ohne daß der geheime Schlüssel kommuniziert werden muß. Es muß lediglich vereinbart werden, welche öffentlichen Schlüssel zur Anwendung kommen sollen. Mit dem geheimen Schlüssel können dann Nachrichten symmetrisch verschlüsselt werden, was oftmals deutlich schneller möglich ist als asymmetrische Verschlüsselung mit Public-Key-Verfahren.

Der Standard P1363 spezifiziert entsprechende Schemata für DL- und EC-Schlüssel. Im Schema `DLECKAS_DH1` wird pro Partei ein Schlüssel benötigt, wobei beide Schlüssel die gleichen Bereichsparameter haben müssen. Das Schema `DLECKAS_MQV` benötigt zwei Schlüssel pro Partei, wobei alle Schlüssel die gleichen Bereichsparameter haben müssen. Im Schema `DLECKAS_DH2` werden zwar ebenfalls zwei Schlüssel pro Partei verwendet, jedoch müssen nur jeweils zwei Schlüssel die gleichen Bereichsparameter besitzen, nämlich jeweils die korrespondierenden Schlüssel der ersten und der zweiten Partei. So können DL- und EC-Schlüssel

in diesem Schema gleichzeitig verwendet werden.

Wenn als Eingabeparameter immer die gleichen Schlüssel gewählt werden, berechnen die Schemata die gleichen gemeinsamen geheimen Werte. Um daraus aber neue, unterschiedliche geheime Schlüssel bilden zu können, bekommen die Schemata noch *Key Derivation Parameter* zugewiesen, die bei der Generierung des geheimen Schlüssels mit dem gemeinsamen geheimen Wert einer Hashfunktion übergeben werden. Durch die Änderung der *Key Derivation Parameter* erhält man unterschiedliche geheime Schlüssel.

6.1.2. Konformität zu P1363 Anhang D (*Security Considerations*)

Alle Schemata-Klassen erhalten bei der Instanziierung einen Schlüssel in Form eines `KryptoKey`-Objekts, mit dem die jeweiligen Operationen des Schemas durchgeführt werden. Je nach Operation wird der private oder der öffentliche Schlüssel aus dem `KryptoKey` benötigt. Die Auswahl der Schlüssel ist Aufgabe des Anwenders. Manche Schemata-Klassen benötigen zusätzlich einen Zufallszahlengenerator, um ihre Operationen durchführen zu können. Außerdem erhalten alle Schemata-Klassen einen Primzahltest, um die Schlüssel validieren zu können.

Der Anhang D von P1363 empfiehlt, daß bei der Anwendung der Schemata die Schlüssel und Bereichsparameter validiert werden, damit bestimmte kryptographische Angriffe ausgeschlossen werden können. Die Schemata-Klassen wurden folglich so entwickelt, daß bei ihrer Instanziierung automatisch die übergebenen Schlüssel und Bereichsparameter validiert werden. Sind die Schlüssel oder Bereichsparameter ungültig, wird das jeweilige Schema-Objekt erst gar nicht erzeugt. Auf diese Weise wird verhindert, daß Operationen mit ungültigen Schlüsseln ausgeführt werden können.

Zum Teil mußten die Funktionen zur Validierung der Schlüssel und der Bereichsparameter neu entwickelt werden, da sie in den bestehenden Teilimplementierungen nicht vorhanden waren. Insgesamt wurden alle implementierten Funktionen überprüft und zum Teil reimplementiert, damit sie standardkonform arbeiten.

Der `KryptoKey` für ein Schema-Objekt wird außerdem nur bei der Instanziierung übergeben und kann für die Lebensdauer des Schema-Objekts nicht mehr geändert werden. Das bedeutet, daß für jede Operation, die mit einem anderen Schlüssel durchgeführt werden soll, ein eigenes Schema-Objekt erzeugt werden muß. So wird die Konsistenz in einer Anwendung bewahrt. Ansonsten könnte beispielsweise der Schlüssel zum Signieren geändert werden, nachdem ein Signaturschema-Objekt erzeugt und bevor die tatsächliche Signaturoperation durchgeführt wurde.

In Anhang D des Standards P1363 wird empfohlen, bei der Generierung von DL-Bereichsparametern sowohl die Größe des Primkörpers als auch der Untergruppenordnung variabel zu

wählen. In einer bestehenden Teilimplementierung wurde die Untergruppe immer konstant mit einer Ordnung von 160 Bits Länge berechnet, da dies bei Signaturschemata üblich ist und der Länge der Ausgabe der Hashfunktion entspricht. Die Generierungsfunktion für DL-Bereichsparameter wurde neu implementiert, so daß die Untergruppenordnung variabel gewählt werden kann, wie es auch der Standard vorsieht.

Laut [IEE99, Anhang D.5.2.2] gilt für die Untergruppenordnung r:

- $|r| < 160$ Bits: Die Sicherheit der Codierungsfunktion EMSA1 kann beeinträchtigt werden.
- $|r| > 160$ Bits: Der Repräsentant einer Nachricht ist nicht mehr eine zufällige Zahl zwischen $[0..r-1]$.

Letzteres stelle laut Anhang D jedoch kein bekanntes Sicherheitsrisiko dar und schränke die Funktionalität nicht ein. So empfiehlt der Standard [IEE99, Anhang D.4.1.4] für DL-Bereichsparameter:

- die Länge der Primkörperordnung analog zu RSA-Schlüsseln zu wählen, und
- die Länge der Untergruppenordnung analog zu EC-Schlüsseln zu wählen.

Das bedeutet für den Anwender, daß er die Primkörperordnung in Größenordnungen von 1024, 2048, 4096,... Bits wählt und die Untergruppenordnung in Größenordnungen von 160, 190, 240, 360,... Bits. Die Korrelation ergibt sich aus den Schwierigkeitsgraden, wielange es dauert, einen RSA-Schlüssel und einen EC-Schlüssel zu „knacken“[7]. Die Schwierigkeitsgrade sollten gleich hoch sein.

Allgemein wurde die P1363 Bibliothek dementsprechend entwickelt, daß der Anwender seine Schlüssel und Bereichsparameter selbst generiert und für ihre sichere Speicherung verantwortlich ist. Die Verwendung von Zertifikaten und schlüsselverwaltenden oder -ausstellenden Autoritäten wurde nicht implementiert. Nach Anhang D sollten private Schlüssel niemals weitergegeben werden. Die hier implementierte Bibliothek enthält selbst keine Vorkehrungen, dies zu verhindern. Dies ist Aufgabe der Anwendung.

Der Anhang D empfiehlt weiter, daß Schlüssel und Bereichsparameter nur eine beschränkte Lebenszeit haben sollten, nach deren Ablauf sie ihre Gültigkeit verlieren, und daß Schlüssel nicht für mehrere Schemata gleichzeitig verwendet werden sollten. Entsprechende Überprüfungen wurden innerhalb der Bibliothek nicht implementiert, sondern müssen durch die Anwendung erfolgen.

[7] Damit ist gemeint, aus dem öffentlichen Schlüssel den privaten Schlüssel zu berechnen.

6.1.3. Integration in Windows 2000

Zur Demonstration der P1363 Klassen-Bibliothek wurde von Uwe Saliger eine graphische Oberfläche entwickelt, die zusammen mit anderen Komponenten die kryptographische Anwendung *Truesecs* ergibt, mit der sich Dateien verschlüsseln und signieren sowie entschlüsseln und Signaturen prüfen lassen. Die Gesamtarchitektur der Anwendung ist in Abbildung 6.2 dargestellt.

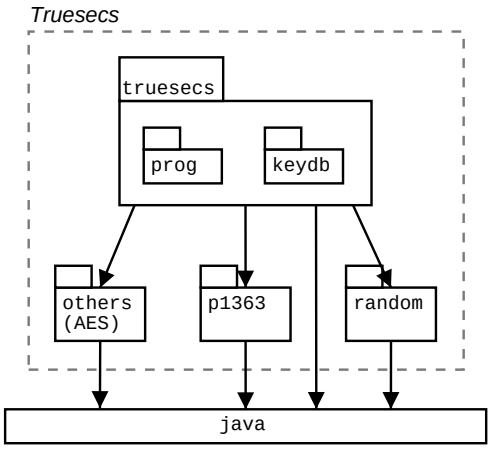

Abbildung 6.2.: Architektur der kryptographischen Anwendung *Truesecs*.

Truesecs besteht im wesentlichen aus zwei Komponenten: die eigentliche Programmkomponente zur Steuerung und Darstellung der graphischen Oberfläche und eine Komponente zur Verwaltung der kryptographischen Schlüssel des Benutzers. Alle Schlüssel eines Benutzers werden in einer Datei, der Schlüsseldatenbank, gespeichert, die im *Home*-Verzeichnis des jeweiligen Benutzers angelegt wird. Die Schlüsseldatenbank wird von *Truesecs* symmetrisch mit Hilfe des Verfahrens AES[8] verschlüsselt, wobei der symmetrische Schlüssel ein Paßwort ist, das der Benutzer beim Programmstart von *Truesecs* eingeben muß.

Neben den kryptographischen Operationen bietet *Truesecs* auch eine einfache Schlüsselverwaltung. Dem Benutzer ist es möglich, Schlüssel zu importieren und zu exportieren. Private Schlüssel können nicht aus der Anwendung heraus exportiert werden. Die Schlüssel können in *Truesecs* generiert werden und erhalten dabei vom Benutzer eine Gültigkeitsdauer. Diese dient jedoch nur als Hinweis für den Benutzer, wenn ein Schlüssel abgelaufen ist. Die Anwendung selbst verweigert die Arbeit mit abgelaufenen Schlüsseln nicht, damit verschlüsselte Dokumente noch gelesen werden können, wenn der zugehörige Schlüssel bereits abgelaufen ist.

In *Truesecs* können grundsätzlich dieselben Schlüssel in verschiedenen Schemata verwendet werden. Der Benutzer kann jedoch mehrere eigene Schlüsselpaare generieren und sie durch

[8] siehe [NIS01]

geeignete Namen verschiedenen Aufgaben zuordnen. Die Einhaltung dieser Zuordnung liegt in der Verantwortung des Benutzers und wird nicht programmatisch kontrolliert[9].

Truesecs verwendet neben der P1363 Bibliothek und AES noch eine Komponente mit Zufallszahlengeneratoren. Alle Komponenten verwenden schließlich die Java Standardklassen. Die gesamte Anwendung wurde soweit angepaßt, daß sie auch aus einer JAR-Datei heraus lauffähig ist, d.h. alle Komponenten von *Truesecs* können sich in einer JAR-Datei befinden, die wie in Kapitel 4 beschrieben signiert werden kann.

Für die robuste Integration in Windows 2000 bedeutet dies, daß neben den Komponenten der JVM nur noch diese eine JAR-Datei vom *ModuleStamper* überprüft werden muß. Die Integrität und Vertraulichkeit der Schlüsseldatenbank wird durch die Verschlüsselung mittels AES gewährleistet. Ohne Kenntnis des Paßwortes, mit dem die Datenbankdatei verschlüsselt wurde, können die Schlüssel nicht gelesen werden. Eine Manipulation der Datei würde dazu führen, daß die Entschlüsselung fehlerhafte Daten generieren würde, was der Anwendung auffiele.

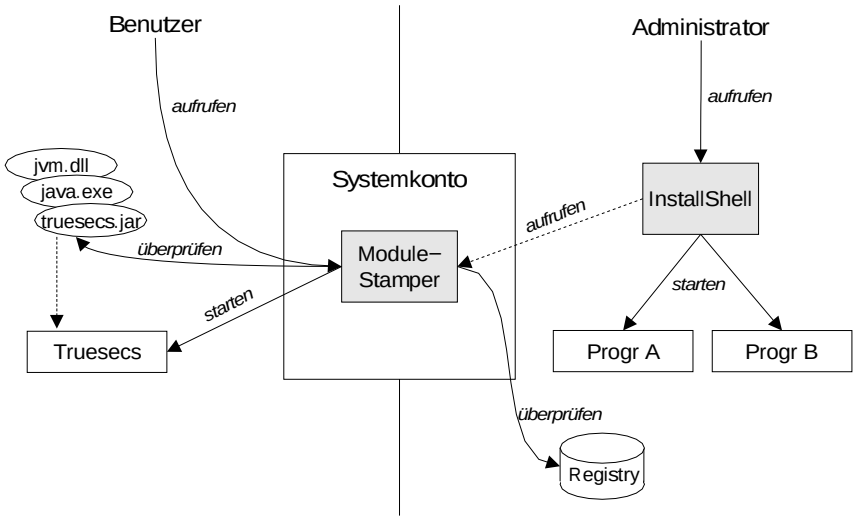

Abbildung 6.3.: Integration von *Truesecs* in Windows 2000

Die Anwendung *Truesecs* kann über den *ModuleStamper*-Dienst aus der JAR-Datei gestartet werden, damit dieser die Integritätsprüfung der verwendeten Module durchführt. Gibt der *ModuleStamper* keine Warnmeldung einer Integritätsverletzung aus, kann der Benutzer mit der Anwendung arbeiten. Damit der *ModuleStamper*-Dienst nicht manipuliert werden kann, darf der Administrator nichtvertrauenswürdige Programme nur in der beschränkten Installationsshell ausführen. Andernfalls wären Manipulationen am Dienst oder an der Anwendung *Truesecs* möglich, die unerkannt bleiben.

[9] Technisch wäre es jedoch kein Problem eine derartige kontrollierte Zuordnung zu implementieren. Aus Gründen der Einfachheit der Anwendung für potentielle Benutzer wurde jedoch darauf verzichtet.

6.2. Exklusive Kommunikation von Programmen

Manchmal ist es wünschenswert, daß zwei Programme exklusiv miteinander kommunizieren können: Funktionen von Programm A sollen nur von Programm B aufgerufen werden können und umgekehrt. Ein Aufruf von einem Programm C soll ebensowenig möglich sein, wie das Schicken einer Nachricht an C.

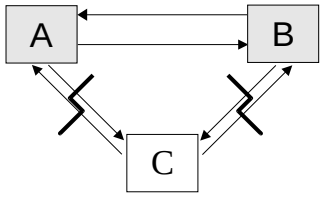

Abbildung 6.4.: Exklusive Kommunikation zwischen zwei Programmen

Beispielsweise könnte A eine Textverarbeitung sein und B der Treiber für eine am Computer angeschlossene Smartcard, die Texte kryptographisch signieren soll. Es ist wichtig, daß in A geschriebene Texte, die signiert werden sollen, direkt an B geschickt werden und nicht an ein Programm C. Es könnte sich bei C um einen Trojaner handeln, der sich gegenüber A als Treiber der Smartcard ausgibt, die Texte modifiziert und dann erst an B schickt. Es würde auf diese Weise ein anderer Text signiert, als der Benutzer beabsichtigte. Wenn der Text zusätzlich von der Smartcard mit einem Public-Key-Verfahren verschlüsselt wird, kann der Benutzer die Manipulation anschließend nicht mehr erkennen.

6.2.1. Architektur einer exklusiven Kommunikation

Die bisher gezeigten Konzepte des *ModuleStamper*-Dienstes und der beschränkten Installationsshell können dafür sorgen, daß für jedes sicherheitskritische Programm die Integrität der jeweils verwendeten Module geprüft werden kann. Um eine exklusive Kommunikation durchführen zu können, müssen sich die beteiligten Programme gegenseitig identifizieren.

Eine gegenseitige Identifizierung könnte erreicht werden, wenn die Kommunikation zwischen den Programmen asymmetrisch verschlüsselt und signiert wird. Jedes Programm besitzt dazu sein eigenes Schlüsselpaar, das bei jedem Programmstart neu generiert wird. Die Haltbarkeit, d.h. die kryptographische Stärke der Schlüssel muß der „Lebenszeit“ der Nachrichtenpakete, die zwischen den Programmen ausgetauscht werden, angepaßt werden. Eine Haltbarkeit bis zu einem Tag dürfte in den meisten Fällen ausreichen. Dementsprechend kann die Schlüssellänge gewählt werden.

Die Programme müssen bei ihrer Initialisierung ihre öffentlichen Schlüssel austauschen. Dies ließe sich dadurch realisieren, indem zu jedem Programm eine Public-Key-Datei existiert,

in die es seinen öffentlichen Schlüssel schreibt. Die anderen Programme können aus dieser Datei den öffentlichen Schlüssel einlesen, ihre Nachrichten an das Programm damit verschlüsseln und mit ihrem eigenen privaten Schlüssel signieren.

Damit die Programme der exklusiven Kommunikation wissen, in welche Datei sie ihren öffentlichen Schlüssel schreiben sollen und wo sich die Public-Key-Dateien der anderen Programme befinden, stellt der Benutzer in den betreffenden Programmen die Pfadangaben der Dateien ein. Es bleibt jedoch die Frage, wie sich der Austausch der öffentlichen Schlüssel schützen läßt, so daß ein Trojaner die Public-Key-Dateien nicht verändern kann.

Die Public-Key-Dateien müssen dazu bei der Installation der betreffenden Programme erstellt und so konfiguriert werden, daß nur die Programme der exklusiven Kommunikation einen Schreibzugriff darauf haben. Alle anderen Programme dürfen die Dateien nur lesen können.

Abbildung 6.5 veranschaulicht die Architektur der exklusiven Kommunikation zweier Programme A und B. Die beiden Programme haben allein Schreibzugriff auf ihre Public-Key-Dateien, alle anderen Programme können sie nur lesen.

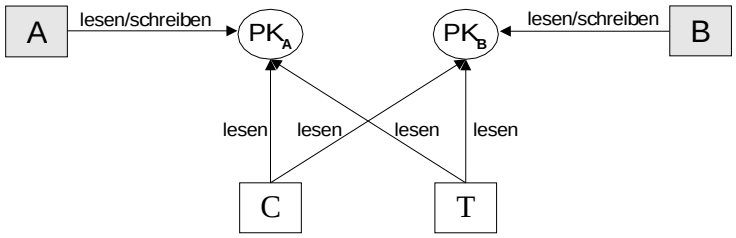

Abbildung 6.5.: Architektur einer exklusiven Kommunikation

Die Public-Key-Dateien sollten niemals gelöscht werden, denn sonst könnte ein Trojaner eine Datei neu anlegen, so daß er Schreibzugriff darauf hat. Damit beim Programmstart nicht die alten Schlüssel der letzten Sitzung verwendet werden, muß eine geeignete Synchronisation in den Programmen implementiert werden. Beispielsweise könnte eine konstante Testnachricht geschickt werden, die vom anderen Programm in festgelegter Weise verändert und zurückgeschickt werden muß. Wenn die Antwort nicht dem erwarteten Ergebnis entspricht, ist noch der alte Schlüssel in der Datei vorhanden, und das eine Programm muß warten, bis das andere seinen neuen Schlüssel erzeugt und in die Datei geschrieben hat.

6.2.2. Implementation in Java und Windows 2000

Zur Demonstration einer exklusiven Kommunikation wurden zwei kleine Java-Programme implementiert, die sich abwechselnd gegenseitig kurze Textnachrichten schicken. Jedes Programm läuft in einer eigenständigen JVM. Die Nachrichten werden als Argumente an Methoden über-

geben, die mittels RMI aufgerufen werden. Zur Verschlüsselung und Signierung der Nachrichten wird die zuvor beschriebene P1363 Bibliothek verwendet. Wenn eines der Programme eine falsch signierte Nachricht erhält, gibt es eine Fehlermeldung aus und beendet sich.

Um die Architektur zur exklusiven Kommunikation unter Windows 2000 zu implementieren, dürfen nur diese beiden Programme, genauer ihre beiden JVM-Prozesse, Schreibzugriff auf die Public-Key-Dateien haben. Schreibzugriffe auf Dateien werden in ihren zugehörigen ACLs geregelt. Dort werden Zugriffsrechte einzelnen Benutzer- oder Gruppen-SIDs zugeordnet, da die Rechtezuordnung in Windows 2000 benutzerbasiert und nicht programmbasiert ist. Alle Programme, die von einem Benutzer gestartet wurden, erhalten eine Kopie seines Zugriffstokens, in dem seine SID steht.

Um die Zugriffsrechte differenzierter vergeben zu können, benötigt jeder Benutzer zwei Benutzerkonten: ein Arbeitskonto (AK) und ein Verwaltungskonto (VK). Die ACLs der Public-Key-Dateien werden so definiert, daß nur Prozesse mit einer VK-SID die Dateien schreiben und lesen dürfen. Für alle anderen SIDs, insbesondere für die AK-SID, werden lediglich Leserechte eingeräumt. Der Benutzer meldet sich im Verwaltungskonto an, um die Zugriffsrechte für diese Dateien zu konfigurieren. Ansonsten meldet er sich im Arbeitskonto an und startet dort die Programme, mit denen er arbeiten will.

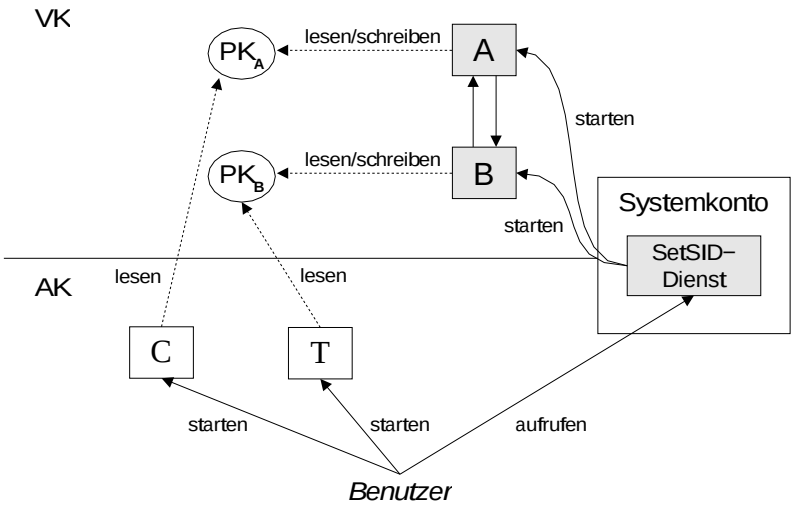

Abbildung 6.6.: Exklusive Kommunikation unter Windows 2000

Alle vom Benutzer gestarteten Programme bekommen somit eine AK-SID. Damit die beiden Programme der exklusiven Kommunikation ihre öffentlichen Schlüssel in die Public-Key-Dateien schreiben können, benötigen sie jedoch eine VK-SID. Prozesse können ihre zugewiesene SID nicht ändern, und der normale Benutzer kann keine anderen SIDs an Prozesse vergeben.

Das Systemkonto besitzt das Privileg zur Erzeugung von Tokens. Ein spezieller Windows

2000 Dienst, der im Systemkonto ausgeführt wird, könnte die beiden Programme der exklusiven Kommunikation in Prozessen starten, die in ihrem Token eine VK-SID haben. Der Benutzer, der im Arbeitskonto angemeldet ist und eine AK-SID besitzt, könnte den Dienst aufrufen, damit er die entsprechenden Programme mit einer VK-SID startet. Abbildung 6.6 veranschaulicht dies.

Auf diese Weise können einzig die beiden Programme der exklusiven Kommunikation in die Public-Key-Dateien schreiben. Weil alle anderen Programme vom Benutzer mit einer AK-SID gestartet werden, könnte ein darunter befindlicher Trojaner diese Dateien nicht verändern. Er könnte auch nicht die ACLs der Public-Key-Dateien zu seinen Gunsten ändern, da dies ausschließlich dem Besitzer der Dateien erlaubt ist, in dem Fall dem Verwaltungskonto.

Entwicklung eines SetSID-Dienstes

Damit man Programme der exklusiven Kommunikation automatisch mit ihrer VK-SID starten kann, ist die Entwicklung eines „SetSID“-Dienstes nötig, der einen entsprechenden Token erstellt. Normalerweise ist die Token-Erstellung nur in Verbindung mit einer Anmeldung mit Kontoname und Kennwort über die Funktion `LogonUser()` möglich. Diese Funktion benötigt das Recht *SeTcbPrivilege*. Der Lokale Sicherheits-Authentifizierungsserver (LSA), der im Prozeß *Lsass* läuft, erstellt beim Aufruf dieser Funktion und bei korrekten Anmeldedaten einen entsprechenden Token.

Diese Vorgehensweise würde bedeuten, daß der SetSID-Dienst zum Starten von Programmen mit einer spezifischen SID das zugehörige Kennwort benötigt. Wünschenswert wäre eine direkte Token-Erstellung mit der SID, ohne daß das Kennwort angegeben werden muß. Die dafür erforderliche Funktion `CreateToken()` ist jedoch nicht öffentlich zugänglich. Sie ist innerhalb von LSA implementiert, und ihre Adresse, um sie aufrufen zu können, bekommen ausschließlich sogenannte *Security Support Packages / Authentication Packages* (SSP/APs)[10]. Dies sind spezielle DLLs, die einen spezifischen Satz von Funktionen implementieren und vom Administrator in der Registry registriert werden müssen. LSA verwendet diese SSP/APs zur Überprüfung der Anmeldedaten.

Es besteht jedoch die Möglichkeit, eigene SSP/APs zu erstellen. Ein eigenes SSP/AP, das bei LSA registriert ist, kann ein Token direkt anhand einer SID erstellen, ohne ein Kennwort zu benötigen. Mit der Funktion `LsaLogonUser()` kann man eine Anmeldung zur Token-Erstellung bei LSA durchführen lassen und gleichzeitig angeben, welche SSP/AP verwendet werden soll. Der SetSID-Dienst würde diese Funktion aufrufen, um LSA anzuweisen, die eigene SSP/AP zu verwenden. Die eigene SSP/AP-DLL würde die Anmeldung unabhängig von einem Kennwort akzeptieren und damit LSA veranlassen, einen entsprechenden Token zu

[10] siehe [MS01g]

erzeugen. Der SetSID-Dienst würde daraufhin von LSA ein Handle auf den neuen Token bekommen, mit dem er das Programm der exklusiven Kommunikation starten kann.

Damit nicht beliebige Anwendungen spezielle SSP/APs verwenden können, müssen sich anfordernde Prozesse bei LSA mit der Funktion `LsaRegisterLogonProcess()` registrieren, wozu das Recht *SeTcbPrivilege* erforderlich ist. Somit können nur Prozesse, die im Systemkonto ausgeführt werden, sich bei LSA anmelden, um eine spezielle SSP/AP zu verwenden. Abbildung 6.7 zeigt die gesamte Aufrufsequenz für einen SetSID-Dienst.

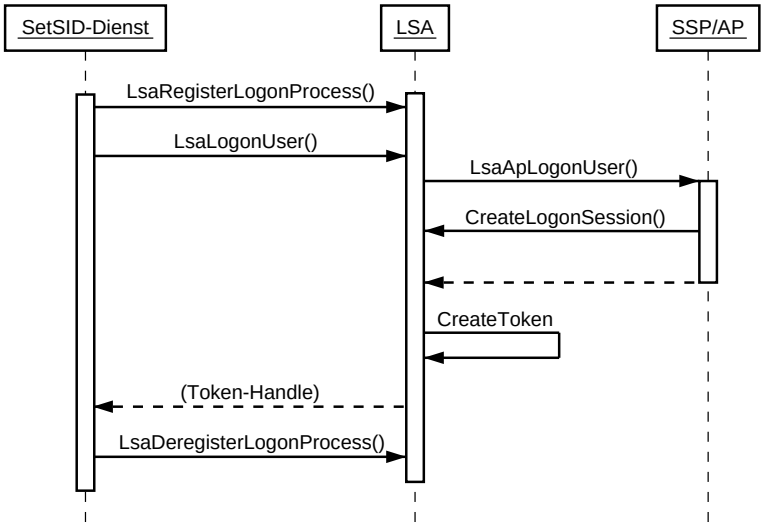

Abbildung 6.7.: Verwendung eines eigenen SSP/AP zur Tokenerstellung

Da der *ModuleStamper* als Dienst im Systemkonto ausgeführt wird, besitzt er die Berechtigung, spezielle SSP/APs verwenden zu dürfen. Die Funktionalität zum Starten von Programmen ist schon integriert, sie läßt sich dahingehend erweitern, daß der *ModuleStamper* auch die Aufgabe eines SetSID-Dienstes übernehmen kann. Beim Start von Programmen der exklusiven Kommunikation kann er so gleichzeitig ihre verwendeten Module auf Integrität überprüfen.

Damit nicht beliebige Programme mit VK-Rechten gestartet werden können, muß der Dienst explizit wissen, für welche Programme er einen Token mit VK-SID anfordern soll. Ausschließlich der Administrator sollte in der Lage sein, diese Informationen zu definieren. Folglich werden die entsprechenden Einstellungen über das Konfigurationsprogramm des *ModuleStamper*-Dienstes vorgenommen und in der zentralen Stampdatei des Dienstes gespeichert. Auf die Stampdatei hat einzig das Systemkonto Zugriff, so daß von normalen Benutzerkonten ausgeführte Trojaner die Einstellungen nicht verändern können.

Zum Starten von Programmen mit spezieller SID erhält der *ModuleStamper*-Client einen zusätzlichen Schalter „setsid“, der bestimmt, daß das als Argument übergebene Programm mit der SID gestartet werden soll, die vom Administrator festgelegt wurde. Ein Aufruf des Clients

sähe beispielsweise wie folgt aus:

```
ModuleStamper -start -setsid X:\Ver\zeich\nis\Programm.exe
```

Im Gegensatz zum normalen Starten von Programmen mit dem *ModuleStamper* wird in diesem Fall der neue Prozeß nicht innerhalb des Clients erzeugt, sondern im Dienst-Prozeß. Der Client wird im normalen Benutzerkonto ausgeführt und kann sich nicht bei LSA anmelden. Statt dessen schickt der Client den Namen der Programmdatei an den Dienst-Prozeß. Der Dienst registriert sich, wie zuvor beschrieben, bei LSA und fordert einen Token mit Hilfe der eigenen SSP/AP an.

Bevor der neue Prozeß mit dem Token gestartet werden kann, sind einige Dinge zu beachten. Der Dienst muß zuvor die Sicherheitsbeschreibung der interaktiven Arbeitsstation („*WinSta0\Default*“) so ändern, daß der neue Prozeß darauf Zugriff hat. Jeder Token besitzt neben der Benutzer-SID eine *Logon-Session-SID*[11], die einmalig für jeden Anmeldevorgang ist und der interaktiven Arbeitsstation erst bekannt gemacht werden muß. Andernfalls kann der neue Prozeß zwar gestartet werden, aber nicht mit dem Benutzer interaktiv kommunizieren.

Weiterhin muß das Benutzerprofil in die *Registry* geladen werden, falls der neue Prozeß daraus Einstellungen einlesen will. Der *ModuleStamper*-Dienst verwaltet dazu eine Liste für jede SID, ob das zugehörige Profil schon geladen wurde oder nicht. Wenn für eine SID erstmalig ein Prozeß gestartet werden soll, wird das zugehörige Benutzerprofil vom Dienst geladen. Wurden alle Prozesse mit einer bestimmten SID beendet, entlädt der Dienst das Benutzerprofil wieder. Zur Überprüfung, ob ein gestarteter Prozeß noch läuft, wird für jeden Prozeß ein eigener Thread im Dienst gestartet, der den neuen Prozeß überwacht. Auf diese Weise wird der Dienst nicht blockiert.

Die Programmdateien, die für einen Start mit SetSID vorgesehen werden, müssen in ihren Sicherheitsbeschreibungen entsprechende ACLs enthalten, so daß normale Benutzerkonten die Dateien nicht austauschen oder verändern können. Ein Trojaner könnte sonst mit VK-Rechten gestartet werden. Andererseits würde der *ModuleStamper* eine Veränderung der Programmdatei erkennen, wenn sie zu seinen zu überprüfenden Programmen gehört.

Zur Demonstration des SetSID-Dienstes wurde zur Vereinfachung der Prototyp des *ModuleStampers* ohne eine eigene SSP/AP-DLL implementiert. Statt dessen verwendet der *ModuleStamper* die Funktion `LsaLogonUser()` mit dem Standard-SSP/AP. Hierzu ist jedoch immer der Kontoname *und* das Kennwort erforderlich. Damit dem Dienst das Kennwort zur Verfügung steht, wird es bei der Konfiguration der SetSID-Einstellungen vom Administrator mit angegeben und in der Stampdatei gespeichert.

Dies ist ein analoges Vorgehen zur allgemeinen Konfiguration von Diensten: Soll ein Dienst in einem anderen Konto als dem Systemkonto laufen, muß bei der Konfiguration des Dienstes Kontoname und Kennwort vom

[11] siehe [Bro00]

Administrator angegeben werden. Diese Informationen werden dann vom SCM in der *Registry* gespeichert und zum Starten des Dienstes wieder gelesen. Der Zugriff auf den entsprechenden *Registry*-Schlüssel wird auf das Systemkonto beschränkt.

Diskussion

Damit die exklusive Kommunikation eingehalten werden kann, muß das Konzept der Installationsshell für Administratoren konsequent angewendet werden. Der *ModuleStamper*-Dienst muß als Integritätsprüfer installiert sein und laufen. Normale Benutzer dürfen nicht zusätzliche Windows 2000 Privilegien besitzen, wie beispielsweise das Recht zum Laden von Gerätetreibern, denn andernfalls könnten Trojaner-Treiber im Kernelmodus die Programme der exklusiven Kommunikation oder die Public-Key-Datei manipulieren.

Die Installation zusätzlicher Dienste muß als Warnung gemeldet werden, denn Prozesse im Systemkonto haben weitgehende Rechte, sie können z.B. Prozesse mit Hilfe des Debug-Privilegs manipulieren und haben durch das Inbesitznahmerecht praktisch jeden Zugriff auf alle Dateien. Würde es auf diese Weise zu einer Integritätsverletzung kommen, könnte die exklusive Kommunikation von Programmen nicht mehr gewährleistet werden.

Durch die Benutzung der Installationsshell und des *ModuleStamper*-Dienstes können neu installierte Dienste oder Treiber erkannt werden. Abgesehen davon können normale Benutzer standardmäßig keine Dienste oder Treiber installieren.

Die hier vorgestellte Architektur der exklusiven Kommunikation hat dennoch eine Schwachstelle: Ein Programm mit VK-SID hat auch Schreibrecht auf die Public-Key-Datei eines anderen Programms der exklusiven Kommunikation. So gibt es keinen Schutz vor gegenseitigen Angriffen der VK-Programme.

Wenn eine derartige Bedrohung nicht ausgeschlossen werden kann, läßt sich die Architektur allerdings dahingehend erweitern, daß eine *disjunkte* exklusive Kommunikation zwischen Programmen etabliert werden kann. Dazu benötigt ein Benutzer mehrere verschiedene VK-Konten, eins für jedes Programm der disjunkten exklusiven Kommunikation. Der Verwaltungsaufwand würde mit der Anzahl der exklusiv kommunizierenden Programme linear ansteigen.

Nachteilig könnte sich zudem auswirken, daß Prozesse mit VK-SID standardmäßig keinen Zugriff auf Dateien haben, die zum AK gehören. Der Benutzer muß folglich für die Dateien, auf die die Programme der exklusiven Kommunikation zugreifen sollen, die ACLs so konfigurieren, daß auch das VK darauf Zugriff hat, was wiederum den Verwaltungsaufwand ansteigen läßt.

7. Zusammenfassung und Ausblick

Kryptographische Anwendungen werden vor allem auf Computersystemen eingesetzt, um Nachrichten vertraulich und unverändert mit anderen Systemen austauschen zu können. Die Stärke der kryptographischen Algorithmen und der verwendeten Schlüssel soll Angriffe auf die Übermittlung der Nachricht verhindern, die ihre Integrität oder Vertraulichkeit verletzen. Wenn ein Angreifer jedoch Zugang zu dem Computersystem hat, auf dem sich die Anwendung befindet, genügt die kryptographische Stärke nicht mehr. Das Software-Design und die Integration ins Betriebssystem muß robust gegenüber Manipulationsversuchen sein und verhindern, daß ein Angreifer vertrauliche Daten wie private Schlüssel auslesen kann.

Java bietet durch seine Typsicherheit und den Verzicht auf Zeigerarithmetik dem Programmentwickler eine Hilfe zur Vermeidung von Sicherheitslücken, die durch Programmierfehler entstehen könnten. Durch die Zuordnung von *Permissions* kann der Zugriff auf Ressourcen beschränkt und die Schnittstellensemantik von Java-Programmen bewahrt werden, wenn neue Komponenten zu der Anwendung hinzukommen. Um zu verhindern, daß Trojaner-Komponenten in die Anwendung gelangen, lassen sich Java-Programme in signierten JAR-Dateien speichern, deren Signatur beim Laden überprüft wird.

Für den Schutz der *Policy*-Datei, in der die Zuordnung der *Permissions* steht, und der *Keystore*-Datei, in der die Schlüssel zur Verifikation der Signaturen stehen, zeichnet Java jedoch das Betriebssystem verantwortlich. Mit Hilfe eines Integritätsprüfprogramms können diese Dateien wie auch alle anderen Module, die eine Anwendung benötigt, basierend auf kryptographischen Hashwerten überprüft werden. Um eine automatische Überprüfung durchführen zu können und um die gespeicherten Referenzwerte zu schützen, wurde ein Windows 2000 Dienst entwickelt.

Es wurde gezeigt, daß ein Integritätsprüfprogramm allein nicht ausreicht, um sicherheitskritische Anwendungen vor Manipulationen zu schützen. Es gibt viele Möglichkeiten für einen Angreifer, vor allem da bei Einzelplatzsystem häufig der Anwender auch Administrator ist und Programme unter diesem Konto installiert werden. Für die robuste Integration einer sicherheitskritischen Anwendung in Windows 2000 bedarf es für den Administrator einer beschränkten Ausführungsumgebung, in der zusätzliche Programme installiert und ausgeführt werden kön-

nen, da sie aus nichtvertrauenswürdigen Quellen stammen könnten.

Die in dieser Arbeit vorgestellte *Installationsshell* liefert in Zusammenspiel mit dem *ModuleStamper*-Dienst eine Möglichkeit, nichtvertrauenswürdige Programme im Administratorkonto auszuführen und etwaige Manipulationen an Betriebssystem- oder Anwendungskomponenten erkennen zu können. Der *ModuleStamper*-Dienst prüft immer beim Shutdown des Betriebssystems und nach Installationen sowohl die *Registry* als auch wichtige Programmodule. Die Installationsshell startet Prozesse mit einem beschränkten Token und deaktiviert temporär die Dienste *SecLogon* und *Schedule*, damit sie nicht für einen Angriff mißbraucht werden können.

Diese Lösung hat den Vorteil, daß bestehende Zuordnungen von Privilegien an Benutzerkonten nicht geändert werden müssen. Es sind keine zusätzlichen Konfigurationsarbeiten notwendig, außer das Installieren des *ModuleStamper*-Dienstes und der Installationsshell sowie das erstmalige „Stampen“ zum Bilden der Referenzwerte. Das temporäre Deaktivieren des *SecLogon*-Dienstes dürfte zu keinen Problemen führen, da Programme sich normalerweise nicht automatisch in andere Benutzerkonten einloggen. Einzig die Deaktivierung des *Schedule*-Dienstes während einer Programminstallation könnte als nachteilig aufgefaßt werden. Allerdings könnten entsprechende Eintragungen nach der Installation manuell gemacht werden, so daß die Funktionalität nicht eingeschränkt wird.

Letzten Endes funktioniert das gesamte Konzept nicht, wenn die Installationsshell nicht konsequent verwendet wird. Hier sind zusätzlich organisatorische Maßnahmen erforderlich, daß Administratoren nichtvertrauenswürdige Programme nur in dieser Shell installieren und ausführen.

Es wurde eine Architektur vorgestellt, um unter Windows 2000 eine exklusive Kommunikation von Programmen zu ermöglichen. Der ModuleStamper wurde dazu um einen SetSID-Dienst erweitert. Die Integration des *ModuleStamper*-Dienstes, des *ModuleStamper*-Clients, des SetSID-Dienstes und des Konfigurationsprogramms in eine ausführbare Programmdatei hält die Anzahl der zu schützenden Systemkomponenten klein.

Die hier vorgestellten Konzepte können allerdings noch verbessert und erweitert werden. So können mit Hilfe des *ModuleStampers* und der Installationsshell zwar Manipulationen von Programmen erkannt werden, jedoch wird die Ausführung nicht verhindert. Der Anwender wird lediglich auf die Manipulation aufmerksam gemacht.

Man könnte die Installationsshell dahingehend erweitern, daß sie die standardmäßige *Explorer*-Shell komplett ersetzt und die Ausführung von Programmen verweigert, die vom *ModuleStamper* als manipuliert gemeldet werden. Die Shell müßte zwei verschiedene Modi anbieten: ein Modus, in dem normal gearbeitet werden kann, und ein Modus, in dem nichtvertrauenswürdige Programme ausgeführt werden. Für letzteres müssen die Dienste *SecLogon* und *Schedule*

deaktiviert und dem Zugriffstoken einige Privilegien entzogen werden. Im Normalfall könnten die Dienste genutzt werden. Es wäre zudem zu untersuchen, inwiefern von der Shell verhindert werden kann, daß von ihr gestartete Prozesse andere Programme starten.

Man könnte darüber hinaus den *ModuleStamper* als Treiber implementieren, so daß er im Kernelmodus läuft. Dort hätte er Zugriff auf das gesamte System und könnte zusätzliche Überprüfungen durchführen – beispielsweise Kernelmodus-Datenstrukturen auf Integrität prüfen. Die Stampdatei könnte mit Hilfe eines Dateisystemfiltertreibers im Dateisystem so „versteckt“ werden, daß andere Anwendungen gar nicht darauf zugreifen können – auch nicht Prozesse im Systemkonto.

Es wäre zu untersuchen, inwiefern sich das Konzept der Installationsshell und des SetSID-Dienstes erweitern ließe, um eine programmbasierte Zugriffskontrolle nach dem Clark-Wilson-Modell unter Windows 2000 zu implementieren. Jedes Programm könnte einen individuell beschränkten Zugriffstoken bekommen, so daß es immer nur eine Teilmenge der Rechte des aufrufenden Benutzers besitzt. Um individuelle Zugriffsrechte von Programmen auf Dateien zu realisieren, wäre für jedes Programm eine eigene SID nötig. Dies ließe sich über unterschiedliche Gruppen-SIDs realisieren. Jeder Benutzer würde eine Menge von Gruppen-SIDs besitzen. Für jedes Programm, das er ausführen darf, wäre eine SID in der Menge enthalten. Die Installationsshell könnte dann je nach Benutzer und Programm den Zugriffstoken durch Entfernen von SIDs und Privilegien beschränken.

Die genauen Implementierungsdetails müßten untersucht und der zusätzliche Verwaltungsaufwand abgeschätzt werden. Es wäre zu klären, wie sich die Erstellung und Zuordnung von Gruppen-SIDs bzw. Gruppenkonten automatisieren ließe, um die zusätzliche Verwaltungsarbeit dem Administrator zu ersparen. Da bei einer sehr hohen Anzahl von SIDs für die Zugriffskontrolle auch die Sicherheitsbeschreibungen und damit die ACLs der Dateien ansteigen würden, wären Performanzverluste aufgrund der Zugriffsprüfung zu analysieren.

In dieser Arbeit wurde nur das Betriebssystem Windows 2000 betrachtet. Die Übertragung der Konzepte auf andere Betriebssysteme könnte genauer untersucht werden. So unterscheidet *Linux* [Tra02] ebenfalls zwischen Kernelmodus und Benutzermodus, die geschilderten Angriffsmöglichkeiten mit Treibern bestehen dort analog mit *Ladbaren Kernelmodulen* (LKM). Um eine Modifikation von Betriebssystemdateien zu verhindern, bietet Linux sogenannte *chroot*-Umgebungen, in denen nichtvertrauenswürdige Programme ausgeführt werden können. Mit *GNU Hurd* [Gnu02] existiert ein Konzept eines Microkernels, bei dem Treiber wie normale Prozesse im Benutzermodus eingebunden werden und Dienste niemals mit *Superuser*-Rechten laufen.

A. Beispiele

Alle im folgenden gezeigten Beispiele wurden auf einem PC durchgeführt mit Betriebssystem Windows 2000 *Service Pack 2*, Pentium III Prozessor mit 850 MHz, 320 MB Hauptspeicher, *NVidia* Grafikkarte, *SoundBlaster* Soundkarte, Netzwerkkarte, ISDN-Karte, SCSI-Festplatten, SCSI-CDROM-Laufwerk, SCSI-DVD-Laufwerk und ATAPI-ZIP-Laufwerk. Die Ergebnisse können auf Computern mit einer anderen Konfiguration abweichen, insbesondere die Liste der geladenen Treiber.

Java-Programme wurden mit dem JDK 1.4 von Sun und Windows-Programme mit Microsoft Visual C++ 6.0 inkl. Platform SDK von November 2001 entwickelt und übersetzt.

A.1. Gerätetreiber unter Windows 2000

A.1.1. Installation eines Filtertreibers

Um die Installation eines Filtertreibers zu demonstrieren, wurde der vorhandene Filtertreiber `serenum.sys`, der Geräte an der seriellen Schnittstelle auflistet, kopiert. Die Kopie `marcel0.sys` wurde mit einem Hex-Editor geringfügig verändert[1].

Eine manuelle Treiberinstallation wurde simuliert, indem in der Registry unter dem Schlüssel *HKLM\SYSTEM\CurrentControlSet\Enum\Port*PNP0501\PnPBIOS_2* und ...*PnPBIOS_3* jeweils der Wert im Feld *UpperFilters* von „serenum“ auf „marcel0“ geändert wurde[2]. In *HKLM\SYSTEM\CurrentControlSet\Services* wurde ein neuer Schlüssel *marcel0* angelegt, in dem die Konfiguration des Treibers festgelegt wurde. Der Treiber wurde als *Demandstart*-Treiber (Startwert 03) eingetragen und erhielt sonst die gleichen Einstellungen wie *serenum*.

Diese Eintragungen genügen jedoch noch nicht, um den manipulierten Filtertreiber zu installieren. Sogar nach einem Neustart des Systems blieb `serenum.sys` als Treiber gela-

[1] Ein feststehender Text und die Versionsnummer in der Binärdatei wurden mit Hilfe von *Microsoft Visual Studio* geändert.

[2] Standardmäßig hat nur das Systemkonto Zugriff auf diese Schlüssel. Mittels *at.exe* wurde der *Registry*-Editor im Systemkonto gestartet, um die Schlüssel bearbeiten zu können.

den. Es mußte zusätzlich im Feld *UpperFilters* von *HKLM\SYSTEM\CurrentControlSet\Enum\ ACPI\PNP0501\1* und ..*2* analog der Wert „marcel0“ eingetragen werden.

Nach einem Neustart versuchte das System, den Treiber `marcel0.sys` zu laden. Auch ohne Neustart ließ sich der Treiber im laufenden Betrieb laden. Dazu wurden im Gerätemanager von Windows 2000 die Geräteknoten für COM1 und COM2 gelöscht und anschließend der Vorgang *„Nach geänderter Hardware suchen“* aktiviert. In beiden Fällen wurde vom System lediglich *versucht*, den Treiber zu laden. Windows 2000 stürzte jeweils mit einer entsprechenden Fehlermeldung ab. Vermutlich führte die Änderung der Binärdatei zu Inkonsistenzen. Der Grund wurde allerdings nicht näher untersucht, da grundsätzlich demonstriert werden sollte, wie ein neu installierter Treiber im laufenden Betrieb geladen werden kann.

A.1.2. Deaktivierung von Datenträger-Geräteknoten

Geräteknoten lassen sich mit Hilfe der Funktionen `CM_Disable_DevNode()` und `CM_Enable_DevNode()` der Win32-API deaktivieren und aktivieren. Für Demonstrationszwecke geht es jedoch komfortabler über die graphische Oberfläche des Gerätemanagers von Windows 2000. Dort lassen sich Geräteknoten löschen oder nur vorübergehend deaktivieren.

Im Test wurden die Geräteknoten für CDROM-Laufwerke deaktiviert. Dies kann im laufenden Betrieb geschehen. Als Folge wurde sofort nach der Deaktivierung der Datenträgertreiber `cdrom.sys` entladen, was mit dem selbst entwickelten Programm *ListDrv.exe* überprüft werden konnte, das alle aktuell geladenen Treiber auflistet (siehe nächsten Unterabschnitt).

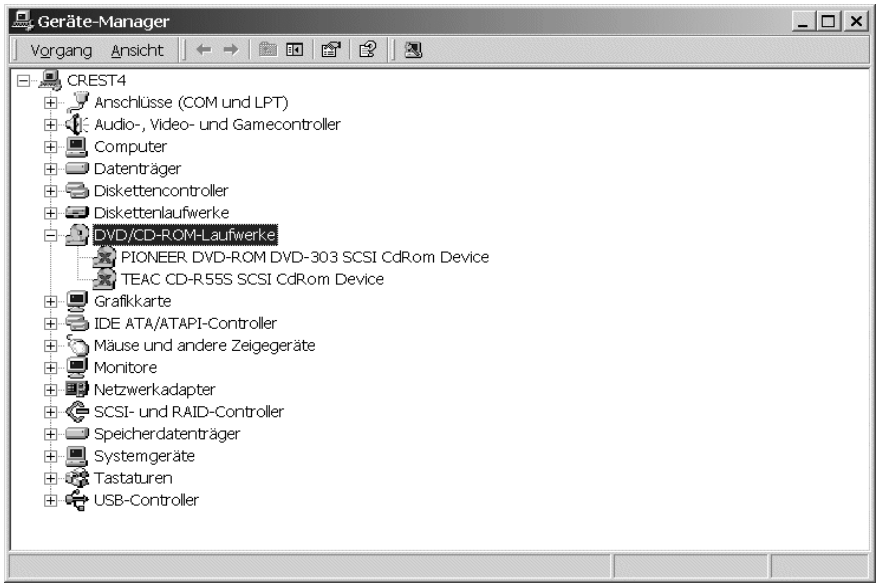

Abbildung A.1.: Gerätemanager mit deaktivierten CDROM-Laufwerken

Wenn mehrere Laufwerke im System installiert sind, müssen erst alle deaktiviert bzw. gelöscht werden, bevor der zugehörige Datenträgertreiber entladen wird. Es wurde beobachtet, daß der für CDROM-Laufwerke zuständige Dateisystemtreiber `cdfs.sys` jedoch immer im Speicher blieb, auch wenn der Datenträgertreiber entladen wurde.

Als einer der deaktivierten CDROM-Geräteknoten im Gerätemanager reaktiviert wurde, wurde der zugehörige Datenträgertreiber `cdrom.sys` wieder geladen, was ebenfalls mit *ListDrv.exe* überprüft wurde. Auch dies konnte im laufenden Betrieb geschehen. Gelöschte Geräteknoten lassen sich durch den Vorgang *„Nach geänderter Hardware suchen“* im Gerätemanager wieder hinzufügen – sofern die Geräte noch physikalisch angeschlossen sind.

Die Datenträger-Geräteknoten für Festplatten ließen sich nicht im laufenden Betrieb deaktivieren oder entfernen. Windows 2000 forderte einen Neustart, damit die Änderungen aktiv wurden. Daraus läßt sich schließen, daß der Datenträgertreiber `disk.sys` für Festplatten nicht im laufenden Betrieb ausgetauscht werden kann. Dies erscheint auch deshalb logisch, da sich das Betriebssystem selbst auf einer Festplatte befindet und permanent darauf zugreifen muß (z.B. auf die Auslagerungsdatei für den virtuellen Speicher).

A.1.3. Auflisten aller geladenen Treiber

Das Platform SDK enthält in der sogenannten PSAPI [MS01f] die Funktion `EnumDeviceDrivers()`, die eine Liste aller aktuell geladenen Treiber liefert. Genau genommen liefert sie eine Liste der Lade-Adressen. Mit der PSAPI-Funktion `GetDeviceDriverFileName()` läßt sich dann für jede Lade-Adresse der zugehörige Name der Treiberdatei bekommen.

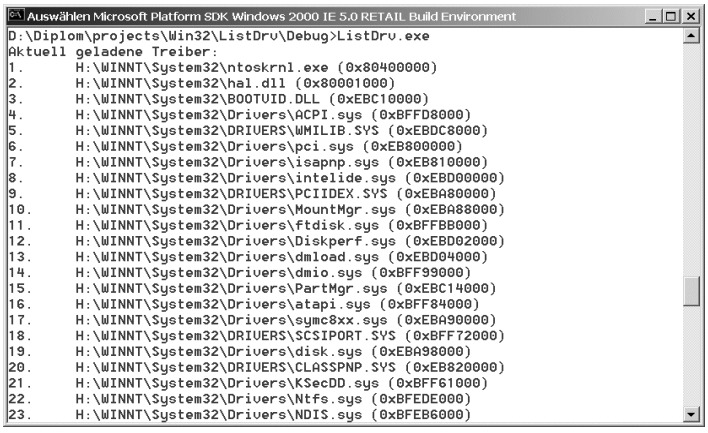

Abbildung A.2.: Ausgabe von *ListDrv.exe*

Diese Funktionen verwendet auch der *ModuleStamper*-Dienst. Für Demonstrationszwecke wurde ein eigenes Programm *ListDrv.exe* implementiert, das die Treiberdateien aller aktuell geladenen Treiber auf der Konsole ausgibt.

Auf dem Testsystem ergab sich folgende Liste der geladenen Treiber:

```
 1.     H:\WINNT\System32\ntoskrnl.exe
 2.     H:\WINNT\System32\hal.dll
 3.     H:\WINNT\System32\BOOTVID.DLL
 4.     H:\WINNT\System32\Drivers\ACPI.sys
 5.     H:\WINNT\System32\DRIVERS\WMILIB.SYS
 6.     H:\WINNT\System32\Drivers\pci.sys
 7.     H:\WINNT\System32\Drivers\isapnp.sys
 8.     H:\WINNT\System32\Drivers\intelide.sys
 9.     H:\WINNT\System32\DRIVERS\PCIIDEX.SYS
10.     H:\WINNT\System32\Drivers\MountMgr.sys
11.     H:\WINNT\System32\Drivers\ftdisk.sys
12.     H:\WINNT\System32\Drivers\Diskperf.sys
13.     H:\WINNT\System32\Drivers\dmload.sys
14.     H:\WINNT\System32\Drivers\dmio.sys
15.     H:\WINNT\System32\Drivers\PartMgr.sys
16.     H:\WINNT\System32\Drivers\atapi.sys
17.     H:\WINNT\System32\Drivers\symc8xx.sys
18.     H:\WINNT\System32\DRIVERS\SCSIPORT.SYS
19.     H:\WINNT\System32\Drivers\disk.sys
20.     H:\WINNT\System32\DRIVERS\CLASSPNP.SYS
21.     H:\WINNT\System32\Drivers\KSecDD.sys
22.     H:\WINNT\System32\Drivers\Ntfs.sys
23.     H:\WINNT\System32\Drivers\NDIS.sys
24.     H:\WINNT\System32\Drivers\Mup.sys
25.     H:\WINNT\System32\Drivers\agp440.sys
26.     H:\WINNT\System32\DRIVERS\VIDEOPRT.SYS
27.     H:\WINNT\System32\DRIVERS\nv4_mini.sys
28.     H:\WINNT\System32\DRIVERS\anvfltr.sys
29.     H:\WINNT\System32\DRIVERS\fdc.sys
30.     H:\WINNT\System32\DRIVERS\parport.sys
31.     H:\WINNT\System32\DRIVERS\serial.sys
32.     H:\WINNT\System32\DRIVERS\serenum.sys
33.     H:\WINNT\System32\DRIVERS\i8042prt.sys
34.     H:\WINNT\System32\DRIVERS\anvosdnt.sys
35.     H:\WINNT\System32\DRIVERS\kbdclass.sys
36.     H:\WINNT\System32\DRIVERS\USBD.SYS
37.     H:\WINNT\System32\DRIVERS\uhcd.sys
```

```
38.     H:\WINNT\System32\DRIVERS\fpcibase.sys
39.     H:\WINNT\system32\drivers\KS.SYS
40.     H:\WINNT\system32\drivers\portcls.sys
41.     H:\WINNT\system32\drivers\emu10k1f.sys
42.     H:\WINNT\System32\drivers\sfman.sys
43.     H:\WINNT\System32\Drivers\EFS.SYS
44.     H:\WINNT\System32\drivers\ctlface.sys
45.     H:\WINNT\System32\DRIVERS\ctljystk.sys
46.     H:\WINNT\System32\DRIVERS\gameenum.sys
47.     H:\WINNT\System32\Drivers\Cdr4_2K.SYS
48.     H:\WINNT\System32\DRIVERS\cdrom.sys
49.     H:\WINNT\System32\Drivers\Cdralw2k.SYS
50.     H:\WINNT\System32\DRIVERS\RTL8139.SYS
51.     H:\WINNT\System32\DRIVERS\avmwan.sys
52.     H:\WINNT\System32\DRIVERS\audstub.sys
53.     H:\WINNT\System32\DRIVERS\rasl2tp.sys
54.     H:\WINNT\System32\DRIVERS\ndistapi.sys
55.     H:\WINNT\System32\DRIVERS\ndiswan.sys
56.     H:\WINNT\System32\DRIVERS\TDI.SYS
57.     H:\WINNT\System32\DRIVERS\raspptp.sys
58.     H:\WINNT\System32\DRIVERS\ptilink.sys
59.     H:\WINNT\System32\DRIVERS\raspti.sys
60.     H:\WINNT\System32\DRIVERS\parallel.sys
61.     H:\WINNT\System32\DRIVERS\swenum.sys
62.     H:\WINNT\System32\DRIVERS\update.sys
63.     H:\WINNT\System32\DRIVERS\flpydisk.sys
64.     H:\WINNT\System32\DRIVERS\lsermous.sys
65.     H:\WINNT\System32\DRIVERS\mouclass.sys
66.     H:\WINNT\System32\DRIVERS\usbhub.sys
67.     H:\WINNT\System32\Drivers\NDProxy.SYS
68.     H:\WINNT\System32\Drivers\XWMSUSB.sys
69.     H:\WINNT\System32\Drivers\Fs_Rec.SYS
70.     H:\WINNT\System32\Drivers\Null.SYS
71.     H:\WINNT\System32\Drivers\Beep.SYS
72.     H:\WINNT\System32\drivers\vga.sys
73.     H:\WINNT\System32\Drivers\mnmdd.SYS
74.     H:\WINNT\System32\Drivers\Msfs.SYS
75.     H:\WINNT\System32\Drivers\Npfs.SYS
```

```
76.     H:\WINNT\System32\DRIVERS\rasacd.sys
77.     H:\WINNT\System32\DRIVERS\tcpip.sys
78.     H:\WINNT\System32\DRIVERS\msgpc.sys
79.     H:\WINNT\System32\DRIVERS\wanarp.sys
80.     H:\WINNT\System32\DRIVERS\netbt.sys
81.     H:\WINNT\System32\DRIVERS\netbios.sys
82.     H:\WINNT\System32\DRIVERS\anvioctl.sys
83.     H:\WINNT\System32\DRIVERS\rdbss.sys
84.     H:\WINNT\System32\DRIVERS\mrxsmb.sys
85.     H:\WINNT\System32\Drivers\Fastfat.SYS
86.     H:\WINNT\system32\win32k.sys
87.     H:\WINNT\System32\nv4_disp.dll
88.     H:\WINNT\System32\NV4_DISP.LRC
89.     H:\WINNT\System32\ANVMINI.DLL
90.     H:\WINNT\System32\drivers\afd.sys
91.     H:\WINNT\System32\Drivers\XWMSPAC.SYS
92.     H:\WINNT\System32\Drivers\XWMSPRO.SYS
93.     H:\WINNT\System32\Drivers\ParVdm.SYS
94.     H:\WINNT\System32\Drivers\Aspi32.SYS
95.     H:\WINNT\System32\Drivers\Fips.SYS
96.     H:\WINNT\system32\drivers\wdmaud.sys
97.     H:\WINNT\System32\PfModNT.sys
98.     H:\WINNT\system32\drivers\sysaudio.sys
99.     H:\WINNT\System32\DRIVERS\srv.sys
100.    H:\WINNT\System32\Drivers\Cdfs.SYS
101.    H:\WINNT\System32\DRIVERS\ipsec.sys
102.    H:\WINNT\System32\ATMFD.DLL
103.    H:\WINNT\system32\drivers\kmixer.sys
```

A.2. Die benötigten Module der Anwendung *Truesecs*

Die Funktion `EnumProcessModules()` aus der PSAPI [MS01f] liefert zu einem gegebenen Prozeß eine Liste seiner benötigten Module (DLLs). Man erhält eine Liste der Handles auf die Module, denn alle Module, die ein Prozeß benötigt, werden automatisch bei der Prozeßinitialisierung geladen. Mit der PSAPI-Funktion `GetModuleFileNameEx()` läßt sich für jedes Module-Handle der zugehörige Name der Moduldatei bekommen.

Diese Funktionen verwendet auch der *ModuleStamper*-Dienst. Für Demonstrationszwecke

wurde ein eigenes Programm *ListMods.exe* implementiert, das die Moduldateien eines über seine Prozeß-ID angegebenen Prozesses auf der Konsole ausgibt.

Auf dem Testsystem wurde das Java-Programm *Truesecs* gestartet und mit Hilfe des Task-Managers von Windows 2000 seine Prozeß-ID ermittelt. Das Programm *ListMods.exe* lieferte für diesen Prozeß folgende Module (ohne Treiber):

```
 1.      H:\Programme\JavaSoft\JRE\1.3.1\bin\javaw.exe
 2.      H:\WINNT\System32\ntdll.dll
 3.      H:\WINNT\system32\ADVAPI32.dll
 4.      H:\WINNT\system32\KERNEL32.DLL
 5.      H:\WINNT\system32\RPCRT4.DLL
 6.      H:\WINNT\system32\USER32.dll
 7.      H:\WINNT\system32\GDI32.DLL
 8.      H:\WINNT\system32\MSVCRT.dll
 9.      H:\Programme\JavaSoft\JRE\1.3.1\bin\hotspot\jvm.dll
10.      H:\WINNT\System32\WINMM.dll
11.      H:\WINNT\System32\AvmSnd.dll
12.      H:\Programme\JavaSoft\JRE\1.3.1\bin\hpi.dll
13.      H:\Programme\JavaSoft\JRE\1.3.1\bin\verify.dll
14.      H:\Programme\JavaSoft\JRE\1.3.1\bin\java.dll
15.      H:\Programme\JavaSoft\JRE\1.3.1\bin\zip.dll
16.      H:\Programme\JavaSoft\JRE\1.3.1\bin\awt.dll
17.      H:\WINNT\System32\WINSPOOL.DRV
18.      H:\WINNT\System32\IMM32.dll
19.      H:\WINNT\system32\ole32.dll
20.      H:\Programme\JavaSoft\JRE\1.3.1\bin\fontmanager.dll
21.      H:\WINNT\System32\DCIMAN32.dll
22.      H:\WINNT\System32\nvoglnt.dll
23.      H:\WINNT\System32\anvioctl.dll
24.      H:\DOKUME~1\marcel\LOKALE~1\Temp\A5355s._0a
25.      H:\WINNT\System32\nvoglnt.lrc
26.      H:\WINNT\System32\INDICDLL.dll
27.      H:\Programme\JavaSoft\JRE\1.3.1\bin\jpeg.dll
28.      H:\WINNT\System32\ddraw.dll
```

B. Klassendiagramme der P1363 Bibliothek

B.1. Mathematische und kryptographische Hilfsklassen

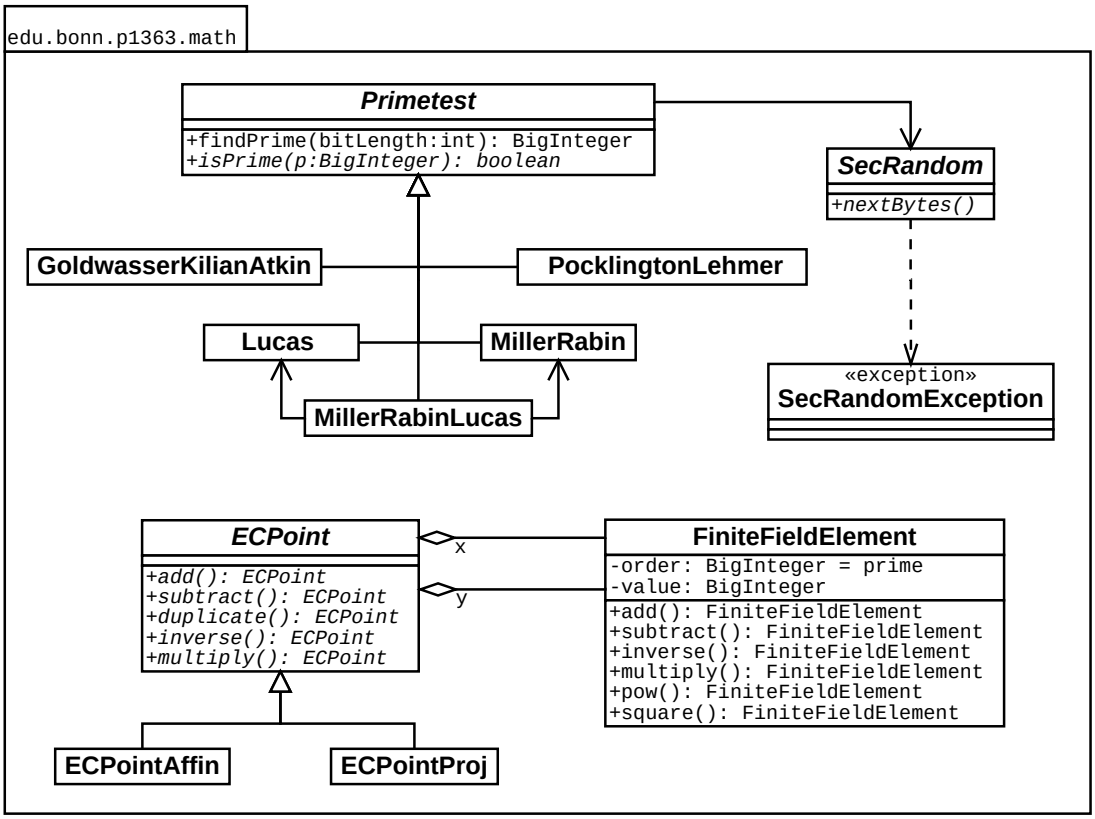

Abbildung B.1.: Klassendiagramm der mathematischen Hilfsklassen

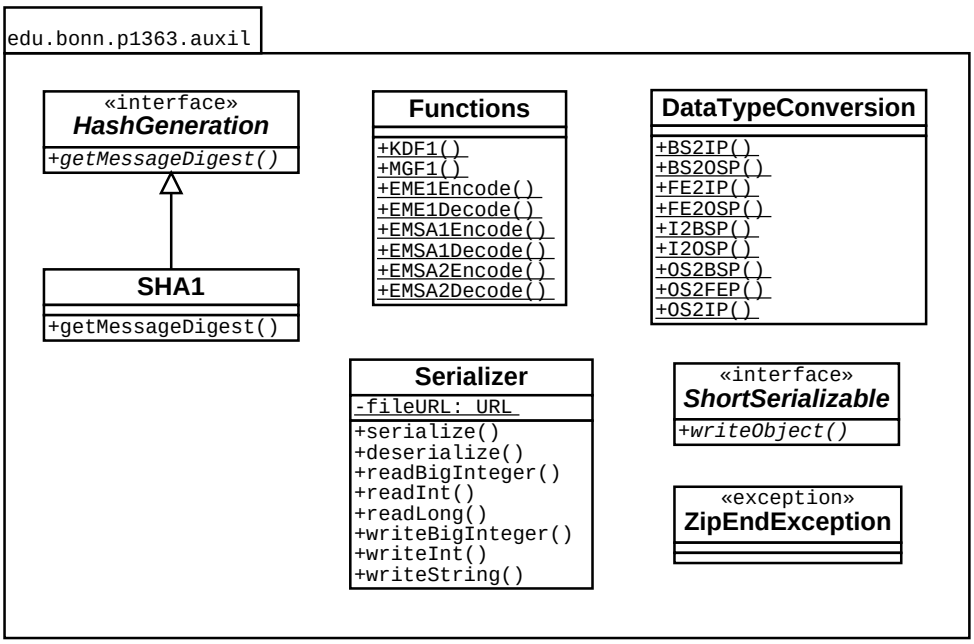

Abbildung B.2.: Klassendiagramm der kryptographischen Hilfsklassen

B.2. Schlüssel und Schlüsselgeneratoren

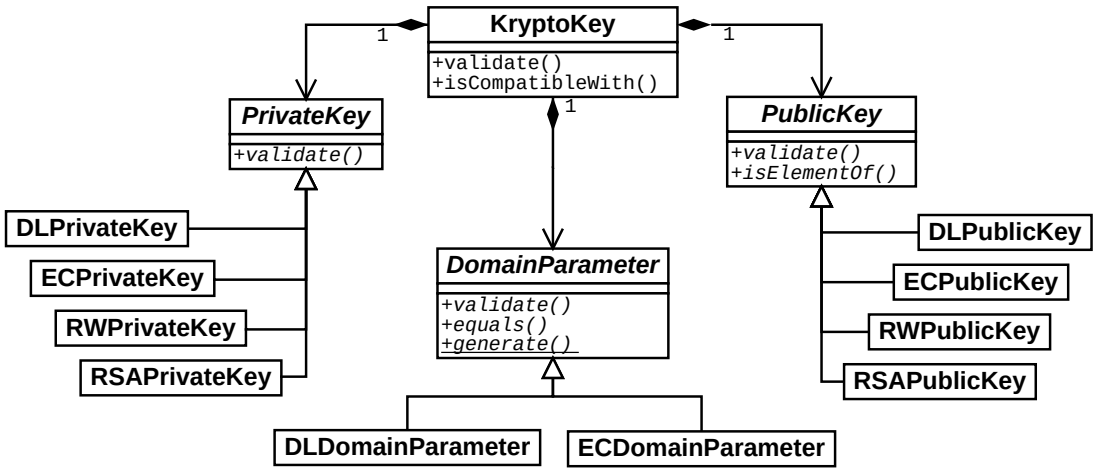

Abbildung B.3.: Klassenhierarchie der Schlüssel

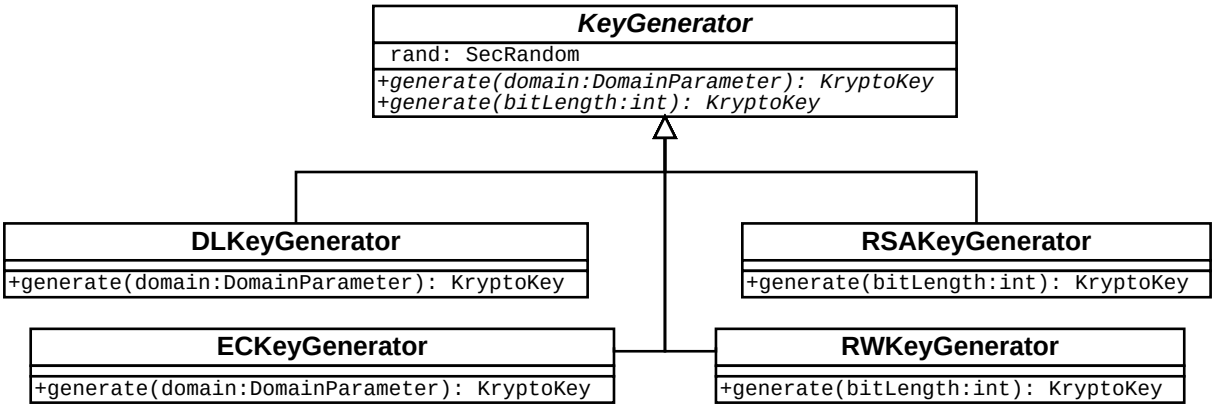

Abbildung B.4.: Schlüsselgeneratoren

B.3. Schemata

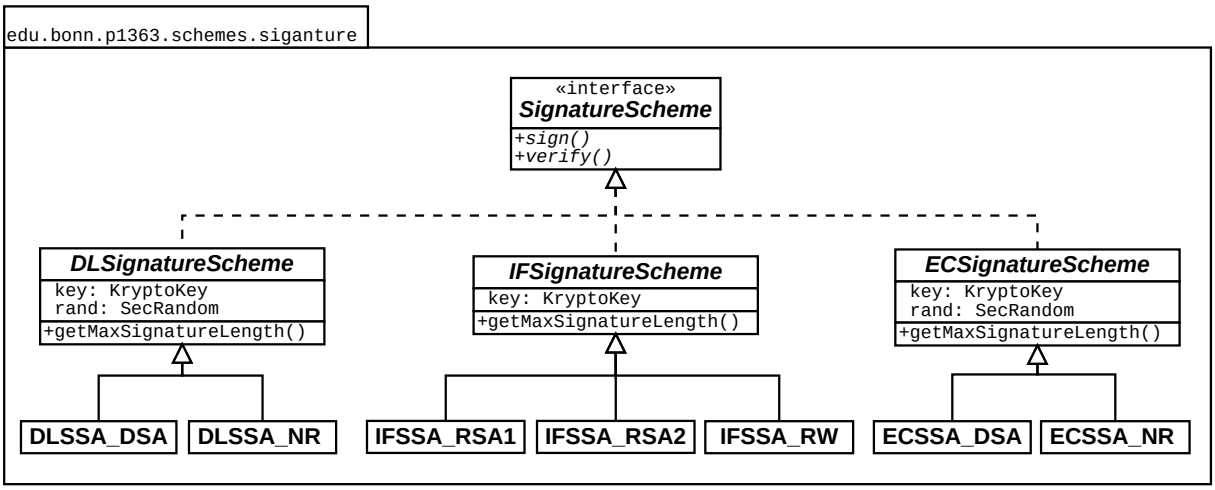

Abbildung B.5.: Klassendiagramm der Signaturschemata

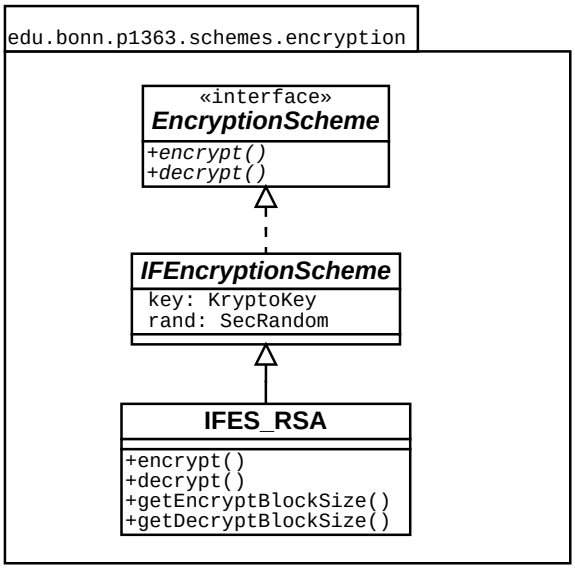

Abbildung B.6.: Klassendiagramm der Verschlüsselungsschemata

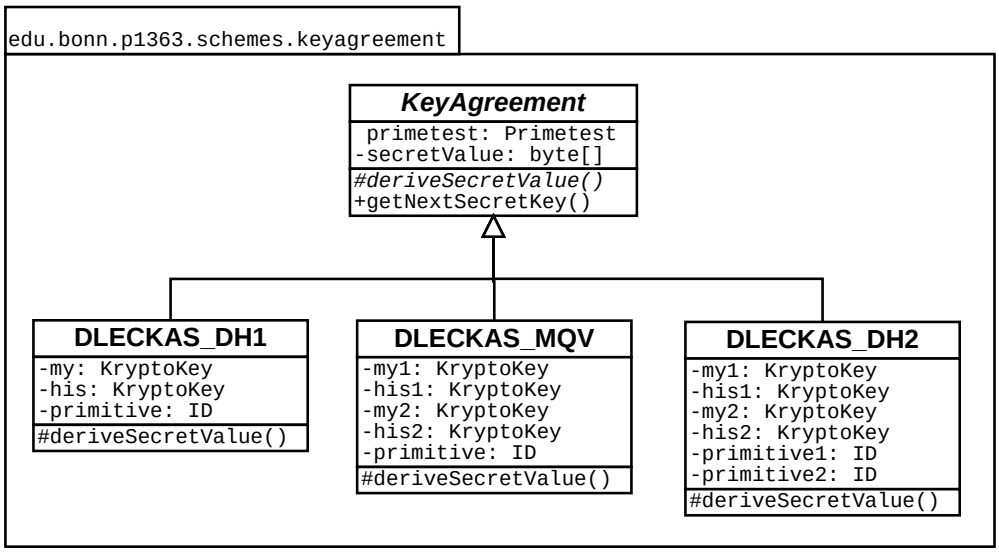

Abbildung B.7.: Schemata zur Vereinbarung gemeinsamer geheimer Schlüssel

Erklärung gemäß Diplomprüfungsordnung

Hiermit erkläre ich, daß ich diese Diplomarbeit einschließlich aller Abbildungen selbständig angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen sind, habe ich in jedem Einzelfall unter Angabe der Quelle als Entlehnung kenntlich gemacht.

Bonn, 21. Juni 2002

Literaturverzeichnis

[Bro99a] K. Brown, „Security Briefs“, *Microsoft Systems Journal*, Vol. 14 No. 5, 1999, `http://www.microsoft.com/msj/0599/security/security0599.htm`

[Bro99b] K. Brown, „Security Briefs“, *Microsoft Systems Journal*, Vol. 14 No. 8, 1999, `http://www.microsoft.com/msj/0899/security/security0899.htm`

[Bro00] K. Brown, „Handle Logons in Windows NT and Windows 2000 with Your Own Logon Session Broker“, *Microsoft Systems Journal*, Vol. 15 No. 2, 2000, `http://www.microsoft.com/msj/0200/logon/logon.asp`

[Fol97] S. Foley. „The Specification and Implementation of ’Commercial’ Security Requirements Including Dynamic Segregation of Duties“, *Proceedings of the 4th ACM Conference on Computer and Communications Security*, S. 125-134, ACM, 1997.

[FS00] M. Fowler, K. Scott. *UML konzentriert*. 2. Auflage, Addison-Wesley, 2000.

[Gol98] A. Goldberg. „A Specification of Java Loading and Bytecode Verification“, *Proceedings of the 5th ACM Conference on Computer and Communications Security*, S. 49-58, ACM, 1998.

[Gon98] L. Gong. „Secure Java Class Loading“, *IEEE Internet Computing*, 2 (6): S. 56-61, IEEE, 1998.

[Gon99] L. Gong. *Inside Java 2 Platform Security*. Addison-Wesley, 1999.

[GD99] L. Gong, S. Dodda. „Security Assurance Efforts in Engineering Java 2 SE (JDK 1.2)“, *Proceedings of the 4th IEEE International Symposium on High-Assurance Systems Engineering*, IEEE, 1999.

[GHJ95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Gnu02] GNU Project. *The GNU Hurd*, 2002, `http://www.gnu.org/software/hurd/hurd.html`

[GS98] L. Gong, R. Schemers. „Signing, Sealing, and Guarding Java Objects“, *Lecture Notes in Computer Science*, Vol. 1419, S. 206-216, Springer-Verlag, 1998.

[GWC98] A. Ghosh, J. Wanken, F. Charron. „Detecting Anomalous and Unknown Intrusions Against Programs“, *Proceedings of the 14th Annual Computer Security Applications Conference*, IEEE, 1998.

[Har01] J. M. Hart. *Win32 System Programming, Second Edition*. Addison-Wesley, 2001.

[Heu97] A. Heuser. „Bedrohungsangemessene Sicherheit – das Konzept des BSI für die Empfehlung und die Zulassung von Kryptosystemen“, *Tagungsband zum 5. Deutschen IT-Sicherheitskongreß des BSI*, S. 285-288, SecuMedia-Verlag, 1997.

[HKK00] M. Hauswirth, C. Kerer, R. Kurmanowytsch. „A Secure Execution Framework for Java“, *Proceedings of the 7th ACM Conference on Computer and Communications Security*, S.43-52, ACM, 2000.

[IB99] P. Iglio, F. U. Bordoni, „TrustedBox: a Kernel-Level Integrity Checker“, *Proceedings of the 15th Annual Computer Security Applications Conference* (ACSAC ’99), IEEE, 1999.

[IEE99] Institute of Electrical and Electronics Engineers, Inc. *IEEE P1363/D13 (Draft Version 13), Standard Specifications for Public Key Cryptography.* 12.11.1999.

[IGO98] J. Iliadis, S. Gritzalis, V. Oikonomou. „Towards Secure Downloadable Executable Content: The JAVA Paradigm“, *Lecture Notes in Computer Science*, Vol. 1516, S. 117 ff., Springer-Verlag, 1998.

[JBR99] I. Jacobson, G. Booch, J. Rumbaugh. *The Unified Software Development Process.* Addison-Wesley, 1999.

[JR98] S. Jovalekic, B. Rist. „Impact of Object-Oriented Software Engineering Applied to the Development of Security Systems“, *Lecture Notes in Computer Science*, Vol. 1516, S. 143 ff., Springer-Verlag, 1998.

[Kim98] F. Kim, „Why Do Certain Win32 Technologies Misbehave in Windows NT Services?“, *Microsoft Systems Journal*, Vol. 13 No. 3, 1998, `http://www.microsoft.com/msj/0398/service2.htm`

[Kok97] A. Koke. „Java und ActiveX – Gefahr aus dem Internet?“, *Tagungsband zum 5. Deutschen IT-Sicherheitskongreß des BSI*, S. 123-136, SecuMedia-Verlag, 1997.

[KS94a] G. H. Kim, E. H. Spafford, „Experiences with Tripwire: Using Integrity Checkers for Intrusion Detection“, *Proceedings of the Third Annual System Administration, Networking and Security Conference (SANS III*), S. 89-101, 1994.

[KS94b] G. H. Kim, E. H. Spafford, „The design and implementation of tripwire: a file system integrity checker“, *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, S. 18-29, ACM, 1994.

[Lan81] C. E. Landwehr, „Formal Models for Computer Security“, *ACM Computing Surveys Vol. 13, No. 3*. ACM, 1981.

[Lee99] E. S. Lee, *Essays about Computer Security*. University of Cambridge, 1999, `http://www.cl.cam.ac.uk/~mgk25/lee-essays.pdf`

[LGK99] C. Lai, L. Gong, L. Koved, A. Nadalin, R. Schemers. „User Authentication and Authorization in the Java(tm) Platform“, *Proceedings of the 15th Annual Computer Security Applications Conference*, IEEE, 1999.

[MOV97] A. J. Menezes, P. C. van Oorschot, S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[MPG00] S. Malabarba. R. Pandey, J. Gragg, E. Barr, J. F. Barnes. „Runtime Support for Type-Safe Dynamic Java Classes“, *Lecture Notes in Computer Science*, Vol. 1850, S. 337-361, Springer-Verlag, 2000.

[MQR97] M. Moriconi, X. Qian, R. Riemenschneider, L. Gong. „Secure Software Architectures“, *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, IEEE, 1997.

[MS00] Microsoft. *Microsoft Windows 2000 Security Technical Reference*. Microsoft Press, 2000.

[MS01a] Microsoft, „Driver Development Tools: Windows DDK – Disabler“, *Microsoft Developer Network Library*, 2001, `http://msdn.microsoft.com/library/en-us/ddtools/hh/ddtools/disabler_029f.asp`

[MS01b] Microsoft, „Platform SDK: Debugging and Error Handling“, *Microsoft Developer Network Library*, 2001, http://msdn.microsoft.com/library/en-us/debug/dbgerr_intro_6dbb.asp

[MS01c] Microsoft, „Platform SDK: DLLs, Processes, and Threads“, *Microsoft Developer Network Library*, 2001, http://msdn.microsoft.com/library/en-us/dllproc/dllproc_intro_0e43.asp

[MS01d] Microsoft, „Platform SDK: DLLs, Processes, and Threads – LocalSystem Account“, *Microsoft Developer Network Library*, 2001, http://msdn.microsoft.com/library/en-us/dllproc/services_9bhw.asp

[MS01e] Microsoft, „Platform SDK: Network Management“, *Microsoft Developer Network Library*, 2001, http://msdn.microsoft.com/library/en-us/netmgmt/ntlmapi_5ng4.asp

[MS01f] Microsoft, „Platform SDK: Performance Monitoring – Process Status Helper“, *Microsoft Developer Network Library*, 2001, http://msdn.microsoft.com/library/en-us/perfmon/psapi_25ki.asp

[MS01g] Microsoft, „Platform SDK: Security“, *Microsoft Developer Network Library*, 2001, http://msdn.microsoft.com/library/en-us/security/Security/security.asp

[MS01h] Microsoft, „Platform SDK: Windows File Protection“, *Microsoft Developer Network Library*, 2001, http://msdn.microsoft.com/library/en-us/wfp/wfpstart_85r9.asp

[MS01i] Microsoft, „PnP Configuration Manager Functions“, *Microsoft Developer Network Library*, 2001, http://msdn.microsoft.com/library/en-us/install/hh/install/cfgmgrfn_88oi.asp

[MS01j] Microsoft, „How to Replace In-Use Files at Windows Restart (Q181345)“, *Microsoft Knowledge Base*, 2001, http://support.microsoft.com/default.aspx?scid=kb;en-us;Q181345

[MS01k] Microsoft, „How to Enable or Disable Automatically Running CD-ROMs (Q155217)“, *Microsoft Knowledge Base*, 2001, http://support.microsoft.com/default.aspx?scid=kb;en-us;Q155217

[MZ97] G. Mohay, J. Zellers, „Kernel and Shell Based Application Integrity Assurance“, *Proceedings of the 13th Annual Computer Security Applications Conference* (ACSAC ’97), IEEE, 1997.

[Neu00] P. Neumann. „Robust Nonproprietary Software“, *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, IEEE, 2000.

[NIS95] National Institute of Standards and Technology. *FIPS PUB 180-1, Secure Hash Standard*. U.S. Department of Commerce, 1995.

[NIS99] National Institute of Standards and Technology. *FIPS PUB 140-2, Security Requirements for Cryptographic Modules.* Washington: U.S. Government Printing Office, 1999.

[NIS01] National Institute of Standards and Technology. *FIPS PUB 197, Advanced Encryption Standard (AES)*, 2001, `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`

[Pay00] C. Payne. „The Role of the Development Process in Operating System Security“, *Lecture Notes in Computer Science*, Vol. 1975, S. 277-291, Springer-Verlag, 2000.

[Pos98] J. Posegga. „Die Sicherheitsaspekte von Java“, *Informatik-Spektrum* 21: 16-22 (1998), Springer-Verlag, 1998.

[QGC00] Z. Qian, A. Goldberg, A. Coglio. „A Formal Specification of Java Class Loading“, *Proceedings of OOPSLA’00*, S.325-336, ACM, 2000.

[Ric97] J. Richter, „Design a Windows NT Service to Exploit Special Operating System Facilities“, *Microsoft Systems Journal*, Vol. 12 No. 10, 1997, `http://www.microsoft.com/msj/1097/WINNT.htm`

[Ric98] J. Richter, „Manipulate Windows NT Services by Writing a Service Control Program“, *Microsoft Systems Journal*, Vol. 13 No. 2, 1998, `http://www.microsoft.com/msj/0298/service.htm`

[Sch96] B. Schneier. *Applied Cryptography.* John Wiley & Sons, 1996

[Spa94] E. H. Spafford. „Computer viruses as artificial life“, *Journal of Artificial Life*, 1(3): S. 249-265, MIT Press, 1994.

[SR00] D. Solomon, M. Russinovich. *Inside Microsoft Windows 2000, 3. Auflage*. Microsoft Press, 2000.

[ST98] T. Sander, C. F. Tschudin. „Towards Mobile Cryptography“, *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, IEEE, 1998.

[Ste96] A. Sterbenz. “An Evaluation of the Java Security Model”, *Proceedings of the 12th Annual Computer Security Applications Conference*, IEEE, 1996.

[Sun02a] Sun Microsystems. *Java 2 SDK, Standard Edition Documentation*, 2002. `http://java.sun.com/j2se/1.4/docs/index.html`

[Sun02b] Sun Microsystems. *The Java Tutorial*, 2002. `http://java.sun.com/docs/books/tutorial/`

[Sun02c] Sun Microsystems, *RMI Specification*, 2002, `http://java.sun.com/j2se/1.4/docs/guide/rmi/spec/rmiTOC.html`

[Tra02] Transmeta Corporation, *The Linux Kernel Archives*, 2002, `http://www.kernel.org`

[WF98] D. S. Wallach, E. W. Felten. „Understanding Java Stack Inspection“, *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, IEEE, 1998.

[WS99] I. Welch, R. Stroud. „Supporting Real World Security Models in Java“, *Proceedings of the 7th IEEE Workshop on Future Trends of Distributed Computing Systems*, IEEE, 1999.

[ZL97] A. Zakinthinos, E.S. Lee. „A General Theory of Security Properties“, *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, S. 94-102, IEEE, 1997.