Sicherheitsdienste in Java2

Abstract

In einer Welt, die sich immer mehr vernetzt, erlangen Sicherheitsdienste für Menschen wie auch für Firmen eine immer größere Bedeutung. Dieser Artikel zeigt, wie die junge Programmiersprache Java dieser Entwicklung Rechnung trägt, und Antworten auf Sicherheitsbedürfnisse gibt. Dabei wird zunächst ein kurzer Überblick über die Entwicklung der Sprache und ihres Sicherheitsmodells gegeben. Im Anschluß wird das aktuelle Sicherheitsmodell vorgestellt, indem zuerst die ihm zugrundeliegenden Konzepte und danach die Verwirklichung dieser Konzepte anhand von Beispielen erläutert wird. Diese beiden Punkte sind jeweils nach Sicherheitsdiensten gruppiert, wobei die Authentisierung und Autorisierung einen besonderen Stellenwert einnehmen. Adressaten dieses Artikels sind Java-Interessierte, die einen Überblick über die Sicherheit in und mit dieser Sprache verstehen und kennenlernen wollen.

Inhalt

Abbildungsverzeichnis

Diese Version des Artikels ist vom 11.01.2001 und aktualisierte Fassungen sind unter www.ThorstenFischer.com verfügbar.

1 Einleitung

Nachdem nun Java bereits in Version „Java 2“ vorliegt, hat sich der „Hype“ um die Sprache zu Gunsten seriöser Betrachtungen etwas gelegt. Nun zeigt sich, daß etliche in Java verwirklichte Konzepte von Kritikern vorschnell mit dem Argument „Das ist doch nichts neues, das hat es alles schon gegeben“ abgetan wurden. Denn erst durch den Zusammenhang mit dem World Wide Web und durch Java haben sie einen erneuten Durchbruch gefunden.[1]

1.1 “Netzwerksprache Java”

Das frühe WWW war statisch. Dafür war es ursprünglich gedacht, und diesen Zweck erfüllte es im wissenschaftlichem Umfeld gut. Ein Mehrwert bestand ein der Verbesserung des Zugans zu Informationen und wurde durch Verbindung von Inhalten mit Hilfe von Links geschaffen; sogar Bilder konnten innerhalb von Webseiten angezeigt werden. Später konnten sogar in Browsern Formulare ausgefüllt werden und deren Inhalt an Programme, die auf den Rechnern der Webserver liefen, geschickt werden. Diese Programme konnten als Ergebnis wieder Webseiten an den Anwender zurückliefern. Doch etliche Anwendungen laufen besser auf den Client-Rechnern, etwa dann, wenn die Antwortzeiten reduziert werden sollen, oder die Netzbelastung in Betracht gezogen wird.

Genau in diesem Zeitpunkt, 1995, als das kommerzielle Web einen enormen Wachstumsschub erfuhr, präsentierte Sun die Programmiersprache Java, die nicht nur für Netzwerkanwendungen, sondern speziell auch für clientseitige Dynamik entworfen war. Ziel war es, Webinhalte dynamisch gestalten zu können, ja sogar komplexe Webanwendungen zu realisieren, welche die Zustandslosigkeit des HTTP überwinden konnten. Doch Code, der auf dem Client für den Anwender transparent geladen und ausgeführt wird, birgt das Risiko des „executable content“. Es gibt schon genug Viren, die über das WWW verbreitet werden, wie soll das erst werden, wenn man unbekannten Code auf seinem Rechner per Besuch einer Webseite ausführt?

Verteilte Anwendungen, auch außerhalb des Web, erfordern Kommunikation zwischen einzelnen Programmteilen. 3-Tier-Architekturen werden typischerweise an unterschiedlichen Stellen im Schichtenmodell in Client und Server unterteilt. Doch der Code

[1] Eine kritische Betrachtung der Faktoren, die bei der Markteinführung von Java zusammenwirkten, findet sich in [Fran98]

für den Client muß irgendwie auf den entfernten Rechnern installiert werden. Java wurde speziell für diesen Zweck konzipiert: Code kann dynamisch über ein Netz geladen werden. Doch genau diese Art der Auslieferung von Code stellt einen Schwachpunkt für mögliche Angriffe auf die Sicherheit dar.

Heute wird Java zunehmend auch in verteilten Multi-User Umgebungen eingesetzt. Dabei spielt es natürlich eine Rolle, welcher Anwender mit welchen Rechten auf welche Daten Zugriff erhält. Wie sieht die Authentisierung und Autorisierung der Anwender in Java aus?

1.2 Das Konzept der „Plattformunabhängigkeit“

Nicht nur das Problem des „executable content“ stellt sich mit dem Konzept clientseitiger Dynamik im Web. Für die zuvor statischen Inhalte des Web stellte die Heterogenität der Betriebssysteme der Clients kein Problem dar. Wie aber soll erreicht werden, daß Programme auf jedem Rechner, egal welchen Systems, ausgeführt werden können, ohne daß verschiedene Kompilate der Programme zur Verfügung gestellt werden müssen?

Die Idee der virtuellen Maschine, die Java als Lösung dieses Problems verwendet, ist nicht neu [Franz99]. Java-Quellcode wird eben nicht in plattformabhängigen Code, sondern in ein Zwischenformat, den sogenannten Bytecode übersetzt. Dieser Bytecode wird dann zur Programmausführung von einer – plattformabhängigen – Java Virtual Machine („JVM“) interpretiert[2].

2 “Java 1” (Historie)

Für die meisten Programmiersprachen stellt sich die Frage „Wie sicher ist das?“ nicht sofort. „Sicherheit“ wird mit diesen Sprachen in der Anwendung realisiert. Das Umfeld, für das Java konzipiert wurde, sorgte jedoch dafür, daß die in der Einleitung aufgeworfenen Fragen so grundlegend behandelt wurden, daß sie in den Entwurf der Sprache eingegangen sind.[PRGN99 S. 5] bezeichnen sogar „Java as a Part of Security“. Ein Java Sicherheitsmodell soll demnach *ganzheitlich*, *angemessen* und *nachhaltig* sein.

Ganzheitlich bedeutet dabei, daß in einem System, in dem Java benutzt wird, die Sicherheit an allen Stellen des Datenflusses im Netzwerk betrachtet werden muß. Insbe-

[2] Seit der Einführung der Just-In-Time Compiler und der HotSpot-Compiler stimmt der Ausdruck „interpretiert“ nicht mehr uneingeschränkt. Dazu mehr in Abschnitt 3.1.

sondere, wenn Java Applets über das Internet geladen werden, sollten folgende drei Punkte betrachtet werden:

- Netzwerksicherheit, realisiert mit Proxys und dazugehörigen Sicherheitspolitiken
- Datensicherheit, realisiert mit Kryptographie
- Authentisierung der Benutzer, realisiert mit digitalen Signaturen oder geschützten Paßwörtern

Angemessenheit bedeutet, daß das Sicherheitsmodell lediglich so stark sein muß, damit Java nicht zum schwachen Punkt für Angreifer wird, denn es macht keinen Sinn Geld oder Zeit aufzuwenden, wenn nicht zumindest einer der beiden folgenden Punkte zutrifft:

- Angreifer wollen nicht die Sicherheit des Systems knacken, sondern sie wollen speziell die Java-Sicherheit knacken, weil sie diese als schwächstes Glied ansehen.
- Die Anwender haben ein spezielles Mißtrauen Java gegenüber, und es sind Maßnahmen nötig, um ihr Vertrauen wiederzugewinnen.

Nachhaltigkeit bedeutet, daß die Vorkehrungen nicht nur einmalig gegen heute bekannte Angriffe getroffen werden dürfen, sondern daß das Sicherheitsmodell einer laufenden Kontrolle unterliegen muß und anpaßbar ist, wenn sich die Sicherheitsanforderungen ändern.

Das Java-Sicherheitsmodell hat sich nach und nach entwickelt. Im folgenden werden die Sicherheitskonzepte der Versionen vor Java2 dargestellt. Darstellungen dieser Konzepte finden sich in [GMPS97], [PRGN99] oder [McFe97].

2.1 Version 1.0

Das ursprüngliche Sicherheitsmodell von Java ist bekannt geworden als „Sandkastenmodell“. In diesem Modell wird lediglich zwischen lokalem und entferntem Code unterschieden. Während der Ausführung eines Programms wird lokaler Code komplett als vertrauenswürdig angesehen. Ihm werden alle Rechte zum Zugang zu Systemressourcen zugestanden. Entfernter Code wird hingegen als unsicher betrachtet. Er läuft nur mit eingeschränkter Funktionalität in der „Sandbox“. Diese Einschränkungen sind

im einzelnen unter [SUNa] und [SUNb] sowie in [McFe97] dokumentiert und hier zusammengefaßt wiedergegeben. Der Begriff „Applet[3]“ meint hierbei ein Java-Programm, das innerhalb eines Webbrowsers ausgeführt wird. Hierzu enthält die HTML-Seite ein `<applet>`-Tag, anhand dessen der Browser erkennen kann, von wo er die Java Klassendateien laden kann.

Applets können keine Bibliotheken laden, oder native Methoden deklarieren. Applets dürfen lediglich ihren eigenen Java Code nutzen und auf das Java API zugreifen, das ihnen ihr Appletviewer zur Verfügung stellt. Jeder Appletviewer muß jedoch Applets mindestens den Zugang zu dem API ermöglichen, das in den java.* packages definiert ist.

Applets haben keinen Zugang zum Filesystem des Rechners, auf dem sie ausgeführt werden. Dazu gehört das Lesen, Schreiben, Umbenennen, Löschen oder das Auslesen der Größe oder der Existenz einer Datei und das Auslesen von Verzeichnisinhalten, das Erzeugen, Umbenennen oder Löschen von Verzeichnissen. Es ist jedoch möglich, Dateien über voll qualifizierte URLs zu lesen.

Ein Applet darf keine Netzwerkverbindungen, außer zu dem Rechner, von dem es geladen wurde, öffnen. Außerdem dürfen keine Ports geöffnet werden, um auf eingehende Netzwerkverbindungen zu lauschen und auch keine Netzwerkkontrollfunktionen implementiert werden. Zu letzteren gehören zum Beispiel `ContentHandlerFactory`[4], `SocketImplFactory`[5] oder `URLStramHandlerFactory`[6].

[3] applet = application snippet = Anwendungsstückchen

[4] Das `public interface ContentHandlerFactory` definiert eine Fabrik zur Erzeugung von `ContentHandler` – Objekten. Eine Implementierung dieser Schnittstelle sollte einen MIME-Typ auf eine Instanz eines solchen Objektes abbilden.

[5] Das `public interface SocketImplFactory` definiert eine Fabrik zur Erzeugung von Sokkets. Es wird von den Klassen `Socket` und `ServerSocket` benutzt, um konkrete Implementierungen zu erzeugen.

[6] Das `public interface URLStreamHandlerFactory` definiert eine Fabrik zur Erzeugung von Objekten, mit denen Url Stream Protokolle behandelt werden können. Es wird von der Klasse URL benutzt, um ein URLStreamHandler – Objekt für ein spezielles Protokoll zu erzeugen.

Ein Applet darf weder Programme auf dem Rechner auf dem es ausgeführt wird starten, noch Threads, die nicht zur gleichen `ThreadGroup` wie es selbst gehören, erzeugen oder manipulieren. Weiterhin darf ein Applet nicht über zum Beispiel einen Aufruf von `System.exit()` die Virtual Machine, von der es ausgeführt wird, beenden.

Ein Applet darf lediglich bestimmte Systemeigenschaften lesen. Diese sind:

Systemeigenschaft	Bedeutung
java.home	Java Installationsverzeichnis
java.class.path	Java classpath
user.name	Account-Name des Anwenders
user.home	Home-Verzeichnis des Anwenders
user.dir	Aktuelles Arbeitsverzeichnis

Weiterhin dürfen Applets auch keine Systemeigenschaften definieren.

Applet-Fenster sind immer durch eine besondere Warnung gekennzeichnet. Das ermöglicht es dem Anwender, Applet-Fenster von Fenstern anderer Anwendungen zu unterscheiden.

Applets dürfen den System Klassenlader nicht erweitern, überladen, überschreiben, oder ersetzen. Dieser wird beim Start des Webbrowsers festgelegt. Zudem dürfen Applets keinen eigenen Klassenlader definieren.

Applets dürfen den System `SecurityManager` nicht erweitern, überladen, überschreiben oder ersetzen. Dieser wird beim Start des Webbrowsers festgelegt. Zudem dürfen Applets keinen eigenen `SecurityManager` definieren.

Das Java Sicherheitsmodell ist bereits in Java Version 1.0.x mittels unterschiedlicher Mechanismen realisiert, die sich nach [McFe97 S.14] und [GMPS97] in vier Punkte eingliedern lassen.

(1) Entwurf der Sprache Java

Java ist *streng typisiert* und *typsicher*. Das bedeutet, daß lediglich Zugriff auf genau bestimmte, kontrollierbare Speicherbereiche möglich ist, die dafür spezielle Repräa-

sentationen besitzen. Als Folge ergibt sich, daß beliebige Speichermanipulationen nicht möglich sind. Java kennt lediglich Objektreferenzen, keine echten Zeiger, mit denen auch arithmetische Operationen zugelassen wären.

Java ist im Vergleich zu C bzw. C++ eine relativ einfache Sprache, denn Konzepte und Konstrukte wie Macros, das `goto`-Statement, Header-Files, `struct` und `union`, Operatorenüberladung und Mehrfachvererbung fehlen.

Weiterhin umfaßt Java ein automatisches Speichermanagement, das Konzept der Garbage-Collection, Bereichsüberprüfungen von Strings und Arrays, eine Unterstützung für Multi-Threading, sowie ein Konzept zur Ausnahmebehandlung.

All diese Punkte des Entwurfs der Sprache Java sollen dazu führen, daß weniger Fehler in der Programmierung auftauchen, die sich als Sicherheitslöcher herausstellen können. Warum zum Beispiel die Typsicherheit vor Sicherheitslücken vorbeugt, illustrieren [McFe97 S. 51] an einem Beispiel:

Zur Laufzeit eines Java-Programms existieren zwei Objekte `Wecker` und `SecurityManager`. Beide stellen als erstes Feld eine boolsche Variable bereit. Im `SecurityManager`-Objekt soll diese Variable kennzeichnen, ob der Zugriff auf eine Datei erlaubt ist, während der Wecker anhand dieser Variable ein- und ausgestellt werden kann. Wäre Java nicht typsicher, könnte man eine Objektreferenz `w` vom Typ `Wecker` auf ein Objekt `SecurityManager` zeigen lassen und über den Aufruf der Methode `w.schalten(true)` eine Sicherheitslücke eröffnen. Die Methode weist der ersten boolschen Variable im zugrundeliegenden Objekt den Wert true zu. Sie ist eigentlich nur für Wecker-Objekte definiert. Diese Möglichkeit wird also durch die Typsicherheit der Sprache unterbunden.

(2) Class Loader

Die für das Laden einer Klasse zuständige Komponente der Laufzeitumgebung definiert für jede geladene Klasse einen eindeutigen Namensraum, damit keine gegenseitigen unerwünschten Beeinflussungen verschiedener Klassen zustande kommen können.

(3) Bytecode Verifyer

Der Prozeß der Bytecode-Verifikation, der nach dem dynamischen Laden einer Klasse zur Laufzeit eines Java-Programms ausgeführt wird, soll anhand einer statischen

Analyse der Syntax des Bytecodes gewisse Eigenschaften des Codes sicherstellen. Geprüft werden unter anderem

- die Syntax des Bytecodes,
- die Einhaltung der Zugriffsbeschränkungen für `private`, `protected` und `final` deklarierte Methoden und Felder und
- die Typsicherheit, soweit es eine statische Analyse zuläßt.

(4) SecurityManager

Diese Komponente regelt den Zugriff auf Systemressourcen zur Laufzeit des Programms.

An dem Sicherheitsmodell von Java 1.0.x sind im wesentlichen folgende Kritikpunkte geäußert worden:

Durch die Verteilung des Modells auf die drei letztgenannten Komponenten ist ein genaues Zusammenspiel nötig, um die Funktion des gesamten Modells zu ermöglichen. Der Anschein, das Modell würde aus drei Sicherheitsbarrieren bestehen täuscht und genau das Gegenteil ist richtig: wenn eine dieser Barrieren oder das Zusammenspiel fällt, ist die Sicherheit gebrochen.

Das Sicherheitsmodell ist nicht formal spezifiziert. Ohne eine solche Spezifikation ist aber die Bedeutung des Wortes „sicher“ nicht klar, zumal die Implementierung des Klassenladers und des `SecurityManager`s im Applet-Fall den Browserherstellern überlassen ist. Diese auf der einen Seite gewünschte Flexibilität macht auf der anderen Seite eine genaue Betrachtung einer jeden Implementierung nötig, um das Modell insgesamt Beurteilen zu können.

Das Sicherheitsmodell ist viel zu restriktiv und nicht einstellbar. Lokaler Code wird als komplett vertrauenswürdig eingestuft und erhält daher Zugang zu allen Systemressourcen, während entfernter Code nicht vertrauenswürdig ist, und seine Funktionalität so weit eingeschränkt wird, daß kaum mehr mit Applets anzufangen ist, als Webseiten nett zu dekorieren[7].

[7] Vollkommener Datenschutz bedeutet jedoch immer eine Einschränkung der Funktionalität. Am sichersten sind Daten wohl auf einem Rechner gespeichert, der sich ausgeschaltet in einem unzugänglichen Raum befindet.

2.2 Version 1.1

Das JDK 1.1 stuft lokalen Code ebenso wie die Vorgängerversionen als vollständig vertrauenswürdig ein. Ihm wird kompletter Zugang zu allen Systemressourcen zugestanden. Entwicklern von entfernetm Code ist es jedoch möglich, ihren Code digital zu signieren. Anhand dieser Signatur kann dann der Anwender feststellen, wer den Code erstellt hat und daran entscheiden, ob er diesem signierten Code vertrauen möchte, oder nicht. Je nachdem, wie diese Entscheidung ausfällt, läuft der Code dann wie bisher in der "Sandbox", oder wie völlig vertrauenswürdiger lokaler Code.

Auf eine Beschreibung, wie digitale Signaturen in Version 1.1 erstellt wurden, soll hier verzichtet werden. In Abschnitt 4.2.1.4 findet sich stattdessen das Vorgehen für Java2.

Das Sicherheitsmodell des JDK 1.1 stellt eine Verbesserung gegenüber dem der Version 1.0.x dar, jedoch ist es immer noch nicht möglich genau einzustellen, was Code, der als vertrauenswürdig eingestuft wird, darf, und was nicht. Weiterhin geschieht die Nutzung der Privilegien völlig automatisch, wenn entfernter Code einmal anhand des Zertifikates als vertrauenswürdig eingestuft wurde. Es ist für den Anwender nicht feststellbar, ob zum Beispiel in Dateien geschreiben wird, irgendwelche Netzwerkverbindungen benutzt werden oder Prozesse gestartet werden. Mit Java2 ist erstmals ein Mechanismus zum Protokollieren dieser Tätigkeiten verfügbar (Siehe Abschnitt 4.2.1.2) [PRGN99 S. 13]

3 Java 2

Bevor in Abschnitt 4 auf die Änderungen des Sicherheitsmodells in der neuen Version von Java eingegangen wird, soll hier zunächst ein Überblick über die weiteren Änderungen gegeben werden, welche die Entwicklung der Sprache mit sich brachte.

Mit der Einführung der Version 1.2 ließ Sun den Namen „Java 2" als Warenzeichen eintragen. Sun unterscheidet folgende Begriffe [SUNc]:

„Java™ 2 Platform, Standard Edition, v 1.2 (J2SE)"

Dieser Name bezeichnet die abstrakte Java-Plattform. Darunter wird die Technologie und/oder die Umgebung verstanden, die in Suns Spezifikationen definiert ist. Darunter fällt zum Beispiel auch das „Java™ 2 SDK, Standard Edition, v 1.2 (J2SDK)", das die Implementierung dieser Plattform von Sun selbst darstellt.

„Java™ 2 SDK, Standard Edition, v 1.2 (J2SDK)“

Dieser Name bezeichnet Suns Produkt, das eine Implementierung der J2SE darstellt. Es ist ein „Software Development Kit“ (SDK), das zur Anwendungsentwicklung sowohl die nötigen Tools (Compiler usw.) als auch die J2RE enthält. Dieser Name entspricht dem früheren „JDK version 1.2“

„Java™ 2 Runtime Environment, Standard Edition, v 1.2 (J2RE)“

Dieser Name bezeichnet Suns Implementierung der Laufzeitumgebung, mit der Anwendungen, die für die J2SE entwickelt sind, ausgeführt werden können. Dieser Name entspricht dem früheren „JRE 1.2“

Mittlerweile ist Java in Version 1.3 erhältlich. Die Begriffe werden analog verwandt, ebenso wie für die „Enterprise Edition“ (J2EE; zusätzlich zur Standard Edition werden Erweiterungen zur Entwicklung verteilter Anwendungen, wie Enterprise Java Beans - EJB, Servlets, Java Server Pages – JSP, zur Verfügung gestellt) und die „Micro Edition“ (J2ME; eine schlanke Variante der Standard Edition speziell zur Entwicklung von Anwendungen für Endgeräte). Dieses Kapitel soll einen kurzen Überblick über die Änderungen geben, die sich mit diesem Versionswechsel des JDK Version 1.1 auf das J2SDK ergaben. Eine ausführliche Darstellung bietet [SUNd].

3.1 Performanceverbesserungen

Ein Kritikpunkt an Java ist die absolut gesehen schlechtere Performance, die sich zum Beispiel im Vergleich zu in C++ erstelltem und zu Maschinencode kompilierten und statisch gelinktem Code ergibt. Es sollte jedoch bezweifelt werden, ob ein reiner Performancevergleich sinnvoll ist, denn man kann durchaus den Standpunkt begründen, daß Performanceverluste zugunsten der Plattformunabhängigkeit in Kauf genommen werden, weil Bytecode interpretiert wird. Mit der Einführung der *Just-In-Time Compiler* (*JIT-Compiler*) sind aber dennoch wesentliche Performanceverbesserungen für Java zu verzeichnen.

Diese Compiler übersetzen zur Laufzeit Bytecode vor der Ausführung methodenweise in Maschinencode. Mit J2SE Version 1.3 liegen nun auch *Hotspot-Compiler* vor, die Bytecode zunächst interpretieren. Meßwerte belegen, daß viele Programme 80 bis

90% ihrer Laufzeit mit der Ausführung von lediglich 10 bis 20% des Codes zubringen. Diese 10 bis 20% des Codes werden als *Hotspot* bezeichnet und können von einer *Hotspot-Virtual Machine* durch Beobachten des interpretierten Codes erkannt werden. Diese Teile des Codes werden dann zur Laufzeit kompiliert und als Maschinencode ausgeführt. Dieses gemischte Vorgehen zwischen Interpretierung von Bytecode und Ausführung von Maschinencode hat nicht nur Performancevorteile, sondern führt auch zu einer geringeren Speicherbelastung zur Laufzeit im Vergleich zu komplett kompiliertem Code.

Weitere Performanceverbesserungen sind in der Garbage-Collection und der Thread Synchronisation vorgenommen worden.

[Venn98] gibt einen Überblick über weitere Ansätze zur Performanceverbesserung von Java-Programmen.

3.2 Änderungen der Java Plattform (Auswahl)

Die folgende Auswahl stellt keine komplette Übersicht über alle Änderungen im angesprochenen Versionswechsel dar, denn eine solche würde den Rahmen dieses Artikels deutlich sprengen. Dennoch zeigt die Zusammenstellung, daß Java eine lebendige Sprache ist, die sich laufend fortentwickelt.

3.2.1 JFC / Swing

Das Projekt „Swing“ ist Teil der *Java Foundation Classes* (*JFC*) Software, die GUI-Komponenten mit einem *Pluggable Look & Feel* implementiert. Diese Implementierung ist komplett in Java Version 1.1 realisiert. Über das *Pluggable Look & Feel* kann der Programmierer dem Anwender auf höchst einfache Weise auch zur Laufzeit zur Auswahl stellen, welchem Stil in Aussehen und Bedienung das GUI der Anwendung entsprechen soll. Es stehen dabei das Design von MS Windows, Solaris, Macintosh sowie Motif und ein Java-eigenes Design zur Verfügung. Weiterhin sind über Swing die GUI Komponenten des *Abstract Windowing Toolkit* (*AWT*) und weitere GUI-Komponenten höherer Komplexität in einer reinen Java Implementierung zugreifbar.

Die Klassen, die das Swing-Framework bisher ausmachten, wurden aus ihrem bisherigen Namensraum `com.sun.java.*` nach `javax.*` verschoben.

Viele Verbesserungen dieses Frameworks wurden vorgenommen. Diese betreffen zum Beispiel die Performance, das Drucken, Benutzereingabenüberprüfungen oder die Unterstützung von HTML/CSS als Auszeichnungssprache für Beschriftungen des GUI.

3.2.2 Sound

Ab J2SE Version 1.3 steht ein neues API zur Verfügung, mit dem Audio- und MIDI (*Musical Instrument Digital Interface*) Daten geladen, verarbeitet und wiedergegeben werden können. Die Referenzimplementierung dieses APIs von SUN stellt dabei folgende Funktionen zur Verfügung:

- Audio Dateiformate: AIFF, AU und WAV. Dabei werden für diese Formate lineare, a-law und mu-law codierte Daten unterstützt.
- Musik Dateiformate: MIDI Type 0, MIDI Type 1 und Rich Music Format (RMF)
- Sound Formate: 8- und 16-bit Audiodaten in mono und stereo mit Sampleraten von 8 bis 48 kHz
- MIDI Wavetable Synthese und Sequenzierung als Softwareunterstützung und Zugang zu MIDI Hardwaregeräten
- Einen softwarebasierten Mixer zum Abmischen und Wiedergeben von bis zu 64 Kanälen digitaler Audio- und synthetischer MIDI Musik

3.2.3 Collections

Das *Collection-Framework* ist eine vereinheitlichte Architektur zur Darstellung und Manipulation von Kollektionen (Mengen, Listen und Abbildungen), die dabei unabhängig von der konkreten Implementierung ist. Es reduziert den Programmieraufwand und bietet Performanceverbesserungen. Weiterhin wird die Interoperabilitiät mit anderen APIs unterstützt, der Entwurf- und Lernaufwand für neue APIs erleichtert, und die Wiederverwendung von Code forciert. Das Framework basiert im wesentlichen auf sechs Interfaces und umfaßt Implementierungen dieser Interfaces incl. der nötigen Algorithmen.

3.2.4 Dateiorganisation

Alle Klassenbibliotheken des ehemaligen JDK sind nun in JAR-Archiven zusammengefaßt. ZIP-Archive funktionieren zwar auch weiterhin, jedoch führten diese oft zu Fehlbedienungen, wenn sie von Anwendern fälschlich entpackt wurden. In den früheren Versionen enthielt die Klassenbibliothek `classes.zip` sowohl die Klassen der Java-Plattform, als auch die Tool-Bibliothek für z.B. `javac`, `javadoc` usw. Diese ist nun in zwei Bibliotheken aufgeteilt:

- `jre/lib/rt.jar` für die Java Plattform Klassenbibliothek

- `lib/tools.jar` für die Tool-Bibliothek

Demzufolge wird auch das SDK ähnlich einer Java-Anwendung mit einer eigenen JRE vertrieben.

3.2.5 Primordial Classloader

Aus der geänderten Dateiorganisation (siehe 3.2.4) ergibt sich auch, daß Anwendungen, die bisher die Klassenbibliothek `classes.zip` direkt („hartcodiert“) ansprachen, in der J2SE nicht mehr funktionieren. Anwendungen sollten nunmehr nicht mehr Klassenbibliotheksnamen in dieser Weise verwenden, sondern die Funktionalität des neuen *Primordial Classloaders* (manchmal auch: *Bootstrap Classloader*) nutzen, der einen eigenen Suchpfad zum Auffinden der Systemklassen verwendet. Es ist lediglich nötig, die Klassen der Anwendung in den Classpath aufzunehmen. Das feste Eincodieren von Klassenbibliotheksnamen war für manche Anwendungen nötig, die den Classpath gesondert definieren mußten und somit auch die Klassenbibliothek `classes.zip` explizit aufnahmen. Weitere Einzelheiten hierzu finden sich in Abschnitt 4.2.1.2

3.3 Änderungen der Tools des JDK/SDK

- Der Javacompiler `javac` ist aus Performancegründen komplett neu implementiert worden. Dabei ist die Fehlerbehandlung geändert worden.
- Die reservierte Stackgröße der Java-Application-Launcher für Win32 Plattformen `java.exe` und `javaw.exe` ist von 1 MB auf 256 KB reduziert worden. Dadurch ist die Ausführung von bis zu vier mal mehr Threads möglich, bevor ein Speicherüberlauf eintritt.
- Der Java-Application-Launcher `javaw` gibt nun Fehlermeldungen aus, wenn sie das Starten der Anwendung an sich betreffen.
- Das Tool `javadoc` ist mit neuen tags (z.B.: `@docRoot`, `@deprecated`) und neuen Aufrufparametern verbessert worden.

4 Sicherheit in Java2

Mit J2SE lag erstmals auch die Spezifikation des Sicherheitsmodells von Java2 vor [Gong98]. Aufgrund der in Abschnitt 2.2 angedeuteten Kritikpunkte des bisherigen Sicherheitsmodells wurden nun folgende Punkte verbessert:

Ausdehnung der Sicherheitsüberprüfungen auf alle Java-Programme.

Es wird nun nicht mehr automatisch die Annahme getroffen, daß lokaler Code automatisch völlig vertrauenswürdig ist. Vielmehr kann nun aller Code, egal ob Applet oder Applikation, signiert oder nicht signiert, lokal oder entfernt, der Anwendung einer Sicherheitspolitik unterworfen werden. Hiervon ausgenommen bleiben die Klassen, die den Kern der Sprache bilden und als vertrauenswürdig angesehen werden können (Siehe 4.2.1.2)

Fein-granulare Zugriffskontrolle

Zwar gab es schon ab Version 1.0 die Möglichkeit, über das Ableiten des `SecurityManager`s und des Klassenladers dem Anwender eines Browsers anzubieten, was ein Applet tun darf, und welche Aktionen ihm auf dem Rechner des Browsers verboten bleiben sollten, doch dies erforderte sehr hohen und sicherheitskritischen Programmieraufwand. Der Browser HotJava 1.0 von Sun nutzte diese Möglichkeit, um den Anwender eine kleine Auswahl an Sicherheitseinstellungen treffen zu lassen. Diese Möglichkeit war jedoch nur für Webbrowser- oder Webserverhersteller gegeben, welche die Java-Plattform in ihren Produkten bereitstellten. Das neue Sicherheitsmodell vereinfacht diesen Prozeß zum einen wesentlich, und zum anderen ist es erstmals der Anwender, der die Zugriffskontrolle steuern kann.

Einfach konfigurierbare Sicherheitspolitik

Die Möglichkeit, eine Sicherheitspolitik zu definieren, bestand bereits auch schon ab Version 1.0, doch wiederum war Programmieraufwand nötig, um diese Sicherheitspolitik in Java-Programmen umzusetzen. Mit dem neuen Sicherheitsmodell ist es nun ohne Programmieraufwand möglich, eine Sicherheitspolitik anzuwenden.

Einfach zu erweiternde Zugriffskontrollstrukturen

Um den Zugriff auf Ressourcen in den Versionen vor Java2 einzuschränken, war es nötig, den `SecurityManager` mit neuen Methoden auszustatten, die diese Einschränkungen realisierten. Dies ist mit dem neuen Sicherheitsmodell nicht mehr nötig. Dieses bietet nämlich über die Klasse `Permission` die Möglichkeit, konkrete Zu-

griffskontrollen zu formulieren und vom neuen `SecurityManager` anwenden zu lassen.

Im nun folgenden sollen die Änderungen des Sicherheitsmodells von Java2 näher betrachtet werden. In Abschnitt 4.1 werden dabei zunächst die zugrundeliegenden Konzepte erläutert, während in Abschnitt 4.2 die Umsetzung dieser Konzepte in Java dargestellt wird.

4.1 Konzepte

4.1.1 Role-Based Access Control

Wie bereits in der Einleitung erwähnt, werden in Java zunehmend auch Anwendungen für Multi-User Umgebungen verwirklicht. Dabei ist die Zugriffskontrolle[8] ein Sicherheitsdienst, der sich auf viele Arten realisieren läßt. Ein recht altes Konzept, bereits vielfach in existierenden Systemen angewendet, fand erst relativ spät Eingang in wissenschaftliche Betrachtungen. So stellten [SCFY96] und [Sand98] das Konzept des „Role-Based Access Control“ wie folgt vor.

Die grundlegende Idee des RBAC besteht darin, Rechte zu Rollen zuzuordnen. Diese Rollen werden von Anwendern erfüllt. Wenn ein Anwender nun in einer bestimmten Rolle auftritt, erhält er die Rechte, die seiner Rolle zugeordnet sind. Für eine erste Beurteilung dieses Konzeptes sei auf [Sand98] verwiesen, hier soll lediglich ein Einblick gegeben werden, worin es besteht.

Eine *Rolle* ist ein semantisches Konstrukt, mit dessen Hilfe eine Zugriffskontrollpolitik formuliert wird, im angewandten Sinne beispielsweise eine Arbeitsplatzaufgabe innerhalb eines Unternehmens, die Befugnisse und Verantwortlichkeiten der Person einräumt, die sie erfüllt. In Abgrenzung dazu bedeutet der Begriff der *Gruppe*, der zum Beispiel von Unix-Systemen implementiert wird, eine bestimmte Menge von Anwendern eines Systems, weniger eine Menge von Rechten. Eine Rolle verbindet hingegen beide dieser Mengen – die Menge der Anwender mit der Menge der Rechte. Dieser Unterschied zwischen *Rollen* und *Gruppen* wird an einem Beispiel deutlicher: In Unix-Systemen ist es prinzipiell einfach, die Zugehörigkeit eines Anwenders zu einer Gruppe,

[8] Unter Zugriffskontrolle soll hier der Prozeß verstanden werden, der den Zugriff auf die Ressourcen eines Systems nur solchen Programmen, Prozessen oder anderen Systemen gestattet, die dafür autorisiert sind. Im weiteren wird der Begriff Autorisierung als Synonym für diesen Sicherheitsdienst verwendet.

oder alle Mitglieder einer Gruppe zu bestimmen. Gruppen werden Rechte zugestanden oder verweigert, indem jeder Eintrag im Filesystem entsprechend gekennzeichnet wird. Das hat im wesentlichen zwei Folgen:

- Der Wunsch festzustellen, welche Rechte eine bestimmte Gruppe hat, macht ein Durchsuchen des gesamten Filesystems nötig.
- Die Zuordnungsgewalt von Rechten zu Gruppen ist in hohem Maße dezentralisiert.

Der Begriff der Rolle impliziert jedoch folgendes:

- Es ist relativ einfach möglich, festzustellen, welcher Anwender in welcher Rolle auftreten kann und
- welcher Rolle welche Rechte zugeordnet sind.
- Die Zuordnungsgewalt ist zentralisierbar.

RBAC stellt eine Familie von Modellen bereit, deren Komplexität recht unterschiedlich ist. Das Grundmodell $RBAC_0$ spezifiziert als Basismodell die Minimalanforderungen, die zur Unterstützung von RBAC gegeben sein müssen. Es erfährt in $RBAC_1$ und $RBAC_2$ voneinander unabhängige Erweiterungen. $RBAC_1$ fügt dem Grundmodell Rollenhierarchien hinzu, während $RBAC_2$ Einschränkungen auf die verschiedenen Komponenten des $RBAC_0$ zuläßt. $RBAC_1$ und $RBAC_2$ sind zunächst nicht miteinander vergleichbar. $RBAC_3$ stellt das konsolidierte Modell dar, das die Möglichkeiten von $RBAC_1$ und $RBAC_2$ (und damit implizit auch von $RBAC_0$) integriert.

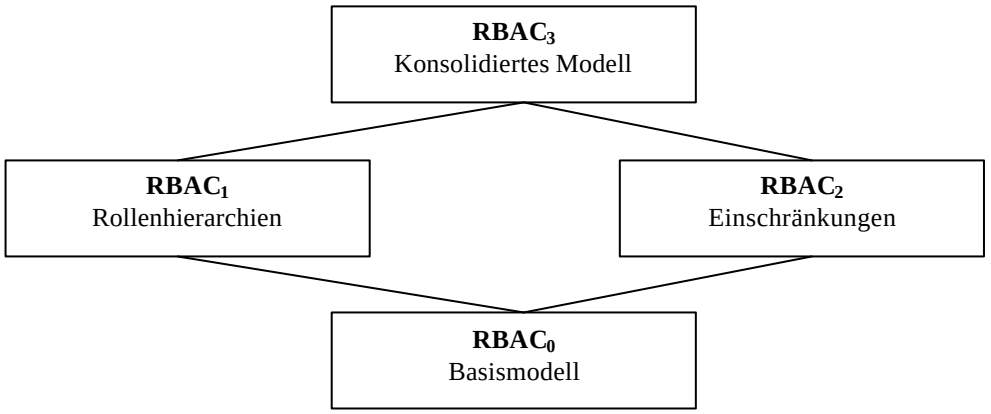

Abbildung 1 - Die RBAC Modellfamilie

4.1.1.1 $RBAC_0$

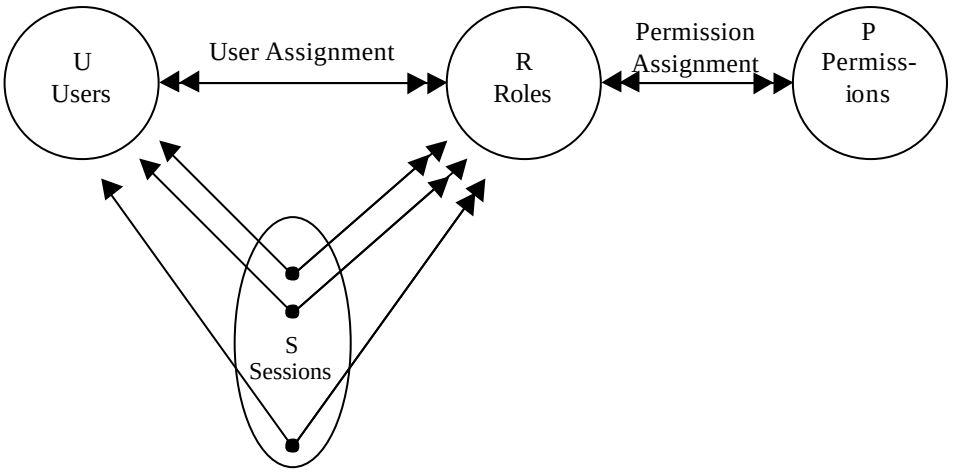

Abbildung 2 - $RBAC_0$

In $RBAC_0$ ist ein *Anwender* („*user*“) eine Person. Erweiterungen dieses Modells können diesen Begriff auch auf Dienste, Softwareagenten oder Netzwerke von Computern usw. ausdehnen. Zur Vereinfachung soll hier jedoch zunächst nur eine Person gemeint sein. Eine Rolle ist dann die Funktion oder Stelle, die diese Person in einem Unternehmen einnimmt und welche die damit verbundene Befugnisse ausdrückt.

Ein *Recht* („*permission*“) ist die erfolgte Genehmigung des Zugriffs[9] auf bestimmte Objekte des Systems. Die Begriffe Autorisierung, Zugriffsrecht und Privileg werden in der Literatur oft synonym verwendet, unter dem Begriff Autorisierung sei hier jedoch der Sicherheitsdienst verstanden, der diesen Genehmigungsprozeß darstellt, und ein Recht ist das Resultat dieses Prozesses. Rechte sind nur im positiven Sinne gemeint. *Negative Rechte* („Verbote“) können in RBAC durch Einschränkungen (also erst in $RBAC_2$ oder $RBAC_3$) annähernd abgebildet werden. Objekte seien sowohl Systemressourcen als auch Daten. RBAC unterstützt hier das Prinzip der Datenabstraktion. Demnach kann ein Recht im konkreten Anwendungsfall beliebig ausgestattet sein. Somit bleibt es der Anwendung überlassen, was sie als Recht definiert. Als Beispiele seien Lese- oder Schreibrechte für eine Datei, das Recht ein Konto zu belasten, einen Scheck zu ziehen oder einen Datensatz einer Datenbank aktualisieren zu dürfen genannt. Ebenso wird davon abstrahiert, auf wie viele Objekte sich ein Recht erstreckt.

In „Abbildung 2 - $RBAC_0$“ sind *Anwenderzuordnungen* („*User Assignment*“) und *Rechtezuordnungen* („*Permission Assignment*“) eingetragen. Beide dieser Zuordnungen sind n:m-Beziehungen. Ein Anwender kann in verschiedenen Rollen agieren und eine Rolle kann von verschiedenen Anwendern ausgeübt werden. Eine Rolle kann viele Rechte umfassen und ein und das selbe Recht kann vielen unterschiedlichen Rollen zugeordnet sein.

Indem die Rolle als vermittelnde Stelle zwischen Anwender und Recht dient, bietet sie der Zugriffskontrolle mehr Möglichkeiten, als wenn jedem Anwender direkt seine Rechte zugeordnet werden.

Jede *Sitzung* („*session*“) ist eine Abbildung eines Anwenders auf gegebenenfalls mehrere Rollen, was bedeutet, daß ein Anwender eine Sitzung eröffnet, während der er eine Teilmenge der Rollen aktiviert, die ihm zugeordnet sind. Der Pfeil mit der Doppelspitze von den Sitzungen zu den Rollen in „Abbildung 2 - $RBAC_0$“ soll bedeuten, daß ein Anwender diese Teilmenge seiner ihm zugeordneten Rollen aktiviert hat, während der Pfeil mit nur einer Spitze von den Sitzungen zu den Anwendern bedeutet, daß jede Sitzung von genau einem Anwender eröffnet wurde. Die Rechte, die ein Anwender während einer Sitzung ausüben darf, ergeben sich als Vereinigungsmenge aller Rechte, die den aktivierten Rollen dieser Sitzung zugeordnet sind.

$RBAC_0$ unterstützt das „*Prinzip des geringsten Rechts*“, denn ein Anwender, der mehrere Rollen erfüllen kann, hat mit dem Mittel verschiedene Sitzungen zu eröffnen, die Möglichkeit, nur diejenigen Rechte zu aktivieren, die er gerade zur Erfüllung seiner Aufgaben braucht.

[9] Unter Zugriff sei hier die Interaktion eines Subjektes mit einem Objekt gemeint, die einen Informationsfluß zwischen diesen beiden nach sich zieht.

4.1.1.2 $RBAC_1$

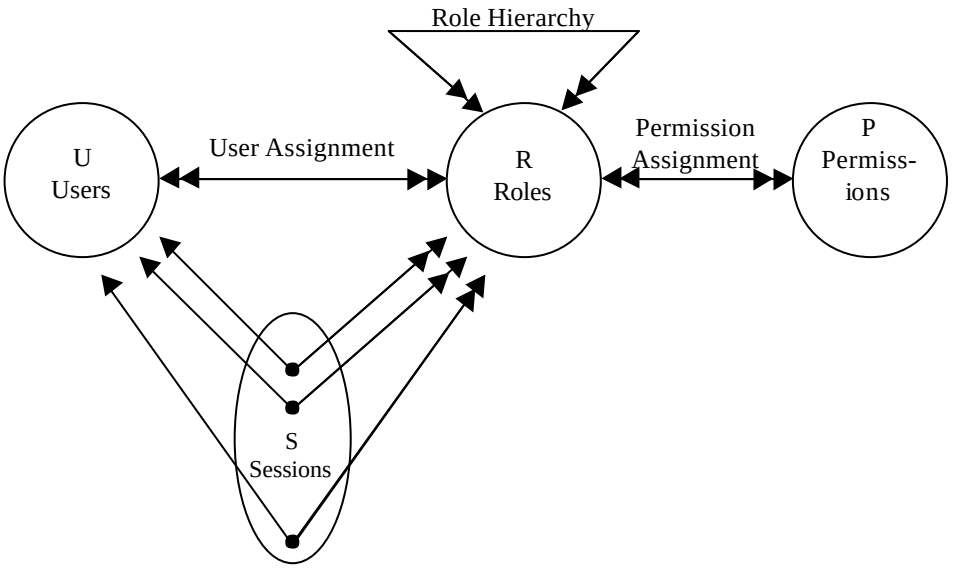

Abbildung 3 - $RBAC_1$

Zusätzlich zu $RBAC_0$ wird in $RBAC_1$ das Konzept der *Rollenhierarchien* eingeführt. Mit diesem Konzept lassen sich Rollen so strukturieren, daß sie die tatsächliche Organisation der Befugnisse und Verantwortlichkeiten von Unternehmen widerspiegeln. Dabei werden Vererbungsbeziehungen zwischen Rollen aufgebaut, was zur Folge hat, daß eine Seniorrolle alle Rechte von einer Juniorrolle erbt. Diese Vererbungsbeziehungen stellen in $RBAC_1$ partielle Ordnungen dar. Partielle Ordnungen haben diese drei Eigenschaften und deren Folgen:

- **Reflexivität** Die Vererbung ist reflexiv, was bedeutet, daß jede Rolle von sich selber ihre Rechte erbt.
- **Transitivität** Die Vererbung geschieht transitiv, was bedeutet, das eine Seniorrolle die Rechte auch aller Juniorrollen ihrer direkten Juniorrolle erbt. Somit sind beliebig tiefe Vererbungsstufen möglich.
- **Antisymmetrie** Die Vererbung ist antisymmetrisch, was zur Folge hat, daß gegenseitiges erben ausgeschlossen ist.

4.1.1.3 $RBAC_2$

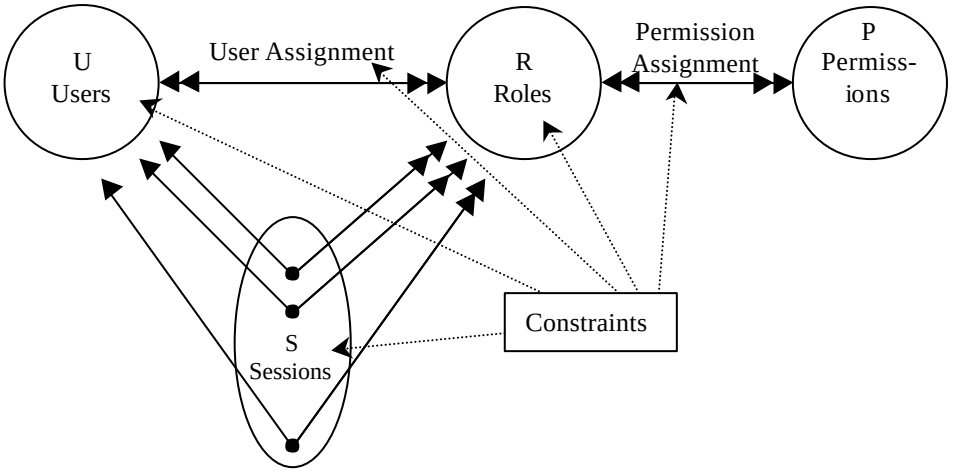

Abbildung 4 - $RBAC_2$

Zusätzlich zu $RBAC_0$ wird in $RBAC_2$ das Konzept der *Einschränkungen* („*Constraints*“) eingeführt. Abbildung 4 - $RBAC_2$ soll zeigen, daß alle Komponenten des $RBAC_0$ Einschränkungen unterliegen können. Eine Ausnahme bilden hier die Rechte, die ja uninterpretierte Symbole darstellen und von der Anwendung selbst beliebig implementiert werden. Als Beispiel für eine Einschränkung, die auf die Anwenderzuordnungen wirkt, kann man bestimmen, daß zwei Rollen sich gegenseitig ausschließen sollen, daß also ein Anwender nicht gleichzeitig diese beiden Rollen erfüllen darf. Beispielsweise sollten in der Softwareentwicklung die Rollen des „Programmierers“ und des „Testers“ einer solchen Einschränkung unterliegen. Diese Einschränkung wird auch als „*Prinzip der Gewaltenteilung*“ bezeichnet und somit von $RBAC_2$ unterstützt.

Eine Einschränkung in $RBAC_2$ bestimmt also, ob Werte der Komponenten des $RBAC_0$ zulässig sind, oder nicht. Man kann sie als Prädikate ansehen, die zu diesen Komponenten zugeordnet sind und welche die Aussagen „zulässig“ oder „nicht zulässig“ für die Komponenten zurückgeben.

4.1.1.4 $RBAC_3$

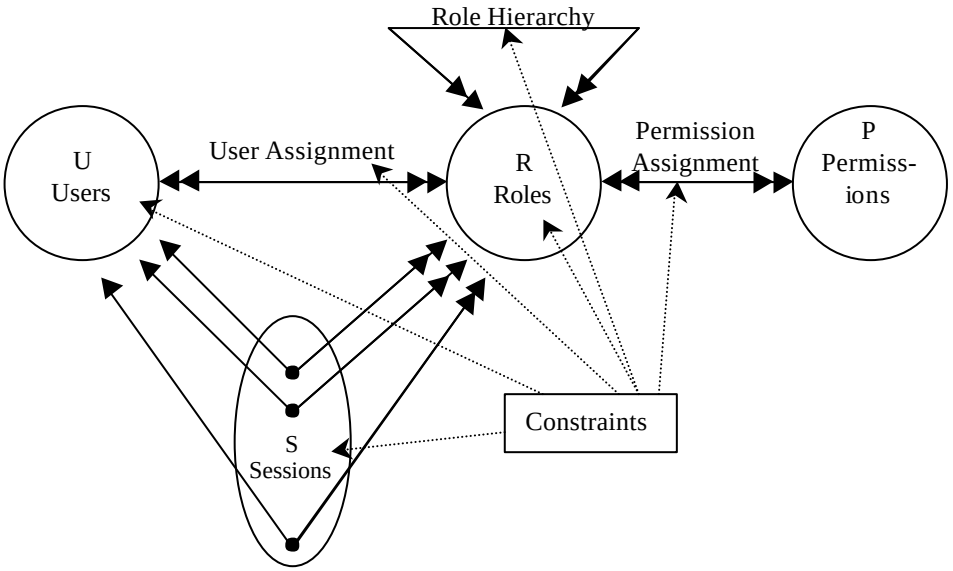

Abbildung 5 - $RBAC_3$

Bei der Konsolidierung von $RBAC_1$ und $RBAC_2$ zu $RBAC_3$ sind verschiedene Gesichtspunkte näher zu betrachten, um die Konzepte der Rollenhierarchien und der Einschränkungen in Einklang zu bringen.

Zunächst kann die Rollenhierarchie selbst Einschränkungen unterliegen. Daß die Rollenhierarchie eine partielle Ordnung darstellt, ist zum Beispiel eine Einschränkung, die implizit schon in $RBAC_0$ gegeben wird. Doch auch weitere Einschränkungen der Rollenhierarchie sind denkbar. Zum Beispiel könnte die Anzahl der Seniorrollen einer Rolle beschränkt werden.

Weiterhin können subtile Wechselwirkungen zwischen Einschränkungen und der Rollenhierarchie entstehen. Wenn die Einschränkung besteht, daß ein bestimmter Anwender nur einer einzigen Rolle zugeordnet werden darf, muß die Frage beantwortet werden, ob die Ableitung einer Seniorrolle dieser Rolle eine Verletzung der Einschränkung darstellt oder nicht.

4.1.2 “Fine Grained Access Control” in Java2

Unter feingranularer Zugriffskontrolle verstehen [PRGN99 S.74] die Möglichkeit, einem bestimmten Codestück bestimmte Rechte für den Zugriff auf bestimmte Ressourcen des Clients einzuräumen, die von der Signatur oder der Herkuft des Codes abhängen. Als Beispiel soll es möglich sein, daß der Code, der von

`http://www.ThorstenFischer.com` geladen wurde, die Datei `C:\Daten.txt` lesen aber nicht schreiben kann. Somit ermöglicht eine feingranulare Zugriffskontrolle dem Anwernder von Fall zu Fall entscheiden zu können, anstatt eine grobe Einteilung zwischen lokalem, entferntem signiertem und entferntem unsigniertem Code treffen zu müssen. Aller Code bekommt Zugriff auf Systemressourcen nur so, wie es die Sicherheitspolitik zuläßt. Diese Sicherheitspolitik kann in einem sogenannten *policy file* spezifiziert werden. Auf diese Weise kann sich der Anwender sicher sein, daß Code, den er aus dem Netz herunterlädt, installiert und ausführt, nur die Rechte hat, die er ihm explizit zugestanden hat.

Versteckte Funktionen kann ein Code dann nicht mehr ausführen, denn der Hersteller des Codes kann dem Anwender genau bekanntgeben, welche Rechte sein Programm braucht, um ordnungsgemäß abzulaufen, und der Anwender kann dann genau diese Rechte einrichten. Weiterhin kann in Multi-User Umgebungen ein Systemadministrator eine Sicherheitspolitik vorgeben. Jeder Anwender kann zusätzlich eine eigene Sicherheitspolitik definieren, die mit der Vorgabe kombiniert wird.

Die Grundlage für diese Art der Zugriffskontrolle bilden die Konzepte der *Schutzdomäne* (“*protection domain*”), der *Codeherkunft* (“*code source*”) und der *Sicherheitspolitikdatei* (“*policy file*”)[10].

policy file

Im *policy file* werden Rechte anhand der Herkunft (spezifiziert über einen URL) und/oder der digitalen Signatur des Codes diesem zugeordnet.

code source

Ein *code source*-Objekt für einen bestimmten Code ist die Kombination des URLs, von wo der Code geladen wurde und der Person oder der Personen, die den Code digital signiert haben.

protection domain

Eine protection domain umfaßt die Menge der Objekte, die zu einem gegebenen Zeitpunkt für einen *principal* direkt zugreifbar sind. Dabei sei ein *principal* eine Entität

des Systems, der Rechte zugeteilt wurden. Ein principal kann auf die Objekte in der protection domain zugreifen, weil er dafür bestimmte Rechte zugeteilt bekommen hat. Eine protection domain ist also die Zuordnung eines code source-Objektes zu den Rechten, die diesem Objekt im policy file zugestanden werden. Wenn nun eine Klasse geladen wird, wird sie automatisch einer protection domain zugeordnet. Diese Zuordnung geschieht anhand ihres code sources und den Rechten, welche der Klasse im policy file zugestanden werden. Klassen, denen zwar die gleichen Rechte zugestanden werden, die aber von verschiedenen Orten geladen worden sind, gehören deshalb auch zu unterschiedlichen protection domains. Protection domains werden in zwei Kategorien eingeteilt: die *system domain* und *application domains.* Unter der *system domain* kann man sich alle die Klassen vorstellen, die zum Java-Laufzeitsystem selbst gehören, und die deshalb keinerlei Einschränkungen unterliegen (müssen). Eine *application domain* wird durch ein Java-Programm (Application oder Applet) bestimmt. Es ist wichtig, daß die Zugriffe auf die Systemressourcen nur durch Zugriffe der system domain geschehen dürfen. Wenn ein Programmstück ausgeführt wird, können verschiedene Klassen zusammenwirken, also verschiedene protection domains einbezogen sein. Doch nur die Klassen, die der system domain angehören, können auf die Systemressourcen wie File I/O, Netzwerk I/O, Drucker, Bildschirm oder Tastatur zugreifen. Dies veranschaulicht Abbildung 6 - Zusammenspiel von protection domains.

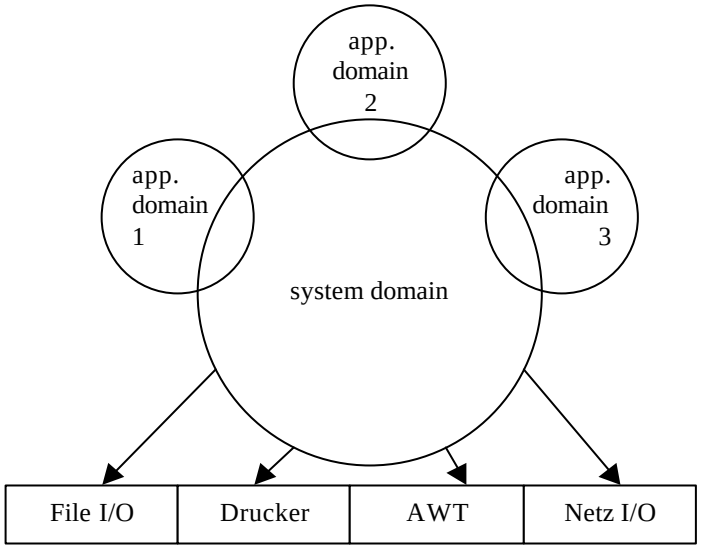

Abbildung 6 - Zusammenspiel von protection domains

[10] Im folgenden sollen diese Konzepte mit ihren englischen Namen bezeichnet werden, die somit besser als Termini erkennbar bleiben.

Ein Beispiel soll das Zusammenspiel nochmals verdeutlichen. Eine Anwendung, deren einzige Klasse die Meldung “Hallo Welt” auf dem Bildschirm ausgeben soll, sei in eine application domain eingeordnet. Um ihre Arbeit zu tun, muß sie allerdings Klassen der system domain benutzen, die den einzigen Zugangspunkt für Bildschirmausgaben darstellen. Bei diesem Vorgang ist es jedoch wichtig, daß die Klasse durch diese Benutzung der Klassen der system domain nicht weitere Rechte erhalten darf, als ihr selbst zugestanden worden sind. Ebenso muß diese Überlegung im umgekehrten Fall angestellt werden. Wenn eine Klasse der system domain eine Klasse einer application domain aufruft, darf durch diesen Aufruf keine Überschreitung der Rechte entstehen, welche der Klasse der application domain zugestanden worden sind. Zusammenfassend gilt, daß eine mit geringeren Rechten ausgestattete domain nicht zusätzliche Rechte erhalten darf, wenn sie von einer mächtigeren domain aufgerufen wird, oder diese aufruft.

Jede Klasse, die in die JVM geladen wird, wird genau einer protection domain zugeordnet, denn diese Zuordnung wird ja anhand des für jede Klasse eindeutig bestimmbaren code source-Objekt vorgenommen. Eine protection domain kann jedoch mehrere Klassen umfassen, denn verschiedene Klassen können sowohl den gleichen code source haben, als auch gleiche Rechte. Eine protection domain kann verschiedene Rechte umfassen, und ein und dasselbe Recht kann verschiedenen protection domains zugeordnet werden. Abbildung 7 veranschaulicht diesen Sachverhalt. Auf diese Art vermittelt eine protection domain zwischen den Klassen eines Programms und den Rechten, die diesen Klassen zugestanden werden, anstatt den Klassen direkt diese Rechte zuzuordnen. Es ergibt sich ein praktischer Mechanismus zur Gruppierung und Unterscheidung von verschiedenen Schutzzuständen und Schutzebenen.

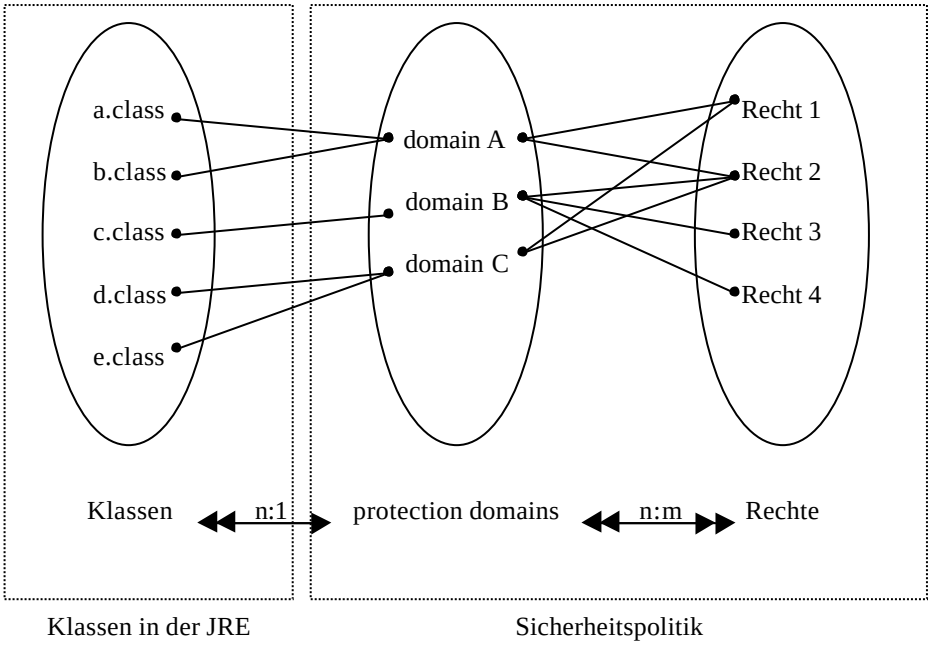

Abbildung 7 - protection domain Zuordnungen

Abbildung 7 - protection domain Zuordnungen - fordert einen Vergleich mit Abbildung 2 - $RBAC_0$ - heraus. Es stellt sich die Frage, ob das Konzept der feingranularen Zugriffskontrolle nicht eine Implementierung des RBAC-Modells darstellt. Unter einigen Voraussetzungen ist das tatsächlich so. Betrachtet man die Klassen in der JRE als die Anwender im RBAC-Modell, so erweitert man letzteren Begriff wie in Abschnitt 4.1.1.1 angedeutet in zulässiger Weise. Die protection domains stellen die Rollen dar; beide Konzepte dienen ja dazu, eine Zugriffspolitik zu formulieren. Der Begriff des Rechts wird in beiden Modellen analog verwandt. Eine Session des RBAC-Modells wird durch die konkrete Instanz des in Ausführung befindlichen Java-Programms in der JVM abgebildet. Dabei entsteht jedoch die Einschränkung, daß eine Klasse lediglich einer protection domain zugeordnet werden kann. Diese Einschränkung kann mit Hilfe $RBAC_2$ formuliert werden. Das Konzept der Rollenhierarchie kann für die Standardimplementierung des Sicherheitsmodells der J2SE in Form des J2SDK jedoch nicht übertragen werden, denn ein Mechanismus zur Formulierung von Vererbungsbeziehungen zwischen den protection domains ist nicht vorgesehen. Hierzu ist jedoch anzumerken, daß Anwendungsprogrammierer durchaus von der standardmäßigen Implementierung des policy files als Textdatei in der in

Abschnitt 4.2.1.2 beschriebenen Form abweichen können. Somit ließe sich auch ein solcher Vererbungsmechanismus implementieren. Sieht man davon zunächst ab, stellt die feingranulare Zugriffskontrolle unter den genannten Gesichtspunkten eine Implementierung des $RBAC_2$ dar.

Jedoch bleibt ein wesentlicher Unterschied zwischen RBAC, so wie es in Abschnitt 4.1.1 vorgestellt wurde und der feingranularen Zugriffskontrolle anzumerken: RBAC dient vornehmlich dazu, eine Zugriffskontrolle aufgrund der Tatsache, wer Code ausführt zu definieren, während das Java Sicherheitsmodell eine Zugriffskontrolle aufgrund der Tatsache definiert, von wo und von wem Code stammt. Diese letztere codebasierte Zugriffskontrolle stellt die Lösung für das Problem des "executable content" dar. Als anwerderbasierte Zugriffskontrolle, wie sie RBAC formuliert, kann das Framework *Java Authentication and Authorization* (*JAAS*) angewendet werden. Dieses soll als Umsetzung des RBAC in Abschnitt 4.2.2 vorgestellt werden.

4.1.3 Kryptographie

Zugriffskontrolle ist nicht der einzige Sicherheitsdienst, der ein sicheres System ausmacht. „Sicherheit" umfaßt weitere Dienste, die hier im Kontext angewendet definiert werden sollen.

(1) Authentisierung

Der Nachweis der angegebenen Identität oder Gruppenzugehörigkeit eines Subjektes bedeutet hier, daß ein Client sicher sein muß, daß Code von einem bestimmten Server stammt, bzw. von einer bestimmten Person erstellt wurde. Dieser Dienst läßt sich grundsätzlich durch drei unterschiedliche Prinzipien realisieren:

- Wissen (Paßwortverfahren unterteilt in konventionelle, one-time-, challenge/response- und public-key-Verfahren)
- Besitz (Identifikationsnachweise in körperlicher Form)
- biometrische Eigenschaften (Netzhautmuster, Fingerabdruck, Stimme...)

(2) Integrität

Während einer Übertragung sollen Daten unverändert bleiben, insbesondere auch dann, wenn die Übertragung über unsichere Medien, wie etwa das Internet erfolgt. Dieser Dienst kann mit Hilfe von Hashing-Verfahren und Kryptographie realisiert werden.

(3) Vertraulichkeit

Nur der oder die vom Absender bestimmten Empfänger sollen Daten lesen können. Dieser Dienst wird regelmäßig mit symmetrischer Verschlüsselung realisiert.

(4) Nicht Abstreitbarkeit (Verbindlichkeit)

Dieser Dienst gewährleistet die Bindung des Urhebers an sein Werk. Insbesondere soll sichergestellt werden, daß eine verläßliche Zuordnung zwischen einer bestimmten Transaktion und ihrem Verursacher möglich ist. Für die Gewährleistung dieses Dienstes ist der Einsatz asymmetrischer Kryptographie nötig.

(5) Verfügbarkeit

Für ein zu bestimmendes service-level ist ein verfügbarer Dienst immer erreichbar.

Während die ersten vier Sicherheitsdienste mit Hilfe kryptographischer Verfahren in Java gelöst werden können, so ist die Verfügbarkeit auch in dem neuen Sicherheitsmodell von Java2 ein nicht gelöster Punkt. Ein Anwender eines Applets kann nicht über das Sicherheitsmodell von Java bestimmen, wie viele Systemressourcen diesem in einer bestimmten Zeit zur Verfügung stehen sollen. Gleiches gilt für die Ausführung von Java-Applikationen.

Auf eine Erläuterung, auf welche Weise die in den Definitionen der einzelnen Dienste angesprochenen kryptographischen Verfahren konkret eingesetzt werden, um die einzelnen Sicherheitsdienste zu gewährleisten, soll hier verzichtet werden. Vielmehr sollen hier kurz nach [PRGN99 S. 480] die Prinzipen des *Java Cryptography Architecture* (*JCA*) Frameworks vorgestellt werden, die seinem Entwurf zugrunde liegen.

Das JCA ist das Framework, das den Zugang zu kryptographischer Funktionalität für Java-Entwickler schafft. Sein Entwurf berücksichtigt folgende Prinzipien:

(1) Unabhängigkeit von konkreten Implementierungen

Dieses Entwurfsprinzip wird durch eine providerbasierte Architektur realisiert. Der Begriff *cryptographic service provider* (kurz: *provider*) bezieht sich auf ein packa-

ge oder eine Menge von packages, die eine konkrete Implementierung der kryptographischen Aspekte des Java Security APIs unterstützen. Diese packages implementieren einen oder mehrere kryptographische Dienste wie etwa Algorithmen für digitale Signaturen oder Hash-Algorithmen. Provider können für die Anwendung transparent ausgetauscht werden. Dies ist nützlich, wenn z. B. schnellere oder sicherere Implementierungen benötigt werden.

(2) Interoperabilität der konkreten Implementierungen

Dieses Entwurfsprinzip bedeutet, daß verschiedene Implementierungen miteinander agieren können, daß sie zum Beispiel – gleiche Algorithmen vorausgesetzt – gegenseitig Schlüssel benutzen können oder gegenseitig erstellte Signaturen verifizieren können.

(3) Unabhängigkeit von konkreten Algorithmen

Dieses Entwurfsprinzip wird durch die Definition von Typen von kryptographischen Diensten und entsprechenden Klassen, die diese Dienste bereitstellen, verwirklicht. Diese Klassen werden als *engine classes* bezeichnet. Beispiele für solche engine classes sind die Klassen `MessageDigest`, `Signature`, oder `KeyFactory`.

(4) Erweiterbarkeit der konkreten Algorithmen

Dieses Entwurfsprinzip bedeutet, daß neue Algorithmen, die zu einer der unterstützten engine classes passen, auf einfache Weise hinzugefügt werden können.

4.2 Umsetzung

Dieser Abschnitt soll zum einen zeigen, wie die vorgestellten Konzepte aus Abschnitt 4.1.1 und 4.1.2 in Java umgesetzt sind, und wie Programmierer sie für sichere Anwendungen benutzen können. Insbesondere die Dienste Authentisierung und Autorisierung sollen in ihrer Umsetzung erläutert werden.

Der Abschnitt 4.2.1 beschreibt dabei, wie das Sicherheitsmodell von Java2 realisiert ist, und eine feingranulare Zugriffskontrolle benutzt wird, um das Ausführen einer Anwendung abzusichern.

Abschnitt 4.2.2 stellt die Umsetzung des RBAC-Modells in Java vor.

4.2.1 Die Elemente von Java

[PRGN99 S. 35] sehen das Sicherheitsmodell von Java2 in drei Elementen[11] der Sprache verwirklicht: die Entwicklungsumgebung, die Ausführungsumgebung und die Schnittstellen und Architekturen der Sprache.

4.2.1.1 Die Entwicklungsumgebung

Das J2SDK umfaßt als Entwicklungsumgebung die Werkzeuge und Programme, die zum Kompilieren und Testen von Java-Programmen nötig sind. Zum Kern der Sprache gehören komplett implementierte Frameworks zum Erstellen von graphischen Benutzeroberflächen ebenso wie für Netzwerk- und Datei- Ein- und Ausgaben.

Java bindet Klassen dynamisch d.h. zur Laufzeit von Programmen ein. Das hat zur Folge, daß zur Entwurfszeit beispielsweise nur zwei Klassen bearbeitet werden müssen, zur Laufzeit aber weitere Klassen benötigt und eingebunden werden.

Zur Illustration soll folgendes kleines Applet dienen, dessen Einbindung in eine HTML-Seite zum Beispiel mit `<applet Code="SmallApplet.class" width=200 Height=50></applet>` erfolgen kann. Das Applet besteht aus nur zwei Klassen, die Abbildung 8 zeigt.

[11] Der Begriff „*Components*“, den [PRGN99] an dieser Stelle verwenden, sei hier mit „*Elemente*“ übersetzt, um Verwechslungen mit dem Komponentenbegriff der Softwareentwicklung als Softwarebaustein mit wohldefinierten Schnittstellen zu vermeiden.

```
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class SmallApplet
           extends java.applet.Applet
           implements  ActionListener {

  Button machtNix = new Button
                       ("Keine Aktion");
  int    count    = 0;

  /** Button gedrückt */
  public void actionPerformed
                       (ActionEvent e) {
    machtNix.setLabel("Keine Aktion " +
                      ++count + " mal");
  }

  public void init() {
    setLayout(new BorderLayout());
    this.add("Center", machtNix);
    machtNix.addActionListener(this);
  }


}
```

```
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;

public class Button
                 extends    java.awt.Button
                 implements MouseListener   {

  public Button(String title) {
      super(title);
      addMouseListener(this);
      setBackground(java.awt.Color.white);
  }

  public void mouseEntered(MouseEvent m) {
    setBackground(java.awt.Color.yellow);
  }

  public void mouseExited(MouseEvent m) {
    setBackground(java.awt.Color.white);
  }

  public void mouseClicked(MouseEvent e) {}
  public void mousePressed(MouseEvent e) {}
  public void mouseReleased(MouseEvent e) {}

}
```

Abbildung 8 - Ein kleines Applet

Das Beispiel verdeutlicht, daß Klassen in packages eingeteilt werden, die von der JVM implementiert sein müssen. Weiterhin erbt jede Java-Klasse (bis auf die Klasse Object) von ihrer Oberklasse und sie kann selektiv Klassenmember überschreiben oder verschatten. Java identifiziert Klassen über ihren voll qualifizierten Namen, der aus der Kombination des package-Namens und des Klassennamens besteht. Das ermöglicht es, wie im Beispiel die Klasse Button zeigt, zwei Klassen gleich zu benennen.

Der Gebrauch der Schlüsselwörter `public`, `protected` und `private` ist ebenfalls ein sicherheitskritischer Aspekt. Wenn zum Beispiel die Liste der Dateien, die für ein Applet zugreifbar sind, im `SecurityManager` als public deklariert wären, könnte ein Angreifer leichter Manipulationsversuche starten. Für eine gegebene (kompilierte) Klasse liefert das Tool `javap` eine Auflistung aller Member dieser Klasse, die man mit den Optionen `–public`, `–protected` oder `–private` gefiltert anzeigen lassen kann.

Die Typsicherheit, als weiteres sicherheitskritisches Merkmal der Sprache ist nicht nur für den Kompilierungsvorgang wichtig, sondern muß gerade auch für den kompilierten Bytecode gelten. Wieder kann das Tool `javap` diesmal mit dem Aufruf `javap –c Calc` ein Beispiel liefern:

```
class Calc {
    public static void main(String[] args) {
      float  f = 3.0f;
      int    i = 2;
      double d;
      d = f*i;
    }
}


[...]
Method void main(java.lang.String[])
   0 ldc #2 <Real 3.0>
   2 fstore_1
   3 iconst_2
   4 istore_2
   5 fload_1
   6 iload_2
   7 i2f
   8 fmul
   9 f2d
  10 dstore_3
  11 return
```

```
Push Konstante #2 auf den Stack
Pop des ersten Items des Stack in Var. #1
Push Integer-Wert 2 auf den Stack
Pop des ersten Items des Stack in Var. #2
Push Wert der Var. #1 auf den Stack
Push Wert der Var. #2 auf den Stack
Typecast erstes Stack-Item von int n. float
Multipliziere die beiden floats auf dem Stack
Typecast erstes Stack-Item von float n. double
Pop des ersten Items des Stack in Var. #3
```

Abbildung 9 - Ein Java-Programm und sein Bytecode

Der komplette Bytecode soll an dieser Stelle nicht analysiert werden. Es sei nur darauf hingewiesen, daß die JVM eine stackbasierte Architektur hat, im Gegensatz zu z.B. Intels 80x86-Prozessoren, die registerbasiert arbeiten. Anweisungen, wie z.B. `iconst_2`, die auf diesem Stack arbeiten, sind typisiert – erkennbar an den Typ-Präfixen `i` für integer, `f` für floating point usw. Diese Typinformationen sind ebenso im Bytecode enthalten, wie Informationen über den Typ von Objektreferenzen. Beide Informationen werden von der Laufzeitumgebung zu Typüberprüfungen genutzt. Stimmen der Typ einer Instruktion und der Typ ihrer Argumente nicht überein, wird die JVM diesen Bytecode als unsicher ansehen und die Ausführung nicht zulassen. Das besondere am Entwurf dieser Typüberprüfungen ist dabei, daß sie nicht für jede Operation einzeln während der Ausführung geschehen, sondern nur ein einziges mal erfolgen müssen, nämlich beim Laden einer Klasse.

Die Informationen, die Bytecode in sich trägt, stellen einerseits die Implementierung des Sicherheitsmodells sicher. Anderesseits machen sie ihn für Dekompilierungsangriffe verwundbar. Decompiler machen den Kompilierungsvorgang rückgängig. Sie transformieren Bytecode in Quellcode. Für den Einsatz von Decompilern sprechen für Hacker verschiedene Gründe:

- Stehlen von Algorithmen
- Verstehen der Absicherung gegen Sicherheitsangriffe, um Lücken zu finden

- Erlangen Vertraulicher Informationen (Paßwörter oder Schlüssel), die von unerfahrenen Programmierern in den Code geschrieben worden sind
- Installieren von Trojanischen Pferden oder Viren
- Beweisen der eigenen Tüchtigkeit oder einfach nur zur Unterhaltung

Für Java-Bytecode existieren verschiedene Decompiler (z.B.: Mocha[12], SourceAgain[13]). [PRGN99 S. 134] stellen zwei Ansätze vor, die Dekompilierung erschweren: die *Code Obfuscation* und *Bytecode Hosing*.

Code Obfuscation

Ziel ist es, Quellcode für Menschen unlesbar zu machen. Dies wird durch Umbenennen von Bezeichnern oder durch Einfügen oder Löschen von Leerzeichen und Zeilenumbrüchen erreicht. Das Tool Crema des Autoren von Mocha geht noch weiter. Es ersetzt Namen für Konstanten im Bytecode, die von der JVM nur als tags zur Referenzierung der Werte benutzt werden, durch illegale Bezeichner oder reservierte Wörter. Dieses Vorgehen ändert nichts an der Ausführbarkeit des Bytecodes, verhindert aber ein erneutes Kompilieren des dekompilierten Bytecodes.

Bytecode Hosing

Die Strukturen des Bytecodes, die ein Decompiler ausnutzt, werden verwischt. Dies kann mit dem Einfügen von Instruktionen geschehen, die nichts bewirken, z.B. der NOP-Befehl oder ein PUSH, gefolgt von einem POP. Nachteilig ist dabei jedoch, daß diese Instruktionen wieder herausgefiltert werden können, und daß die JVM die ursprünglichen Muster auch zur Performanceoptimierung nutzt, die somit zunichte gemacht wird.

Da beide Ansätze keinen wirklichen Schutz vor einer Dekompilierung des Codes bieten, sollten Entwickler in Sicherheitsbetrachtungen immer davon ausgehen, daß ihr Code dekompiliert werden kann. Es sollten daher keinerlei sicherheitskritischen Daten wie z.B. Paßwörter, Schlüssel oder Umsatzzahlen, Gehälter o.ä. in den Code eingebracht werden.

[12] http://www.brouhaha.com/~eric/computers/mocha.html

4.2.1.2 Die Ausführungsumgebung

Bevor die JVM Bytecode interpretiert, richtet sie die Ausführungsumgebung ein, in welcher der *Class Loader*, der *Class File Verifier* und der *SecurityManager* als Komponenten für die Implementierung des Sicherheitsmodells zuständig sind.

Class Loader

[PRGN99 S. 47, 110, 145] Im Java2 Sicherheitsmodell werden Klassen zunächst eingeteilt in solche, die als völlig vertrauenswürdig eingestuft werden, und daher nicht dem Prozeß des Class File Verifiers unterworfen werden müssen, und solche, die als nicht vertrauenswürdig angesehen werden. Daher existieren auch zwei Arten von Klassenladern. Der „primordial Class Loader“ (auch: interner, null, oder default Class Loader) ist ein integraler Bestandteil der JVM. Er lädt alle vertrauenswürdigen Klassen.

In Version 1.1 gehörten dazu alle Klassen, die zum Kern der Sprache gehören oder in der Umgebungsvariable Classpath angegeben waren. Hierauf basierte ein Angriff, in dem eine bösartige Klasse in den Classpath aufgenommen wurde.

In Java2 gehört nun nur noch der Kern der Sprache zu den im vorhinein als vertrauenswürdig eingestuften Klassen. Der ursprüngliche Classpath ist nun für den neuen Klassensuchpfad in drei Teile zerlegt.

(1) Boot Class Path

Unter diesem Pfad ist der Kern der Sprache zu finden. Nach der Standardinstallation des J2SDK auf einem Windows-System gehören dazu die Archive `rt.jar`, `i18n.jar` in `${java.home}${/}lib`[14] (z.B.: `C:\Programme\javasoft\JRE\1.3\lib`) und ein Verzeichnis, das standardmäßig nicht existiert, nämlich `${java.home}${/}classes` (z.B.: `C:\Programme\javasoft\JRE\1.3\classes`). Zur Laufzeit kann dieser Pfad mit dem Befehl `System.getProperty(„sun.boot.class.path“)` ermittelt werden.

[13] http://www.ahpah.com/product.html

(2) Extensions

Das Java-Extension-Framework bietet einen Mechanismus, mit dem es möglich ist, den eigentlichen Kern der Sprache relativ klein zu halten, aber je nach bedarf weitere Komponenten hinzuzufügen, die dann auch als wie zum Kern gehörig betrachtet werden können. Solche Erweiterungen sind standardmäßig im Verzeichnis `${java.home}${/}lib${/}ext` zu finden. Dieser Pfad kann zur Laufzeit mit dem Befehl `System.getProperty(„java.ext.dirs“)` ermittelt werden.

(3) Application Class Path

Der anwendungsabhängige Klassensuchpfad wird über die Systemumgebungsvariable `CLASSPATH` oder durch den Parameter `–classpath` für `java` oder `javaw` eingestellt, und zur Laufzeit mit `System.getProperty(„java.class.path“)` ermittelt werden.

In allen drei Teilen des neuen Klassensuchpfades können Klassen direkt, in Verzeichnissen, JAR- oder ZIP-Archiven angegeben werden. Alle Klassen außerhalb des Boot Class Path werden in Java2 als nicht vertrauenswürdig eingestuft[15]. Diese Klassen werden von möglicherweise unterschiedlichen Implementierungen der Klasse `java.lang.ClassLoader` in die JVM geladen, welche aber alle sicherstellen, daß die geladenen Klassen dem Prozeß des Class File Verifiers unterworfen werden.

Class File Verifier

Fälschlicherweise wird der Class File Verifier von Java2 auch als Bytecode Verifier bezeichnet. Doch der Prozeß der Bytecodeverifikation ist nur ein Schritt, den der Class File Verifier mit einer geladenen Klasse ausführt. Damit Code einer Klasse zur Ausführung gelangen kann, muß sie nach dem Laden vier Prüfungen bestehen [PRGN99 S. 175].

[14] Die ${}-Notation bedeutet, daß der in geschweiften Klammern stehende Ausdruck je nach System oder Installation unterschiedlich sein kann, aber dennoch konkret gemeint ist.
[15] Für Klassen des Extension-Frameworks ist jedoch in der Standardinstallation des J2SDK im policy file die Vorgabe getroffen, daß auch diese als vertrauenswürdig eingestuft werden sollen.

(1) Dateiintegrität

Die Struktur der Klassendatei muß der Spezifikation entsprechen, wie sie in [LiYe99] festgelegt ist. Diese Prüfung wird anhand der Länge der Strukturelemente der Klassendatei vollzogen.

(2) Klassenintegrität

Diese Prüfung umfaßt alle Tests, die ohne eine Untersuchung des Bytecodes direkt möglich sind. Dadurch wird folgendes sichergestellt:

- Jede Klasse hat eine Oberklasse (Ausnahme: Klasse Object)
- Die Oberklasse ist nicht als final deklariert und die zu prüfende Klasse versucht nicht als final deklarierte Elemente der Oberklasse zu überschreiben
- Konstanten sind wohlgeformt und Referenzen auf Methoden und Felder der Klasse haben legale Namen und Signaturen

(3) Bytecodeintegrität

Diese Prüfung nimmt der *Bytecode Verifier* vor. Er überprüft folgende Anforderungen:

- Alle Argumente von Kontrollflußanweisungen (`do`, `while`, `for`, `if`...) müssen zu gültigen Anweisungen führen, wenn die entsprechende Anweisung beendet wird. Ähnliches gilt für Ausnahmebehandlungsanweisungen (`try`, `catch`, `finally`).
- Alle Referenzen auf lokale Variablen müssen deklariert sein.
- Alle Referenzen auf lokale Konstanten müssen den korrekten Typ aufweisen.
- Alle Bytecode-Anweisungen („*Opcodes*“) haben eine korrekte Anzahl an Parametern

Die Prüfungen finden statisch statt, was bedeutet, daß der Bytecode Verifier zu konstruieren versucht, wie sich der Code zur Laufzeit verhält, aber dabei den Code nicht ablaufen läßt. Die Prüfung endet mit der Einteilung des Codes in eine der folgenden Kategorien:

- Das Laufzeitverhalten des Codes ist als sicher bewiesen.
- Das Laufzeitverhalten des Codes ist als unsicher bewiesen.
- Das Laufzeitverhalten kann nicht in eine der beiden vorigen Kategorien eingeordnet werden.

Die Unsicherheit, die sich in einer Einordnung von Code in die letzte Kategorie widerspiegelt, kann zwar reduziert, aber nicht gänzlich eliminiert werden. Ein Beweis hierzu, begründet auf dem Halteproblem, findet sich in [PRGN99 S. 183].

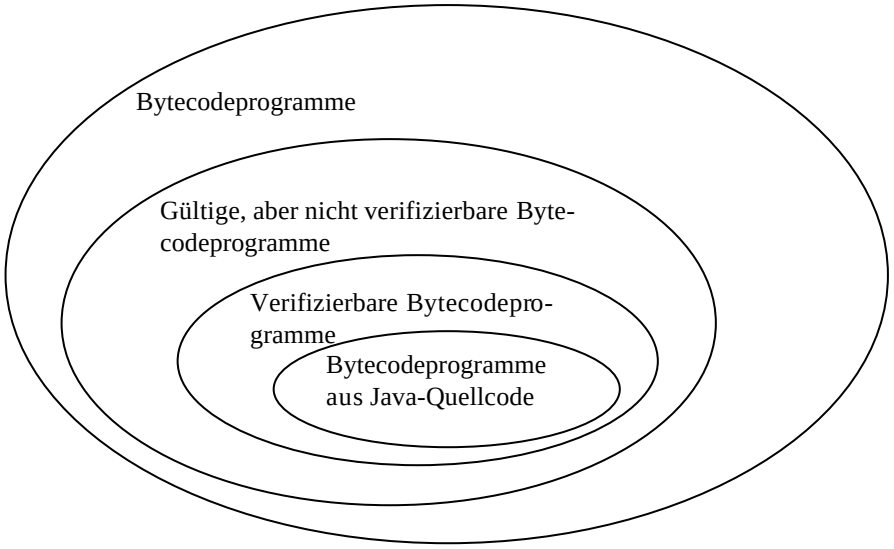

Abbildung 10 - Arten von Bytecodeprogrammen

Wie Abbildung 10 - Arten von Bytecodeprogrammen – zeigt, können Bytecodeprogramme nicht nur durch die Kompilierung von Java-Quellcode entstehen. Auch für andere Sprachen wie etwa `Ada`, `Cobol`, `Basic` und `NetRexx` existieren Bytecodecompiler. Die Regeln des Class File Verifiers sollen jedoch dazu führen, daß die Menge der verifizierbaren Bytecodeprogramme so weit wie möglich der Menge der Bytecodeprogramme aus Java-Quellcode entspricht, weil so für einen Angreifer die Möglichkeit reduziert wird, Bytecode zu konstruieren, der die Sicherheitsvorkehrungen der Sprache selbst und der des `SecurityManager`s zu unterläuft.

(4) Laufzeitintegrität

Während die ersten drei Prüfungen stattfinden, nachdem eine Klasse geladen ist, findet eine weitere Integritätsprüfung zur Laufzeit statt. Ein Beispiel soll diese Prüfung illustrieren. Eine Klasse `Unterklasse` erbt von einer weiteren Klasse `Oberklasse`. Eine dritte Klasse `Test` implementiert zwei Methoden, `getInstanceUnterklasse()` und `getInstanceOberklasse()`, die jeweils Referenzen auf Objekte der ersten beiden Klassen zurückgeben. Gegeben sei nun folgender Codeausschnitt:

```
Test x = new Test();
Oberklasse y = x.getInstanceUnterklasse();
```

Um die Typüberprüfung der letzen Anweisung vorzunehmen müßte jedoch die Unterklasse zunächst geladen werden, um festzustellen, daß sie von Oberklasse erbt. Diese Prüfung findet zur Laufzeit statt, denn Klassen sollen erst geladen werden, wenn sie tatsächlich benötigt werden, also eine Methode einer Klasse aufgerufen, oder ein Feld einer Klasse geändert werden soll.

Security Manager

Seit Java2 ist die Klasse `java.lang.SecurityManager` nicht länger eine abstrakte Klasse, und die Implementierung unterstützt das neue politikbasierte Sicherheitsmodell. Will eine Anwendung feststellen, ob sie ein Recht zur Ausführung einer bestimmten Aktion besitzt, ruft sie die Methode `checkPermission` des installierten `SecurityManager`s auf, und übergibt ein Objekt der Klasse `java.security.Permission`. Ein Codebeispiel hierfür zeigt Abbildung 11.

```
SecurityManager security = System.getSecurityManager();
if (security != null) {
    security.checkPermission(new FilePermission(“foo.txt”, “read”));
}
```

Abbildung 11 - Prüfung, ob ein Leserecht auf "foo.txt" besteht

Zugriffe auf die Systemressourcen sind dabei nur über die Kernklassen von Java realisierbar. Diese Klassen führen solche Tests durch. Die Methode `checkPermission` stellt sicher, daß die protection domain der Klasse das Recht hat, was im übergebenen Permission-Objekt spezifiziert ist. Diese Prüfung erfolgt in der Standardimplementierung des `SecuityManager`s der J2SE auf Basis eines policy files, das weiter unten beschrieben wird.

Obwohl jede Klasse die Klasse `java.lang.SecurityManager` erweitern kann, so erlaubt die JVM lediglich nur einen aktiven `SecurityManager`. Dieser kann von einem Programm durch den Aufruf von `System.setSecurityManager()` gesetzt werden. Weiterhin kann ein Security`SecurityManager` durch den Parameter `–Djava.security.manager` von `java` oder `javaw` installiert werden. Die Standardimplementierung des

`SecurityManager` des J2SDK wird dabei beim Aufruf des Programms `HelloWorld` durch eine der folgenden Aufrufvarianten installiert:

- `java –Djava.security.manager HelloWorld`
- `java –Djava.security.manager=“” HelloWorld`
- `java –Djava.security.manager=default HelloWorld`

Eine andere Implementierung eines `SecurityManager`s (z.B. als Klasse `CustomSecMgr`) kann wie folgt installiert werden:

- `java –Djava.security.manager=CustomSecMgr HelloWorld`

Ein einmal aktiver `SecurityManager` kann von einem Programm zur Laufzeit der JVM nur durch den Aufruf von `System.setSecurityManager()` geändert werden, wenn es die Rechte zur Instanziierung eines `SecurityManager`s oder zur Installation eines `SecurityManager`s hat.

Policy File

Die Standardimplementierung des J2SE sieht vor, daß eine Sicherheitspolitik dateibasiert realisiert wird [PRGN99 S. 80, 93, 242]. Dabei werden diese policy files standardmäßig in der Datei `java.security` spezifiziert. Diese Datei findet sich in `${java.home}${/}lib${/}security` und legt die einstellbaren Eigenschaften des Sicherheitsmodells fest. Die Vorgabe in dieser Datei lautet, daß ein systemweites policy file und ein vom Anwender definiertes policy file zur Sicherheitspolitik kombiniert werden. Die Einträge entsprechen einem URL und können somit auch entfernte policy files angeben.

```
# The default is to have a single system-wide policy file,
# and a policy file in the user's home directory.
policy.url.1=file:${java.home}/lib/security/java.policy
policy.url.2=file:${user.home}/.java.policy
```

Abbildung 12 - java.security: Standard policy files

Angaben über die Anwendung von policy files können zudem über den Aufrufparameter `–Djava.security.policy` von `java` oder `javaw` erfolgen. Dabei bedeutet der Aufruf

```
java –Djava.security.manager –Djava.security.policy=MyPolicy HelloWorld
```

daß das policy file `MyPolicy` zusätzlich zu den policy files aus der Datei `java.security` verwandt wird, und der Aufruf

```
java –Djava.security.manager –Djava.security.policy==MyPolicy HelloWorld
```

daß ausschließlich das policy file `MyPolicy` verwendet werden soll. Auch in diesem Aufrufparameter ist die Angabe eines entfernten policy files über einen URL möglich.

Ein policy file enthält neben einer *keystore Direktive* einen oder mehrere *grant-Einträge*. Die Syntax eines solchen Eintrages ist in Abbildung 13 - Der grant-Eintrag eines policy files – wiedergegeben[16].

```
grant [signedBy signers]
      [, codeBase URL]
{
  permission permission_class [target] [, action] [, signedBy signers];
  [permission ...]
};
```

Abbildung 13 - Der grant-Eintrag eines policy files

Rechte werden in Java2 durch verschiedene Klassen abgebildet, die alle von der abstrakten Klasse `java.security.Permission` abgeleitet sind. Spezifische Rechte werden in den packages spezifiziert, in denen sie am häufigsten benötigt werden. So ist etwa die Klasse `FilePermission` im package `java.io` zu finden. Meist werden die Rechte als Instanzen dieser Klassen über Konstruktoren mit zwei Parametern gebildet, von denen der erste das Ziel („*target*“) des Rechtes beschreibt (z. B. eine Datei) und der zweite die Aktion („*action*“), die auf diesem Ziel erlaubt ist (z. B. lesen, schreiben, ...). Die Klasse `java.sequrity.AllPermission` stellt eine spezielle Klasse dar, die alle anderen Rechte impliziert. Der in Fußnote 15 (Seite 35) erwähnte Eintrag für die Java-Erweiterungen ist in Abbildung 14 wiedergegeben.

[16] Die Schlüsselwörter sind dabei – für Java untypisch – nicht case-sensitiv.

```
// Standard extensions get all permissions by default

grant codeBase "file:${java.home}/lib/ext/*" {
  permission java.security.AllPermission;
};
```

Abbildung 14 - Policy file Eintrag für die Java Extensions

Die Angaben unter `codeBase` und `signedBy` eines jeden grant-Eintrages definieren ein in Abschnitt 4.1.2 besprochenes `CodeSource`-Objekt. Die beiden Felder bedeuten im einzelnen:

`codeBase`

Hinter dem Schlüsselwort `codeBase` wird ein URL angegeben, der aussagt, von wo der Code geladen wird, dem die Rechte zugestanden werden sollen. Im URL können dabei Platzhalter (* oder -) zur Verkürzung benutzt werden (Siehe Abbildung 14).

Die Angabe dieses Feldes ist optional. Wenn es ausgelassen wird, werden die Rechte jedem Code unabhängig seines Ortes zugestanden.

Beispiel: `codeBase “http://www.ThorstenFischer.com“`

`signedBy`

Mit diesem Feld werden die Zertifikate bezeichnet, die zur Signierung des Codes verwendet werden müssen, wenn die Rechte zugestanden werden sollen. Dabei werden die Namen angegeben, die mit den Zertifikaten verknüpft sind. Mehrere Zertifikate sind dabei durch Komma zu trennen. Die Rechte werden in diesem Fall nur zugestanden, wenn alle angegebenen Zertifikatsinhaber den Code signiert haben.

Die Angabe dieses Feldes ist optional. Wenn es ausgelassen wird, werden die Rechte jedem Code unabhängig seiner Signierung zugestanden, auch wenn er gar nicht signiert wurde.

Beispiel: `signedBy “Thorsten, Simone“`

Rechte können anhand von bis zu fünf Feldern definiert werden. Diese bedeuten im einzelnen:

`permission`

Die Angabe des Schlüsselwortes permission ist für die Einräumung jedes Rechtes erforderlich.

permission_class

Die Angabe des voll qualifizierten Namens der `Permission` Klasse ist erforderlich.

Beispiel: `java.util.PropertyPermission`

target

Ein in Anführungszeichen eingeschlossener String, der das Ziel des Rechtes bezeichnet.

Die Angabe dieses Feldes hängt von der *`permission_class`* ab.

Beispiel (für das Ziel einer `PropertyPermission`): `“java.version”`

action

Ein in Anführungszeichen eingeschlossener String, der die Aktionen darstellt, die auf dem Ziel dieses Recht ausmacht. Mehrere Aktionen können durch Komma getrennt angegeben werden.

Die Angabe dieses Feldes hängt von der *`permission_class`* ab.

Beispiel (für eine `java.net.SocketPermission`): `“resolve,accept,listen“`

signedBy

Dieses Feld bezeichnet den Namen eines digitalen Zertifikats, mit dem die mit *`permission_class`* angegebene Klasse signiert sein muß. Dabei ist nur ein Zertifikat erlaubt.

Dieses Feld ist optional. Um Spoofingangriffen vorzubeugen kann das Signieren der *`permission_class`* nötig sein.

Die Syntax eines jeden grant-Eintrages ist genau einzuhalten, ansonsten entstehen unvorhersehbare Folgen, welche das komplette Sicherheitsmodell beeinflussen können. Das zum J2SDK gehörige Werkzeug `policytool` hilft, gültige grant-Einträge zu formulieren. Dieses Tool wird in nachfolgenden Versionen auch dann wichtig, wenn die policy files verschlüsselt werden oder nicht in Klartext abgelegt werden.

In der Eigenschaftendatei `java.security` kann über den Schalter `policy.expandProperties` festgelegt werden, wie die Dateipfad-Trennzeichen

angegeben werden. Ist dieser Schalter, wie in der Voreinstellung in Abbildung 15, auf `true` gesetzt, so können plattform- oder installationsspezifische Angaben über die ${}-Notation getroffen werden. Sonst müssen alle Plattformspezifika genau eingehalten werden. Beispielsweise ist der Eintrag `C:\\java\\policies\\file1.policy` nur für ein Windows-System gültig. Darin wird ein Backslash als Einleitung einer Excape-Sequenz angesehen. Deshalb sind zwei Backslashes zur Angabe eines einzelnen nötig. Der entsprechende plattformunabhängige Eintrag lautet `C:${/}java${/}policies${/}file1.policy`

```
# whether or not we expand properties in the policy file
# if this is set to false, properties (${...}) will not be expanded in policy
# files.
policy.expandProperties=true
```

Abbildung 15 - java.security: Der Schalter policy.expandProperties

Neben einem oder mehreren grant-Einträgen kann ein policy file eine *keystore Direktive* enthalten. Dieser Eintrag ist notwendig, wenn in den grant-Einträgen digitale Signaturen angesprochen werden sollen. Die Syntax dieser Direktive zeigt Abbildung 16 - Die Syntax der keystore-Direktive.

```
keystore "URL" "type"
```

Abbildung 16 - Die Syntax der keystore-Direktive

Im Feld URL kann der Ort des keystore files angegeben werden, das Feld *type* ist optional und bezeichnet den Typ dieser Datei. Die Vorgabeeinstellung ist dafür in der Datei `java.security` mit `jks` getroffen, womit die proprietäre Implementierung von Sun benutzt wird.

Wie die Diskussion des policy files zeigt, ist darin das Konzept der negativen Rechte nicht realisiert. Grundsätzlich ist alles verboten, was nicht explizit gestattet wurde.

Protokollierung von sicherheitsrelevanten Vorgängen

Die Application-Launcher `java` und `javaw` stellen mit dem Aufrufparameter `Djava.security.debug` eine Möglichkeit bereit, alle Zugriffe auf Systemressourcen während der Ausführung eines Java-Programms verfolgen zu können. Dabei kann man zwischen verschiedenen Stufen der Protokollierung wählen. Mit dem Befehl

```
java –Djava.security.debug=help
```

sind detailliertere Informationen über diese Möglichkeit aufrufbar.

4.2.1.3 Schnittstellen und Architekturen

Zur Kommunikation mit nicht-Java-Anwendungen stellt Java Schnittstellen und Architekturen bereit, die dies auf einfache Art erlauben. Einige wichtige sollen hier unter Sicherheitsgesichtspunkten betrachtet werden.

Java Native Interface (JNI)

Über das JNI kann man Bibliotheken nutzen, die in anderen Sprachen implementiert worden sind und die als plattformabhängiger Code kompiliert vorliegen. Allerdings bringt dies die Nachteile mit, daß die Plattformabhängigkeit der gesamten Anwendung an dieser Stelle verloren geht, daß diese Codeteile auf den Rechnern, auf dem sie ausgeführt werden sollen installiert sein müssen, und daß Java die Prüfungen des Klassenladers und des Class File Verifiers nicht für solchen nativen Code durchführen kann.

Java Beans

JavaBeans stellen Softwarekomponenten dar, die folgende Eigenschaften besitzen [PRGN99 S. 43]:

- Sie erlauben einer Integrierten Entwicklungsumgebung das Analysieren ihres Codes (*introspection*) und das Anpassen ihres Aussehens (GUI) und Verhaltens.
- Sie unterstützen *events* (ein Kommunikationsmodell), Eigenschaften, die zur Entwurfs- und/oder Laufzeit über Methodenaufrufe geändert werden können, und Persistenz.
- Sie stellen Schnittstellen zu weiteren Komponentenarchitekturen wie z. B. ActiveX bereit.

Der letztgenannte Punkt ist wiederum aus Sicherheitsüberlegungen problematisch, denn ein JavaBean kann über diese Schnittstellen Code einbeziehen, der weniger strengen Kontrollen unterliegt, als das Bean selbst.

Remote Method Invocation (RMI)

RMI erlaubt die Verteilung von Java-Anwendungen dahingehend, daß ein Mechanismus zur Verfügung gestellt wird, mit dem entfernte Java-Methodenaufrufe reali-

siert werden können. Dabei ist bei der Erstellung des Servers zu bedenken, daß zunächst nicht bekannt ist, wer einen Methodenaufruf vornimmt. Die Gültigkeit der Anfragen muß somit überprüft werden.

RMI ist mit Einführung der Java2 Plattform dahingehend verbessert worden. So kann RMI auch über frei wählbare Kommunikationsprotokolle, wie etwa SSL realisiert werden.

Java Database Connectiviry (JDBC)

JDBC stellt ein API dar, mit dem von Java-Anwendungen aus über SQL-Statements auf Datenbanken zugegriffen werden kann. Dabei ist zu beachten, daß die Zugriffskontrolle der Datenbank nicht im Java-Sicherheitsmodell realisiert sein kann, sondern in der Datenbank selbst behandelt werden muß.

4.2.1.4 Das Zusammenspiel der Elemente von Java

Das Beispiel einer kleinen Anwendung nach [PRGN99 S. 26] soll das Zusammenspiel der Elemente von Java verdeutlichen. Der Anwender hat dabei nicht volles Vertrauen in eine Java-Applikation, die er sich über das Internet beschafft hat. Der Programmierer behauptet zwar, daß der Code lediglich etwas ausdruckt, aber der Anwender hat nicht volles Vertrauen in diese Zusicherung. Er möchte deshalb der Anwendung lediglich das Drucken auf seinem System gestatten, alle anderen Zugriffe auf Systemressourcen sollen gesperrt sein. Das folgende Beispiel zeigt, daß zur Verwirklichung dieser Sicherheitsanforderung in Java keinerlei Programmieraufwand nötig ist.

Die Anwendung besteht aus dem Code in Abbildung 17, aber nur die Klassendatei liegt dem Anwender vor.

```
import java.awt.*;
import java.awt.event.*;

class GetPrintJob extends Frame implements ActionListener {
   boolean p = true;

   GetPrintJob() {
      super("Toolkit.getPrintJob() Test");
      setSize(300, 100);
      setLocation(200, 200);
      Button b = new Button("getPrintJob");
      add(b, BorderLayout.CENTER);
      b.addActionListener(this);

      show();
```

```
        }

        public void actionPerformed(ActionEvent evt)  {
           try {
              // über diesen PrintJob könnte gedruckt werden.
              Toolkit.getDefaultToolkit().getPrintJob(this, "PrintJob", null);
           } catch(Exception e) {
              System.out.println(" Eine Exception ist aufgetreten: "+ e.getMessage());
              e.printStackTrace();
              p=false;
           }
           if (p)
              System.out.println(" Keine Exception. Drucken erlaubt.");
           System.exit(0);
        }

        public static void main(String[] args) {
           new GetPrintJob();
        }

     }
```

Abbildung 17 - Die Anwendung GetPrintJob

Wenn der Anwender die Anwendung mit dem Befehl

`java GetPrintJob`

startet, so wird kein `SecurityManager` angewendet, die Anwendung hat dann als lokale Applikation volle Zugriffsrechte. Deshalb startet der Anwender das Programm mit dem Befehl

`java –Djava.security.manager GetPrintJob`

Jetzt wird der `SecurityManager` der Standardimplementierung von Sun angewendet. Das Fenster des Programms erhält den Warnhinweis „Achtung: Applet Fenster“, um darauf hinzuweisen, daß ein `SecurityManager` angewendet wird. Doch die Anwendung hat noch kein Recht, auf den Drucker zuzugreifen. Sie endet also erfolglos.

Damit die Anwendung das Recht erhält, fügt der Anwender entweder per Hand oder über das `policytool` folgenden grant-Eintrag in die Datei `security.policy` ein:

```
grant codeBase "file:/C:${/}Programme${/}Untrust${/}GetPrintJob" {
  permission java.lang.RuntimePermission "queuePrintJob";
};
```

Damit hat der Anwender sichergestellt, daß er nur dieser einen Klasse, welche die fragliche Anwendung ausmacht, das Druckrecht zugesteht[17].

[17] Dies gilt unter der Voraussetzung, daß sie die einzige Klasse in dem Verzeichnis `C:${/}Programme${/}Untrust${/}GetPrintJob` ist.

Da die Anwendung wie versprochen funktioniert und auch weiterhin nichts negatives über den Programmierer bekannt wird, entschließt sich der Anwender, allen Programmen dieses einen Programmierers Druckrechte einzuräumen.

Der Programmierer stellt seine Programme als signierte JAR-Archive bereit. Im folgenden wird gezeigt, wie er solche erstellt und welche weiteren Schritte zur Ausführung des signierten Programmes nötig sind.

(1) Jar-Archiv erstellen

Mit dem Befehl

```
jar cvfm GetPrintJob.jar MainClass.txt GetPrintJob.class
```

erstellt der Programmierer ein ausführbares Jar-Archiv, denn in der Manifestdatei `MainClass.txt` wird über den Eintrag „`Main-Class: GetPrintJob`“ die Klasse bezeichnet, deren `main -` Methode bei der Ausführung des Archivs gestartet werden soll.

(2) Erstellen einer keystore-Datei und eines asymmetrischen Schlüsselpaares für den Programmierer

Um ein asymmetrisches Schlüsselpaar zu erstellen, und dieses in eine geschützte keystore-Datei abzuspeichern, ruft der Programmierer das Werkzeug `keytool` auf und macht folgende Eingaben:

```
C:\>keytool -genkey
Enter keystore password:  tfpasswort
What is your first and last name?
  [Unknown]:  Thorsten Fischer
What is the name of your organizational unit?
  [Unknown]:  Anwendungsentwicklung
What is the name of your organization?
  [Unknown]:  ThorstenFischer.com
What is the name of your City or Locality?
  [Unknown]:  Duesseldorf
What is the name of your State or Province?
  [Unknown]:  NRW
What is the two-letter country code for this unit?
  [Unknown]:  DE
Is <CN=Thorsten Fischer, OU=Anwendungsentwicklung, O=ThorstenFischer.com, L=Dues
seldorf, ST=NRW, C=DE> correct?
  [no]:  yes

Enter key password for <mykey>
        (RETURN if same as keystore password):  mykeypasswort
```

Die keystore-Datei findet sich anschließend als `${user.home}${/}.keystore`.

(3) Jar-Archiv signieren

Mit dem generierten Schlüsselpaar, das er über „`mykey`“ ansprechen kann, signiert er nun das zuvor erstellte Jar-Archiv, bildet also einen Hash-Wert über den Inhalt des Archivs und verschlüsselt diesen mit seinem privaten Schlüssel:

```
jarsigner GetPrintJob.jar mykey
```

(4) Exportieren des Zertifikats des Programmierers

Damit der Anwender den Programmierer genau identifizieren kann, stellt jener diesem sein Zertifikat über einen sicheren Kanal (Diskette per Einschreiben) zur Verfügung. Das Zertifikat exportiert er aus seiner keystore-Datei in die Datei `TFCert.cer` mit dem Befehl

```
keytool –export –alias mykey –file TFCert.cer
```

(5) Importieren des Zertifikats des Programmierers durch den Anwender

Der Anwender nimmt das Zertifikat des Programmierers unter dem Namen *tfkey* in seine keystore-Datei auf. Dazu benutzt er den Befehl

```
keytool –import –alias tfkey –file TFCert.cer
```

Dabei muß der Anwender bestätigen, daß er dem Zertifikat traut, womit er bestätigt, daß der öffentliche Schlüssel tatsächlich zu der Person des Programmierers gehört. Deshalb ist ein vertrauenswürdiger Austausch des Zertifikates nötig.

(6) Eintragen der Rechte für das Programm und Starten des Programms

Da der Anwender nun einen grant-Eintrag festlegen will, der sich auf dieses importierte Zertifikat bezieht, muß er die keystore-Direktive in die Datei `java.policy` aufnehmen:

```
keystore "file:/C:${/}WINDOWS${/}.keystore";
```

Nun kann der Anwender mit dem grant-Eintrag

```
grant signedBy "tfkey" {
  permission java.lang.RuntimePermission "queuePrintJob";
};
```

allen Programmen, die der Programmierer digital signiert hat, Druckrechte einräumen, und das Programm mit dem Befehl

```
java –Djava.security.manager –jar GetPrintJob.jar
```

starten.

4.2.2 JAAS

Wie schon in Abschnitt 4.1.2 unten angemerkt ist, kann zwischen anwenderbasierter und codebasierter Zugriffskontrolle unterschieden werden. Abschnitt 4.2.1 beschreibt die Umsetzung des codebasierten Zugriffskontrollmodells. Hier soll nun das „*Java Authentication and Authorization Service* (*JAAS*)“-Framework [GKLN99] vorgestellt werden, mit dem ein RBAC-Modell als anwenderbasiertes Zugriffskontrollmodell realisiert werden kann. Eine Implementierung dieses Frameworks von Sun liegt als Java-Erweiterung vor.

4.2.2.1 Konzepte

Zunächst definiert das Framework den Begriff *subject* als jeden Nutzer eines Dienstes. Damit sind sowohl Anwender als auch andere Dienste eingeschlossen. Damit ein subject von einem Dienst identifiziert werden kann, wird ihm ein Name zugeordnet. Dieser Name wird als *principal* bezeichnet. Weil subjects für unterschiedliche Dienste unterschiedliche Namen tragen können, ist diese Zuordnung eine 1:n Zuordnung. Dies zeigt sich auch in der Implementierung:

```
public interface Principal { public String getName(); }
public final class Subject { public Set getPrincipals() {} }
```

Principals werden einem subject zugeordnet, wenn sie sich erfolgreich gegenüber dem subject authentisieren. Authentisierung ist demnach der Prozeß, in dem ein subject die Identität eines anderen subjects verläßlich überprüft.

Einige Dienste ordnen subjects nicht nur principals, sondern weitere sicherheitsrelevante Daten zu. Diese Daten werden in JAAS als *credentials* bezeichnet und entweder als vertraulich oder öffentlich klassifiziert. Demnach enthält die Implementierung der Klasse Subject noch weitere Methoden:

```
public final class Subject {
    // not security checked
    public Set getPublicCredentials();
    // security checked
    public Set getPrivateCredentials();
}
```

4.2.2.2 Authentisierung

Ein zu nutzender Dienst kann unterschiedliche Arten der Authentisierung erfordern. Deshalb basiert das JAAS auf dem *Pluggable Authentication Modules* (PAM)-Framework [SUNf]. PAM unterstützt eine Architektur, die es Systemadministratoren erlaubt, angemessene Authentisierungsdienste zu wählen, die ihren Sicherheitsanforderungen entsprechen. Ändern sich diese Sicherheitsanforderungen, kann auch nur der Authentisierungsdienst ausgetauscht werden, ohne die komplette Anwendung ändern zu müssen. Die JAAS-Klasse `LoginContext` stellt eine Implementierung dieses PAM-Frameworks dar und ermittelt über eine Konfiguration den Authentisierungsdienst, der als Interface `LoginModule` implementiert ist. Abbildung 18 - Die JAAS-Implementierung des PAM – zeigt, daß die Authentisierung durch einen Zwei-Phasenprozeß erfolgt.

```
public final class LoginContext {
    public LoginContext(String name) {}
    public void login() {} // Zwei-Phasen Prozeß
    public void logout() {}
}

public interface LoginModule {
    boolean login(); // erste Phase
    boolean commit(); // zweite Phase
    boolean abort();
    boolean logout();
}
```

Abbildung 18 - Die JAAS-Implementierung des PAM

4.2.2.3 Autorisierung

Die Autorisierung erfolgt in der Standardimplementierung von JAAS durch SUN in Version 1.0 principalbasiert. Dabei wird die in Abschnitt 4.2.1.2 vorstellte Syntax eines grant-Eintrages aus Abbildung 13 (Seite 40) im policy file zu der in Abbildung 19 erweitert [SUNg].

```
grant [signedBy signers]
      [, codeBase URL]
      [, principal [principal_class] [principal_name]]
      [, principal ...]
{
    permission permission_class [target] [, action] [, signedBy signers];
    [permission ...]
};
```

Abbildung 19 - Der grant-Eintrag eines JAAS policy files

In dieser Syntax ist `principal` ist ein erforderliches Schlüsselwort, *`principal_class`* der voll qualifizierte Name der `Principal` Klasse und *`principal_name`* der in Anführungszeichen eingeschlossene Name, der von dieser Klasse zurückgegeben werden soll. Aus der Syntax folgt, daß alle für die Standardimplementierung des policy files erstellten Dateien auch unter JAAS gültig sind.

Das Recht wird dann zugestanden, wenn der Code aus dem im codeBase Feld benannten Ort stammt und er von allen im signedBy Feld benannten Personen signiert ist, und das subject, das den Code ausführt, alle principals enthält, die im principal Feld spezifiziert sind. Fehlt eines der optionalen Felder, so werden für die Information dieses Feldes keine Sicherheitseinschränkungen vorgenommen.

Mit JAAS können auch nach RBAC rollenbasierte oder gruppenbasierte Zugriffskontrollmechanismen realisiert werden, wenn die entsprechenden Rollen oder Gruppen als principals formuliert werden. Abbildung 20 zeigt entsprechende grant-Einträge.

```
// Ein Administrator hat Zugang zu Passwörtern
grant Principal com.thorstenfischer.jaas.Role “administrator“ {
    permission java.io.FilePermission “/passwords/-“, “read, write“;
};

// Eine Fußballmannschaft (Gruppe) kann ihr Verzeichnis lesen
grant Principal com.thorstenfischer.jaas.Group “eintrachtfrankfurt“ {
    permission java.io.FilePermission “/teams/eintrachtfrankfurt/-“, “read“
}
```

Abbildung 20 - Rollen- oder gruppenbasierte Zugriffskontrolle mit JAAS

4.2.2.4 JRBAC99

[Giur99] stellt mit JRBAC99 ein Framework vor, das JAAS benutzt, um RBAC-Modelle in Java zu implementieren. Dazu werden nur relativ wenige Regeln benötigt, um die Begrifflichkeiten aneinander anzupassen. Weiterhin wird demonstriert, daß auch eigene Implementierungen eines policy files dazu benutzt werden können, eine genaue Abbildung von RBAC in Java vorzunehmen.

5 Angriffe auf die Java2-Sicherheit am Beispiel eines Applets

Nur ein Sicherheitsmodell, das offen diskutiert werden kann, hat die Chance ein gutes Sicherheitsmodell zu werden, denn nur so können Lücken erkannt und geschlossen werden. In den Anfängen der Sprache Java konnte man oft den Rat lesen, daß es sicher sei, in Browsern Java zu deaktivieren, also die Ausführung von Applets zu ver-

hindern, weil Sicherheitslücken aufgetaucht waren. [McFe97] stellt ein ganzes Buch voller solcher Sicherheitslücken (der Version 1.0) dar, und auch heute tauchen immer wieder Meldungen über erfolgreiche Attacken gegen Java-Systeme auf, doch diese werden immer weniger und betreffen meist die konkrete Implementierung von Java-Komponenten in einzelnen Browsern. Auch muß allen, die Java aus Sicherheitsgründen in Browsern deaktivieren, entgegengehalten werden, daß sie dann erst recht auch keine anderen ausführbaren Inhalte von zweifelhafter Quelle ausführen dürfen. Dazu gehören zum Beispiel kleine Spaßprogramme oder Bildschirmschoner aus dem Internet oder Mailanhänge mit aktivem Inhalt. Dieser Abschnitt soll in Anlehnung an [PRGN99 S. 60] mögliche Angriffspunkte zeigen, die dazu führen könnten, daß in einem Applet unerwünschte Funktionen implementiert werden. Das Java2 Sicherheitsmodell stellt jedoch gegen alle diese Angriffspunkte Verteidigungsmaßnahmen bereit, die in diesem Artikel vorgestellt worden sind.

Das Beispiel verfolgt in Stichpunkten den Weg eines Applets von seiner Erstellung bis zur Ausführung und stellt dabei Fragen, die Angriffsmöglichkeiten illustrieren sollen.

- Erstellung des Quellcodes
  - Ist der Hersteller vertrauenswürdig und sein Code so, wie er es angibt?
  - Verfügbarkeit ist kein unwichtiger Sicherheitsdienst. Werden Qualitätskontrollen beachtet, die einen bestimmten Grad an Verfügbarkeit sicherstellen?
  - Java forciert die Wiederverwendung von Code. Können Entwickler die Sicherheit allen Codes überblicken, den sie einsetzen?
- Kompilierung des Quellcodes
  - Nutzen die Entwickler möglicherweise unbewußt veraltete Compiler, die Sicherheitslücken im Code eröffnen können?
  - Sind die Compiler der Entwickler vor Angriffen geschützt?
- Bereitstellung von Applets auf Webservern
  - Sichert der Anbieter eines Applets seinen Webserver gegen Angriffe genug ab, so daß Code nicht auf dem Webserver manipuliert werden kann?
  - Java erleichtert den plattformunabhängigen Austausch von Code. Können Webmaster sicher sein, daß sie nicht unbewußt bösartigen Code einsetzen?

- IP-Spoofing
  - Verbergen nicht Dritte die tatsächliche Herkunft eines Applets durch Man-In-The-Middle-Attacks und schieben Code unter, der gar nicht geladen werden sollte?
- Angriffspunkte auf dem Client
  - Ist die JVM des Browsers so implementiert, daß das Sicherheitsmodell greift? Ist die Implementierung gegen Angriffe geschützt?

6 Zusammenfassung

Eine Stärke von Java2 und ein Vorteil gegenüber anderen Sprachen ist das Sicherheitsmodell, das in den Entwurf der Sprache integriert wurde. Mit ihm ist der Aufbau sicherer Systeme deshalb sehr gut möglich, weil es einen ganzheitlichen Ansatz auch zur Integration seines Umfeldes bietet, weil es flexibel auf verschiedene Sicherheitsanforderungen anpaßbar ist, und deshalb angemessen und nachhaltig bleibt.

Jeder einzelne Sicherheitsdienst ist in Java realisierbar, vor allem die Authentisierung und Autorisierung wird weitgehend unterstützt.

7 Literaturverzeichnis

[Fran98] Franz, Michael; **„Java – Anmerkungen eines Wirth-Schülers“**; In: Informatik Spektrum 21/1998 S. 23-26; Springer Verlag

[GKLN99] Lai, Charlie; Gong, Li; Koved, Larry; Nadalin, Anthony; Schmers, Roland; **„User Authentication and Authorization in the JavaTM Platform“**; In: Proceedings of the 15th Annual Computer Security Applications Conference, Phoenix, AZ, December 1999; http://java.sun.com/security/jaas/doc/acsac.html; Stand: 03.01.2001

[Giur99] Giuri, Luigi; **„Role-Based Access Controls on the Web using Java“**; In: Proceedings of the fourth ACM Workshop on role-based access control (S. 11-18); Fairfax, VA USA, October 28-29, 1999

[GMPS97] Gong, Li; Mueller, M.; Prafullchandra, H.; Schemers, R.; **„Going Beyond the Sandbox: An Overwiew of the New Security Architecture in the JAVA Development Kit 1.2“** In: Proceedings of the Usenix Symposium in Internet Technologies and Systems, 12/1997, Monterey, California

[Gong98] Gong, Li; **„Java2 Platform Security Architecture“**; http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc.html; Stand: 29.12.2000

[LiYe99] Lindholm, Tim; Yellin, Frank; **„The Java™ Virtual Machine Specification Second Edition“**; Addison-Wesley, Menlo Park, California 1999

[McFe97] McGraw, Gary; Felten, Edward W.; **„Java Security Hostile Applets, Holes and Antidotes“** Wiley Computer Publishing, New York 1997

[PRGN99] Pistoia, Marco; Reller, Duane F.; Gupta, Deepak; Nagnur, Milind; Ramani, Ashok K.; **„Java 2 Network Security“ 2nd Edition“**, Prentice Hall PTR, New Jersey

[Sand98] Sandhu, Ravi S.; **„Role-Based Access Control“**; In: Zelkowitz, Editor, Advances in Computers, Volume: 46. Academic Press, 1998

[SCFY96] Sandhu, Ravi S.; Coyne, Edward J.; Feinstein, Hal L.; Youman, Charles E.; **„Role-Based Access Control Models“**; IEEE Computer, 29(2):38-47, February 1996

[SUNa] **The Java Tutorial**; http://java.sun.com/docs/books/tutorial/index.html Stand: 13.10.2000

[SUNb] **Frequently Asked Questions - Applet Security**; http://java.sun.com/sfaq/ Stand: 11.12.2000

[SUNc] **A Note on the "Java™ 2" Name**; http://java.sun.com/products/jdk/1.2/java2.html Stand: 22.12.2000

[SUNd] **Java™ 2 SDK Version 1.2 History of Changes**; http://java.sun.com/products/jdk/1.2/previous-changes.html; Stand: 11.12.2000

[SUNe] **Java™ Cryptography Architecture API Specification & Reference**; http://java.sun.com/products/jdk/1.2/docs/guide/security/CryptoSpec.html; Stand: 29.12.2000

[SUNf] Samar, Vipin; Lai, Charlie; **Making Login Services Independent of Authentication Technologies**; Sun Microsystems Inc.; http://java.sun.com/security/jaas/doc/pam.html; Stand: 08.11.2000

[SUNg] **Java Authentication and Authorization Service (JAAS) 1.0 Developer‘s Guide**; http://java.sun.com/security/jaas/doc/api.html; Stand: 19.11.2000

[Venn98] Venners, Bill; **The Hotspot Virtual Machine**; http://www.artima.com/designtechniques/hotspot.html; Stand: 15.10.2000