Fachhochschule Bonn–Rhein–Sieg
Fachbereich Angewandte Informatik
Grantham-Allee 20
53757 Sankt Augustin

Verteilte und Parallele Systeme I
Prof. Dr. Rudolf Berrendorf

Java Security Architecture

am 10. Juni 2003
von
Michael Adams und Marco Krause

Inhaltsverzeichnis

1. Einleitung

Seit der Einführung von Java im Jahre 1995 hat sich die Sprache schneller verbreitet als jede andere Programmiersprache zuvor. Ein Grund hierfür liegt in der Plattformunabhängigkeit, weshalb sie insbesondere bei Internet-Programmierung verwendet werden kann.
Das Aufkommen einer solchen Technologie hat ein neues Problem bei der Nutzung des www gebracht. Bis dahin bestanden www-Seiten ausschließlich aus passiven Elementen wie z.B. Texten, Grafiken, Video- und Sounddaten. Diese Objekte stellten keine direkte Bedrohung für den Anwender dar, da sie nur vom Browser des Benutzers dargestellt werden, aber nicht aktive Kommandos auf dem Zielrechner ausführen können. [Horster_99]
Webseiten werden aber gerade deshalb interessant, wenn sie kleine Programme, so genannte Applets[1], enthalten. Java bietet nun die Möglichkeit solche Programme schnell und bequem zu erstellen. Üblicherweise befinden sich Applets auf dem Webserver – beim Aufruf der entsprechenden Webseite werden sie dann über das Netz auf den Rechner des Anwenders (Webserfer) geladen und dort lokal ausgeführt. Da der Urheber der Applets zumeist unbekannt ist und es sich somit um einen fremden Code handelt, der nicht unbedingt vertrauenswürdig ist, hat das Einbinden solcher Applets in Webseiten in Bezug auf den Sicherheitsaspekt besondere Konsequenzen. Aus diesem Grund haben die Entwickler von Java ein spezielles Sicherheitsmodell entworfen.

Das Sicherheitsmodell soll auch das Thema dieser Ausarbeitung mit dem Titel „Java Security Architecture“ sein.
Hierzu zeigen wir ihnen erst einmal die möglichen Bedrohungen auf, die sich unter anderem auch durch Applets auftun. Dann erklären wir das Java Sicherheitsmodell vor Java 1.2, das so genannte Sandkastenmodell und schließlich die neueren Entwicklungen des Sicherheits-Modells in Java 1.2.

[1] „App“ – von Applikation und „let“ stellt im Englischen eine Verkleinerungsform, wie im Deutschen das „chen“; Applet ist also ein Programmchen bzw. ein kleines Programm; siehe auch http://java.sun.com

2. Mögliche Bedrohungen

Prinzipiell sind die Bedrohungen, die durch das Laden von Java-Applets entstehen können ähnlich denen, die auch bei normalen Programmen bestehen. Werden Sie ohne Beschränkungen auf dem Zielsystem ausgeführt sind mehrere Gefährdungen möglich:

- Das Programm kann Viren enthalten, die das System infizieren und schädigen.
- Das Programm könnte als „Trojanisches Pferd“ unbekannt schädigende Nebeneffekte haben, die vom Entwickler eingebaut wurden.
- Das Programm kann vom Entwickler nicht beabsichtigte Nebeneffekte haben (z.B. durch Programmierfehler)

Die Auswirkungen solcher Gefährdungen können dabei verschiedener Art sein:

- Veränderung des Systems
- Einsicht in vertrauliche Daten des Benutzers
- Angriffe auf die Verfügbarkeit eines System (DoS)

Zum Schutz vor solchen Angriffen werden üblicherweise folgende verschiedene Methoden angewandt:

- Programme werden vor dem ersten Ausführen mit Hilfe von Virenscannern nach bekannten Mustern durchsucht.
- Programme werden zuerst in einer Test-Umgebung (z.B. ein Quarantäne-Rechner) ausgeführt, um zu untersuchen ob sie unerwünschte Funktionen ausführen.
- Es werden nur Programme von vertrauenswürdigen Quellen (z.B. renommierte Hersteller) auf dem System ausgeführt.

Handelt es sich um traditionelle Programme, die fest auf dem Computersystem installiert werden, sind diese Methoden recht wirkungsvoll.
Allerdings sind sie wenig geeignet für Java-Applets. Das Hauptproblem ist, dass Anwendungen, die auf dieser Topologie beruhen, im Gegensatz zu traditionellen Anwendungen nicht langfristig auf dem Zielsystem verbleiben, sondern jeweils beim Aufrufen neu auf das System geladen werden. Dabei ist es ohne weiteres möglich, dass beim erneuten Aufruf ein geänderter Code (neue Version oder Patch) geladen wird.
Des Weiteren lässt sich auch nicht einfach die Herkunft des Programms eindeutig feststellen und somit nur Programme aus vertrauenswürdigen Quellen ausführen. Dazu bedarf es weiterer Maßnahmen. Bei Virenscannern ist das Problem, dass nur solche Viren erkannt werden, die bereits bekannt sind. Die Hersteller von Virenscannern sind aber inzwischen kaum noch in der Lage, die Flut von neuen Viren zu bewältigen. Wir glauben somit, dass es nur eine Frage der Zeit ist, bis das eigene Computersystem von einer noch nicht bekannten Virenart infiziert wird.

Wie bereits erwähnt ist einer der Vorteile von Java die Plattformunabhängigkeit. Hier tut sich eine völlig neue Bedrohung auf. Galt bisher die Vielfalt an verschieden Plattformen, die am Internet angeschlossen sind als „natürliche Barriere“ so gilt dies bei Java nicht mehr. Die Plattformunabhängigkeit an sich stellt zwar kein Sicherheitsrisiko im eigentlichen Sinn dar, aber ein Angriff, der mit Java geführt werden würde, würde nahezu alle gängigen Plattformen gefährden.

3. Das Sandkastenmodell

3.1 Das Sprachmodell von Java

Das Sprachmodell von Java ist die erste Komponente zur Sicherheit. Java gilt als sichere Sprache, da sie schon eigens viele Kontrollen vornimmt und dem Programmierer erspart. Im Gegensatz zu z.B. C/C++ beinhaltet Java viele sicherheitsrelevante Merkmale, die Java so robust, tolerant zu Programmierfehlern und damit unanfälliger zu Seiteneffekten macht.

- **Sichtbarkeits-/Zugriffsbereiche**: private, protected, package protected, public
- **Strenge Typenprüfung**: Verboten sind casts zwischen primitiven Typen und Objekten und zwischen inkompatiblen Objekten
- **Referenzen statt Pointern**: Verbergen das eigentliche Memorylayout, keine Referenzarithmetik
- **Gültigkeit von Variablen**: Variablen müssen initialisiert sein bevor sie verwendet werden können
- **Final Entities**: Variablen dürfen nicht verändert, Methoden nicht überschrieben und Klassen nicht abgeleitet werden
- **Strenge Arraygrenzen**: Verhindern (versehentlichen) Zugriff auf benachbarte Entitäten
- **Exceptions:** Verhindern undefinierte Zustände der VM
- **Garbage Collection**: Verringert das Risiko von Memoryleaks
- **Virtuelle Maschine**: Verbirgt Besonderheiten und Gefahren der ausführenden Architektur und überwacht Programmausführung und Rechte
- **Java Bytecode**: Definiert lediglich Zugriffe auf VM und sonst keine anderen Ressourcen

3.2 Der Verifier (Bytecode Verifier)

Die JavaVirtualMachine legt nicht fest, dass der Bytecode nur vom Java Compiler erzeugt werden muss. Auch Compiler von anderen Sprachen wie Ada oder C, oder per Hand ist es möglich diesen Bytecode zu erzeugen.
Aus diesem Grund überprüft der Verifier den assemblerähnlichen Bytecode auf Korrektheit.
Dazu gibt es vier Punkte, die für die Korrektheit eines Bytecodes beachtet werden müssen:

- Für jeden Sprung liegt das Sprungziel in derselben Methode
- Sprungziele sind immer der Anfang eines Maschinenbefehls
- Ein von einem ExceptionHandler geschützter Block muss Anfang/Ende am Anfang einer Instruktion beinhalten.
- Der Code endet nicht mitten in einer Instruktion

Folgende Kontrollen werden von dem Verifier durchgeführt:

- **Class File Format Prüfung**
  Anwesenheit und Länge der benötigten Strukturen (Magic Number, Version Info, ConstantPool, Attribute)
- **Konsistenzprüfung**
  Final-Entities, Superklasse, Layout des ConstantPool, Gültigkeit von Referenznamen und –typen
- **Method Verification**
  Stackgröße, Parameterzahl von Opcodes und Methodenaufrufen, Objekte vor dem Zugriff definiert
- **Verzögerte Konsistenzprüfungen**
  Klassen nachladen, Existenz von Feldern/Methoden in referenzierten Klassen mit entsprechender Signatur, Arraylängen, NullReferenzen, Casts

3.3 Der ClassLoader

Die Aufgabe des ClassLoaders ist es, Klassen über das Netz zu laden und dabei zu verhindern, dass es zu Namenskonflikten und damit auch zu Typenkonflikten kommt. Insbesondere darf ein Applet keine Systemklassen wie z.B. java.io.FileInputStream oder java.lang.SecurityManager durch eigene Definitionen überschreiben. Auch darf es nicht zu Namenskonflikten zwischen Klassen verschiedener Applets kommen. Jedes Applet besitzt seinen eigenen Klassenlader, der die dazugehörigen Klassen in einem getrennten Namensraum installiert. Auch hierfür gibt es einen separaten Namensraum. Eine Klasse wird nun nicht mehr nur durch ihren Namen, sondern zusätzlich durch ihren Klassenlader eindeutig bestimmt, d.h. es wird das Paar (Klassenname, Klassenlader) zur eindeutigen Identifikation einer Klasse herangezogen.

Der ClassLoader stellt zusätzlich dem auszuführenden Applet alle erforderlichen Klassenbibliotheken zur Verfügung.[2] Von entscheidender Bedeutung ist dabei die Herkunft der Klassen, denn eine Klasse in einem lokalen Verzeichnis (spezifiziert in der Umgebungsvariable CLASSPATH) wird nicht denselben Kontrollen unterzogen wie eine Klasse, die über das Internet geladen wird. Sollte ein Applet eine bestimmte Klasse benötigen, so durchsucht der ClassLoader zuerst das lokale Verzeichnis und erst dann externe Quellen.

3.4 Der SecurityManager

Ein Java-Applet kann nur mit Hilfe der JVM auf Ressourcen des Betriebssystems zugreifen. Der Teil, mit dem die Zugriffskontrolle realisiert wird, ist der Security Manager (siehe Abbildung 1). Er ist seit der Version 1.0 im JDK enthalten. Die Basis-Klassenbibliothek ist so programmiert, dass der Sicherheits-Manager immer angefragt wird, bevor „gefährliche“ Operationen ausgeführt werden.

Die Operationen, die als gefährlich eingeschätzt werden und deshalb mit dem Security Manager kontrolliert werden können, sind:

- Netzwerkzugriffe
- das Laden von Applets
- Zugriffe auf Java-Klassen
- alle Operationen zum Manipulieren von und der Zugriff auf Threads
- der Zugriff auf Systemressourcen
- Zugriffe auf das Dateisystem
- das Aufrufen von lokalen Programmen und Betriebssystem-Kommandos

Welcher Zugriff dabei vom Security Manager auf Grundlage welcher Faktoren (z.B. Herkunft des Programms) gewährt wird, ist nicht im Java-Sicherheitsmodell festgelegt, sondern hängt von der jeweiligen Implementierung der Methoden des Security Managers ab.

[2] Java Applets werden üblicherweise nicht als ein komplettes Programm geladen, sondern die benötigten Java-Klassen werden bei Bedarf nachgeladen.

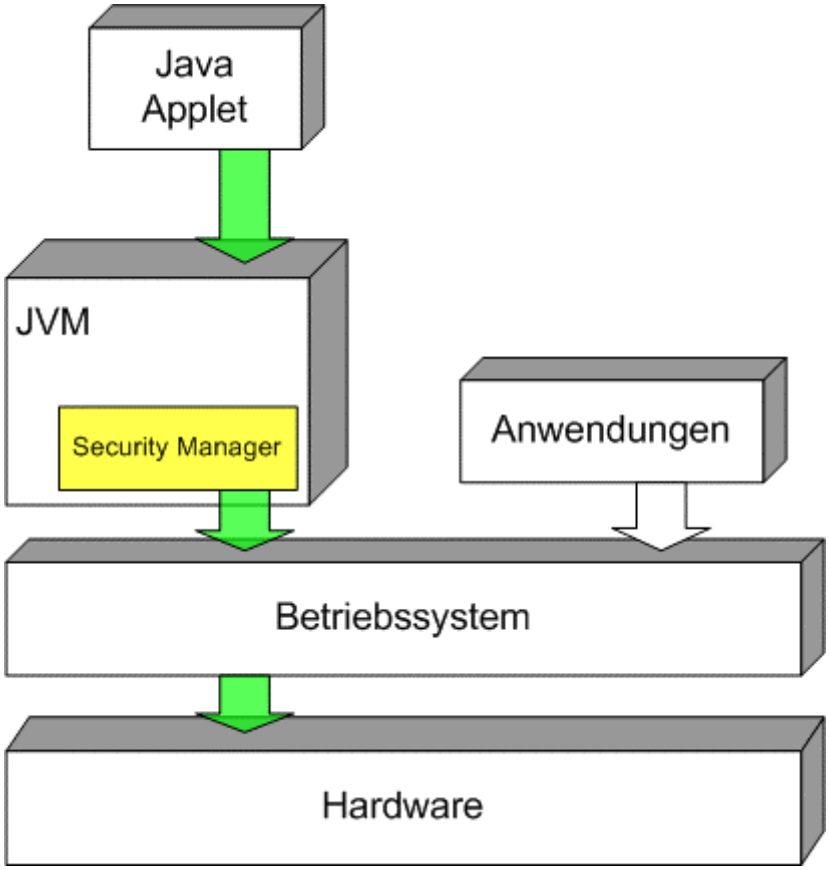

Abbildung 1

Konkret ist der Sicherheits-Manager ein Objekt, das für jede dieser Anfragen eine Methode exportiert, welche für die entsprechende Operation überprüft, ob sie ausgeführt werden darf oder nicht. So wird z.B. die Methode *public void checkDelete(String file)throws SecurityException* immer aufgerufen, bevor eine Datei gelöscht wird. Sie muss überprüfen, ob die als Parameter angegebene Datei gelöscht werden darf und wenn nicht eine Ausnahmebedingung (Exception) erzeugen, so dass die Operation abgebrochen wird. Jede Java-Applikation kann ihre eigene Sicherheitspolitik definieren, indem sie einen Sicherheits-Manager von der abstrakten Klasse SecurityManager ableitet, alle Methoden implementiert und beim Programmstart ein entsprechendes SecurityManager Objekt erzeugt. Ein java-fähiger Web-Browser kann also z.B. den Zugriff eines Applets auf das lokale Dateisystem verhindern, indem er vor dem Start des Applets einen Sicherheits-Manager erzeugt, der bei allen Dateioperationen eine SecurityException auslöst. Aus Gründen der Sicherheit kann ein einmal erzeugter Sicherheits-Manager nicht mehr ersetzt werden. Die momentan gebräuchlichen Web-Browser implementieren alle sehr restriktive Sicherheits-Manager, die bei allen Anfragen grundsätzlich eine Ausnahmebedingung auslösen, ohne vorher irgendwelche Tests durchzuführen.
Mit der Einführung von Java2 (auch bekannt als Java 1.2) wurde der Security Manager leicht modifiziert.
Beim Zugriff auf sicherheitskritische Faktoren wird er zwar auch weiterhin konsultiert, die Zugriffsregeln allerdings sind nicht mehr in ihm implementiert, sondern werden an den sogenannten Access Controller weitergeleitet.

4. Neuere Entwicklung des Sicherheitsmodells

In den ersten Implementierungen des Java-Sicherheitsmodells in Netscape oder Microsoft Browsern hatten die Benutzer 2 Möglichkeiten:

- Das Ausführen von Applets konnte generell unterbunden werden.
- Alle Applets wurden ausgeführt, allerdings mit sehr starken Einschränkungen

Es ist aber wünschenswerter, wenn es möglich wäre, abhängig von der Herkunft des Applets entweder die Ausführung zu unterbinden oder einem Applet weiterreichende Rechte einzuräumen.
Während also bei den Java-Versionen 1.0 und 1.1 in Bezug auf die Granularität der Zugriffskontrolle noch mehr oder weniger nach dem Motto „Alles oder Nichts“ verfahren wurde, hat sich dies ab der Version 1.2 entschieden verändert. Seit dem können zur Laufzeit differenzierte Rechte vergeben werden bzw. differenzierte Rechte können auch nach dem Laden noch verändert werden.

In den folgenden Kapiteln werden wir auf diese neueren Entwicklungen und die damit verbundene Rechteverwaltung, auf die Zuordnung und die Kontrolle der Rechte eingehen.

4.1 Digitale Signaturen

Das Sandkastenmodell der JDK Version 1.0 erwies sich als zu wenig flexibel; es waren hierdurch nicht einmal Standardanwendungen wie z.B. ein einfacher Texteditor als Applet möglich, da man hierzu die entsprechenden Schreib- bzw. Leserechte benötigt. Aus diesem Grund hat Sun das Sicherheitsmodell vom JDK 1.1 um kryptographische Methoden erweitert. Applets können mit einer digitalen Signatur versehen werden. Digitale Signaturen weisen ähnliche Eigenschaften wie Unterschriften auf, mit denen man Textdokumente unterzeichnet. Zu den Eigenschaften digitaler Signaturen gehören:

1. die Unveränderbarkeit des unterzeichneten Dokumentes
2. die Fälschungssicherheit der digitalen Signatur
3. die Identifizierbarkeit des Unterzeichners

Durch die Bedingung 1 garantieren digitale Signaturen die Datenintegrität des Applets, d.h. das Applet kann nach der Unterzeichnung nicht mehr verändert werden. Darüber hinaus sichern die Bedingungen 2 und 3 die Authentizität, d.h. die Identität des Unterzeichners kann eindeutig festgelegt werden. Mit Hilfe der digitalen Signatur kann der Anwender also entscheiden, ob er dem Applet alle Zugriffsrechte gewährt oder nicht, je nach dem ob er dem Unterzeichner vertraut oder nicht.
Hier wurde aber immer noch nach dem „Alles oder Nichts“ Prinzip gehandelt, bei dem ein Applet entweder alle oder aber nur die Rechte der Sandbox besitzt.

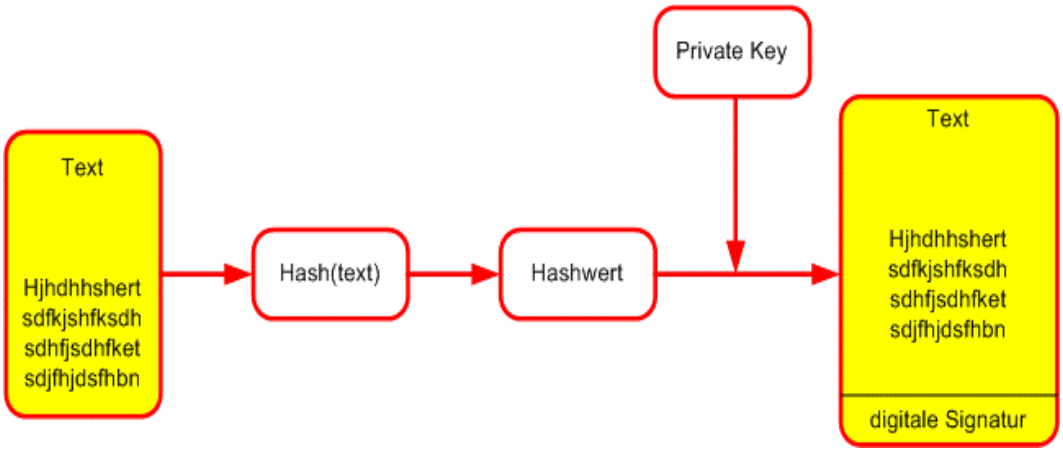

Abbildung 2

Für die digitale Unterzeichnung von Applets wird vom JDK der DAS (DigitalSignatureAlgorithm) oder RSA (Rivest-Shamir-Adleman) verwendet. Hierbei findet das Public-Key-Konzept seine Anwendung, wobei das Applet mit einem geheimen privaten Schlüssel unterzeichnet wird: Es wird von dem Appletcode zunächst mit Hilfe einer öffentlich bekannten Einweg-Hashfunktion, die MD5 oder SHA (SecureHashAlgorithm) verwendet, ein Hashwert gebildet. Dieser Hashwert wird dann mittels des privaten Schlüssels verschlüsselt (siehe Abbildung2). Die Verifikation der Unterschrift erfolgt mit dem öffentlichen Schlüssel, der zur Entschlüsselung des Hashwertes verwendet wird: Nach Entschlüsselung der Signatur muss man den ursprünglichen Hashwert wieder zurückerhalten. Ist dies der Fall, dann kann der Anwender sicher sein, dass der Unterzeichner (mit seiner Unterschrift) für das Applet bürgt und dass das Applet nach der Unterzeichnung nicht mehr verändert worden ist.
Die bei der Signatur verwendeten Algorithmen sind in der Klasse „Java.security.Signatur“ implementiert und werden von einem SecurityProvider zur Verfügung gestellt.

Obwohl digitale Signaturen das Problem der Herkunftsbestimmung technisch zu lösen scheinen, muss diese Technik mit Bedacht eingesetzt werden. Denn es stellt sich zum einen die Frage „Wer garantiert mir denn, dass ein öffentlicher Schlüssel wirklich echt ist“. Es könnte ja schließlich auch Hacker geben, die falsche öffentliche Schlüssel weitergeben und somit korrekte Signaturen für gefälschte Nachrichten berechnen! Zum anderen geht der Trend in der Softwareentwicklung klar dazu über, Programme modular aus fertigen Teilen zusammenzubauen. Eine Signatur kann also lediglich darüber Auskunft geben wer ein Programm signiert, nicht aber wer ein Programm entwickelt hat. Der Benutzer muss sich also bewusst sein, dass er dem Unterzeichner vertrauen muss all diese Teile entsprechend geprüft zu haben.

Neben der technischen Realisierung müssen deshalb beim Einsatz von signierten Programmen noch ein paar wichtige Aspekte geklärt werden:

- Es müssen entsprechende Schlüsselzertifikate bereitgestellt werden, damit Sichergestellt werden kann, dass eine Signatur auch zu dem Signierer passt.
- Digitale Signaturen sind nicht widerrufbar, d.h. ist ein Applet einmal signiert bleibt diese Signatur erhalten, selbst wenn es neuere Versionen eines Programms gibt. Angenommen ein vertrauenswürdiges Unternehmen signiert ein Applet und später stellt sich heraus, dass dieses Applet eine Sicherheitslücke beinhaltet. Selbst wenn diese Sicherheitslücke von dem Unternehmen behoben wird und die fehlerhafte Version ersetzt, kann ein Angreifer die alte Version auch weiterhin einsetzen, und es ist auch weiterhin „vertrauenswürdig“, da es von einer vertrauenswürdigen Quelle signiert ist.
- Es muss festgelegt sein, wer signieren darf. Denkbar wäre zum Beispiel, dass im Bereich mit spezieller Sicherheitsanforderung Applets von unabhängigen Instanzen (z.B. dem BSI) geprüft und signiert werden.
- Es muss außerdem geklärt werden, welche Rechte einem als „trusted“ erklärten Programm eingeräumt werden. Das Ziel sollte sein, dass einem Programm nur die Rechte gewährt werden, die es zur Ausführung seiner Aufgabe benötigt („need-to-know“-Policy).

Gültige Zertifikate erhält man z.B. bei VeriSign (http://verisign.com) oder bei Thawte Certification (http://www.thawte.com).

4.2 Die Sicherheitsarchitektur von Java 2

Package **java.security**:

- **ProtectionDomain**
  implementiert Protection Domains in der JVM (s.u.)
- **CodeSource**
  kapselt die URL, die von einer Klasse empfangen wurde, mit allen an der Klasse angehängten Signaturen
- **Permission**
  abstrakte Klasse für die Zugriffsrechte auf eine Ressource
- **AccessController**
  überprüft Zugriffsrechte von Subjekten auf Objekte
- **Policy**
  entspricht einer Laufzeit-Darstellung der Sicherheitsregeln
- **SecureClassLoader**
  implementiert den abstrakten **java.lang.ClassLoader**

4.2.1 Die Rechteverwaltung

Grundgedanke der Sicherheitsarchitektur in Java 2 war, Klassen in einzelne Schutzbereiche (protection domains) einzuteilen. Innerhalb einer domain besitzen alle Klassen die gleichen Rechte. Zur Implementation dieser Bereiche steht die java.architecture.ProtectionDomain zur Verfügung. Die Kontrolle, ob ein Benutzer auf eine Klasse zugreifen darf oder nicht, wird durch Zugriffsrechte (permissions) durchgeführt. Jedes Zugriffsrecht auf eine Ressource wird durch ein Objekt vom Typ Permission repräsentiert. Durch Unterklassen von Permission können dann die verschiedenen Arten von Ressourcen dargestellt werden (File, Socket, Net,...). Bei einigen kann zusätzlich auch noch eine Aktion angegeben werden. Dies ist z.B. bei Files sehr sinnvoll, um die Attribute *lesen*, *schreiben*, *ausführen* und *löschen* umzusetzen. Aus Sicherheitsgründen ist ein Permission-Objekt normalerweise nicht mehr zu ändern.

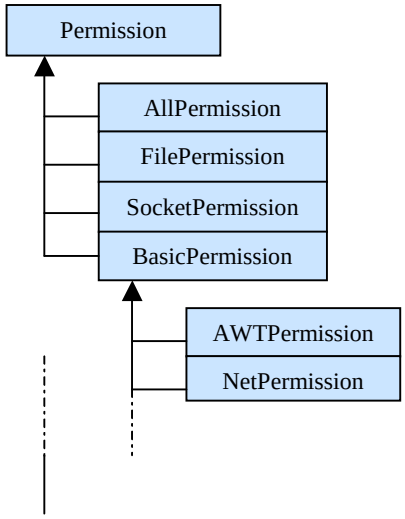

Abb. 3 Teilhierarchie der java.security.Permission

Beispiel:

```
import java.security.*;
// hier wird ein Objekt angelegt, dass Lese- und Schreibrechte
// auf alle Dateien im Verzeichnis temp hat.
Permision p = new FilePermission(”/temp/*”,”read write”);
```

Mit PermissionCollection können Permissions gruppiert werden. Mit add(Permission p) können, wie bei einer Liste, Permissions hinzugefügt werden. Jede Instanz dieser Klasse sollte nur Instanzen von Permission Objekten des gleichen Typs enthalten.

4.2.2 Die Zuordnung der Rechte

Als nächstes stellt sich die Frage, woher die JVM weiß, welche Schutzbereiche zu erzeugen sind, welche Klassen welchen Schutzbereichen zugeordnet werden sollen und welche Zugriffsberechtigungen zu den einzelnen Schutzbereichen gehören. Dies geschieht mit einer oder mehreren Dateien, die ein spezielles ASCII-Format haben und die entsprechenden Sicherheitsregeln beinhalten. Die Überprüfung auf ausreichende Rechte wird von der Klasse AccessController bewerkstelligt. Die JVM liest diese Sicherheitsregeldateien, oder auch policy files genannt, beim Start ein und speichert die Sicherheitsregeln in einem Objekt der Klasse java.security.Policy ab. Der Anwender braucht deshalb die Sicherheitsstrategien nicht mehr selbst zu programmieren (etwa durch Implementieren eines eigenen Sicherheitsmanagers), sondern kann die Sicherheitsregeln relativ bequem in den entsprechenden Dateien festlegen. Im Gegensatz zum SecurityManager kann die Policy auch ausgetauscht werden. In der JVM wird immer höchstens eine Policy-Instanz installiert.
Dies geschieht mit einem Aufruf von Policy.setPolicy(Policy p). Diese Instanz wird dann für alle nötigen Zuordnungen verwendet. Die Policy-Datei ist aus Blöcken aufgebaut, die jeweils eine CodeSource umschreiben. In den Blöcken sind dann die Permission aufgeführt, die zu dieser CodeSource gehören.

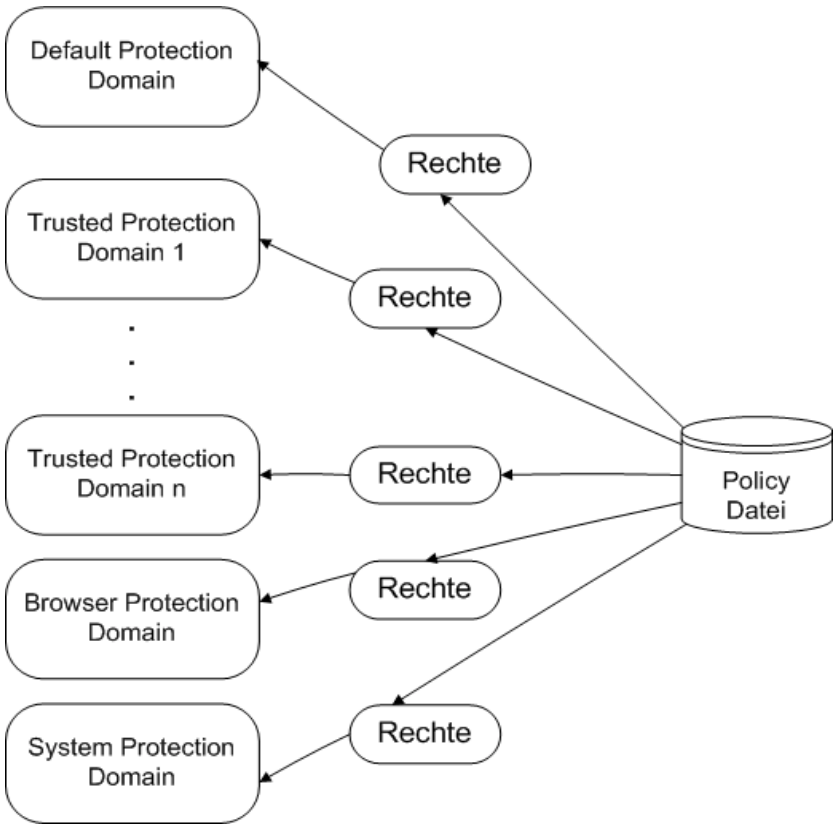

Abbildung 4

Einträge könnten folgendermaßen aussehen:

```
grant SignedBy „Marco Krause und Michael Adams“
CodeBase http://www.inf.fh-brs.de/          {
        permission java.io.FilePermission „/home/public/file.dat“, „read“;
        permission java.io.FilePermission “/VuPS_Lösungen/”, “read write”;
}
```

Dies bedeutet, dass alle Klassen, die von *„Marco Krause und Michael Adams“* (Alias für den öffentlichen Schlüssel) unterzeichnet und von der URL *http://www.inf.fh-brs.de* geladen worden sind, ein Leserecht auf die Datei *file.dat* aus dem Verzeichnis */home/public* und ein Lese- Schreibrecht auf das */VuPS_Lösungen* -Verzeichnis besitzen.

```
grant SignedBy „admin“   {
        permission java.security.AllPermission;
}
```

Alle von "admin" signierten Klassen bekommen AllPermission zugewiesen und dürfen damit auf alle Ressourcen zugreifen.

```
grant{
    permission java.awt.AWTPermission "showWindowWithoutWarningBanner";
}
```

Alle Klassen dürfen Windows ohne einen Warnhinweis ("Java Applet Window") erzeugen.

4.2.3 Die Kontrolle der Rechte

Um die Rechtmäßigkeiten von kritischen Operationen vor der Ausführung zu prüfen, ist eine weitere Klasse notwendig.

4.2.3.1 Der Access Controller

Seit Java 2 ist der Access Controller eine eigenständige Klasse außerhalb des Security Managers. Der Access Controller sammelt alle Rechte, die dem Byte Code zugewiesen wurden und entscheidet anschließend darüber, ob der Zugriff gewährt werden soll oder nicht. Dabei werden nicht die Rechte des gerade unmittelbar aufgerufenen Byte Codes zu Gunde gelegt, sondern es werden erst alle Rechte, aller von ihm aufgerufenen Funktionen, überprüft und anschließend richtet sich das Privileg des Byte Codes nach der Funktion mit dem geringsten Recht („least-privileg“-Prinzip). Damit wird sichergestellt, dass kein Programm, das vom Byte Code mit umfassenden Zugriffsrechten aufgerufen wird, unzulässiger Weise auf Systemressourcen zugreifen kann. Das explizite Untersagen von Aktionen für Applets bestimmter Herkunft oder bestimmter Identitäten ist dagegen nicht möglich.

Um Abwärtskompatibilität zu gewährleisten, delegiert der Security Manager die Kontrolle an den Access Controller. Im Gegensatz zum Security Manager ist der Access Controller robuster („classLoaderDepth“-Problem) und auch durch das leichte Einbinden eigener Rechte wesentlich flexibler.

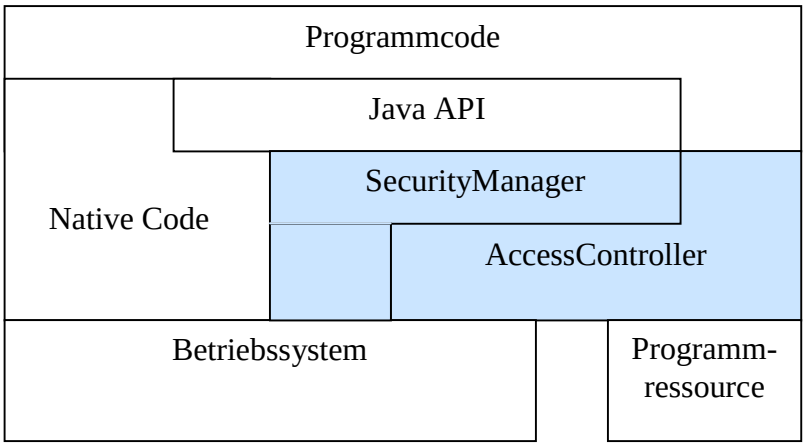

Abb 5: Aufbau der Java Virtual Machine

Der Access Controller hat sämtliche Methoden als static und die Klasse als final deklariert, um weitere Subklassen zu verhindern. In Java muss in erster Linie die Klasse selbst prüfen, ob der Zugriff auf die Ressource berechtigt ist. Beim Dateizugriff ist dies z.B. die Klasse java.io.File. Diese bedient sich allerdings des Access Controllers mit der Methode checkPermission(Permission p).

Ablauf mit dem Access Controller:

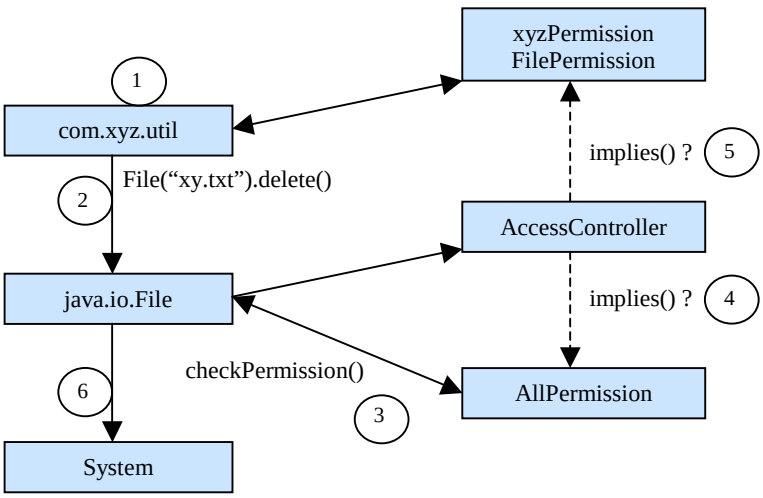

Abb. 6 Beispiel einer Überprüfung (vereinfachte Darstellung)

In der Klasse com.xyz.util wird eine Methode deleteFile(„xy“) aufgerufen (1)
In dieser Methode könnte dann beispielsweise versucht werden, diese Datei zu löschen:

```
public void deleteFile(String filename) {
  …
  try {
        (new File(filename)).delete(); \\ (2)
  }
  catch (Exception e) {
  …
  }
}
```

In der Systemklasse java.io.File erfolgt nun die Überprüfung der Rechte mittels des Access Controllers:

```
Perm = new FilePermission(„xy“,delete);
AccesssController.checkPermission(perm); \\ (3)
```

Der Access Controller prüft nun alle Ebenen des CallStacks, ob alle Klassen die geforderten Rechte aufweisen können. Dies geschieht mit Hilfe des Class Loader der dem Access Controller die zugehörigen Klassen der Protection Domain übermittelt. Hier wird die Klasse java.io.File zuerst mit der Methode implies(perm) auf Zugriffsrecht geprüft (4), aber da File eine Systemklasse ist, ist der Zugriff erlaubt. Nun wird die Methode auf die nächsttiefere Ebene com.xyz.util angewendet (5).

Wenn alle Klassen die benötigten Rechte aufweisen können, wird der Zugriff auf die Systemressource gewährt und das Programm kann die Datei löschen (6). Ist dies nicht der Fall, wirft der Access Controller eine Security Exception und die Operation kann nicht durchgeführt werden.

5. Zusammenfassung und Ausblick

Das Hauptinteresse gilt mittlerweile nicht mehr der Programmierung kleiner Applets, sondern man möchte grosse, kommerzielle Applikationen über das Netz versenden. Für solche Applikationen reicht aber das beschriebene Sicherheitskonzept nicht aus. Ohne Verfahren zur Autorisierung/Identifikation und zur Verschlüsselung von sensitiven Daten während der Übertragung sind solche Applikationen nicht realisierbar. Ein Textverarbeitungs-Applet muss z.B. Texte auf der lokalen Festplatte speichern können. Heute muss es das auf dem Server tun, von dem es geladen wurde. Dies ist die einzige Maschine mit der das Applet kommunizieren kann, während der Zugriff auf die lokale Festplatte verboten ist.
Das Sicherheitskonzept von Java wird in absehbarer Zeit erweitert. Bereits in der Version 1.1 ist es möglich mehrere Class-Files zusammenzufassen und mit einer digitalen Signatur zu versehen. Damit kann der Empfänger den Absender identifizieren und gleichzeitig prüfen, ob die Class-Files unterwegs verändert

wurden. Falls die Klassen von einem Ort stammen, den der Empfänger als vertrauenswürdig einschätzt, so kann der Sicherheits-Manager diesem Applet gegenüber weniger restriktiv sein. Der Sicherheits-Manager könnte also z.B. dem Textverarbeitungs-Applet erlauben, innerhalb eines beschränkten Bereiches auf die Festplatte zu schreiben. Digitale Signaturen erweitern das Sicherheitskonzept von Java um vertrauenswürdigen Code, der über das Netz geladen wird.

Java scheint aus verschiedenen Gründen eine recht sichere Sprache zu sein: Sicherheitsaspekte wurden bereits beim Design berücksichtigt und nicht erst später dazugefügt. Ein klares Sicherheitskonzept ist erkennbar, welches vier sorgfältig aufeinander abgestimmte Schutzschichten auf unterschiedlichen Ebenen vorsieht. Dem Thema Sicherheit wird im Zusammenhang mit Java grosse Aufmerksamkeit entgegengebracht. Die Frage, ob die so erreichte Sicherheit genügend gross ist, um Java im besonders risikoreichen Umfeld von mobilem Code und offenen Netzen problemslos einzusetzen, kann nicht generell beantwortet werden. Diese Frage ist für jede Applikation einzeln und im Rahmen einer umfassenden Sicherheitsstrategie zu beurteilen.
Breite Erfahrung wurde bisher nur mit kleinen Applets in Web-Seiten gesammelt. Die Erfahrungen sind insofern positive, als dass keine grossen Probleme bekannt geworden sind. Daraus kann aber nur bedingt auf die Sicherheit von Java geschlossen werden. Noch fehlen Erfahrungen mit vertrauenswürdigen Applets und mit grossen, kommerziellen Applikationen. Auch ist die Entwicklung von Java im Bereichder Sicherheit noch nicht abgeschlossen. Sicher ist deshalb wohl nur, dass für jeden, der Java verwendet – sei es als Anwender oder Entwickler – die Sicherheit von Java ein Thema bleiben wird.

6. Literaturverzeichnis

- www.informatik.uni-bonn.de/III/lehre/seminare/Softwaretechnologie/WS98/vortraege/sicherheit/JVM.html
- www.joller-voss.ch/ndkjava/notes/security/JavaSecurity.pdf
- www.ifi.unizh.ch/richter/people/pilz/pubb/Pilz97a.pdf
- http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-specTOC.fm.html
- http://wwwswt.fzi.de/~schanne/jasese/
- http://www.secorvo.de/publikat/javaactx.pdf
- http://java.sun.com/products/jdk/1.2/docs/guide/security/PolicyFiles.html
- http://java.sun.com/products/jdk/1.2/docs/guide/security/permissions.html
- http://wwwswt.fzi.de/~schanne/jasese/4_zertifikate_signatur_security-provider.pdf
- http://www.nue.et-inf.uni-siegen.de/themen_und_quellen_zum_proseminar.html
- http://www.secorvo.de/publikat/javaactx.pdf
- http://enterprisesecurity.symantec.de/article.cfm?articleid=1075&EID=0
- http://www.fh-wedel.de/~si/seminare/ws98/Ausarbeitung/3.Flaegel/security3.html#Provider
- Horster, P.: Datenschutz und Datensicherheit – DuD, Verlag Vieweg, Braunschweig 1999
- Goll,J, Weiß,C., Müller,F.; Java als erste Programmiersprache, Teubner-Verlag, Stuttgart/Leipzig/Wiebaden 2001
- Esser, F.; Java 2 – Designmuster und Zertifizierungswissen, Verlag Galileo Computing, Bonn 2001