Abkürzungsverzeichnis

JDK	Java Development Kit
KDE	K Desktop Environment

Abbildungsverzeichnis

Duke von java.sun.com © Sun Microsystems

1. Einleitung

Wie viele Softwareprodukte unterliegt auch Java einem Entwicklungsprozeß. Schon zur Zeit der Einführung von Java bestand großes Interesse daran die Plattform möglichst sicher zu gestalten. Mittlerweile hat sich aber das Sicherheitsmodell der Java Plattform von einem striktem zu einem sehr fein konfigurierbaren Modell entwickelt. Was genau dahinter steckt, soll im folgenden dargestellt werden.
Es steht natürlich außer Frage, daß im Rahmen dieser Arbeit nicht alle Sicherheitsmechanismen behandelt werden können. Vielmehr werde ich nur einen Punkt herausgreifen und auf ihn näher eingehen: Das Sicherheitsmodell der Java 2 Plattform.
Ich möchte im folgenden darlegen, wie man Sicherheitsrichtlinien einrichten und verwenden kann und welche Mechanismen dahinter stehen.
Das Beispiel im Anhang soll die Anwendung der im folgenden näher beschriebenen Sicherheitsmechanismen und -einstellungen verdeutlichen.

2. Sicherheitsmechanismen in den verschiedenen JDK Versionen im Überblick

In allen Java Development Kit (JDK) Versionen steht ein sogenannter Sandkasten („Sandbox“) zur Verfügung, in dem man Anwendungen ausführen („spielen lassen“) kann. Der Sandkasten dient dem Schutz verschiedener Systemressourcen und stellt die Grenze einer darin laufender Anwendung dar. Bildlich gesprochen heißt das: eine Anwendung kann nur mit dem Spielzeug spielen, das der Sandkasten zur Verfügung stellt (siehe auch 2.4 Sandkastenmodell).

Im Laufe der Zeit hat sich das Sicherheitsmodell von Java von einem sehr striktem zu einem sehr fein konfigurierbaren Modell entwickelt. Im folgenden soll ein Überblick über die Sicherheitsarchitektur in den verschiedenen JDK Versionen gegeben werden.

2.1. Sicherheitsmodell im JDK 1.0

Das Sicherheitsmodell im JDK 1.0 wurde eingeführt um Code[1], dessen Autor nicht vertraut wird, und über das Netzwerk geladen wird, eine sehr eingeschränkte Umgebung zu bieten. Lokalen Programmen wird im Gegensatz dazu vertraut und sie sind in der Lage, über die kompletten Systemressourcen wie z.B. das Dateisystem zu verfügen.
Wie in Abbildung 2-1 gut zu sehen, gibt es nur zwei Möglichkeiten: Alles oder „Nichts“! Anwendungen, die aus dem Netz kommen, dürfen sich nur im Sandkasten bewegen, lokalen Anwendungen hingegen stehen alle Ressourcen zur Verfügung.

Dieses Sicherheitsmodell hat natürlich den Vorteil, daß kein fremder Code das eigene System gefährden kann, andererseits besteht aber auch nicht die Möglichkeit, fremden Code in Ausnahmefällen mehr Rechte bzw. Ressourcen zur Verfügung zu stellen.

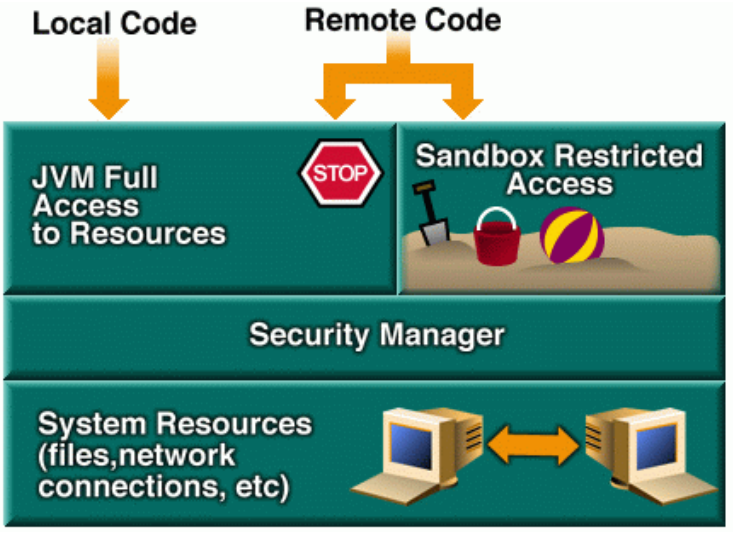

Abbildung 2-1 Sicherheitsmodell im JDK 1.0[2]

2.2. Sicherheitsmodell im JDK 1.1

Das Sicherheitsmodell im JDK 1.1 hat sich in einem wichtigen Punkt gegenüber dem Modell aus Version 1.0 verändert: Aus dem Netz geladene Anwendungen (Java-Applets, Java-Anwendungen) können aus dem Sandkasten „ausbrechen“ und so auf die kompletten Systemressourcen zugreifen. So kann z.B. ein Applet, das man aus dem

[1] Steht im folgenden für Programme
[2] Quelle: http://java.sun.com/docs/books/tutorial/figures/security1.2/scrtymdl.gif

Internet heruntergeladen hat, Dokumente lokal ausdrucken. Ermöglicht wird das durch das sogenannte Signieren des Codes.[3]

Der Java-Code muß nachweislich von einer bestimmten Person stammen, und der Anwender muß sich entscheiden, ob er dem Autor vertraut und somit der Anwendung die gesamten Systemressourcen zugänglich macht.
Auch in diesem Modell gibt es wieder nur Schwarz oder Weiß. Entweder man traut dem Autor des Codes, oder man führt die Anwendung wieder in einem restriktivem Sandkasten aus.

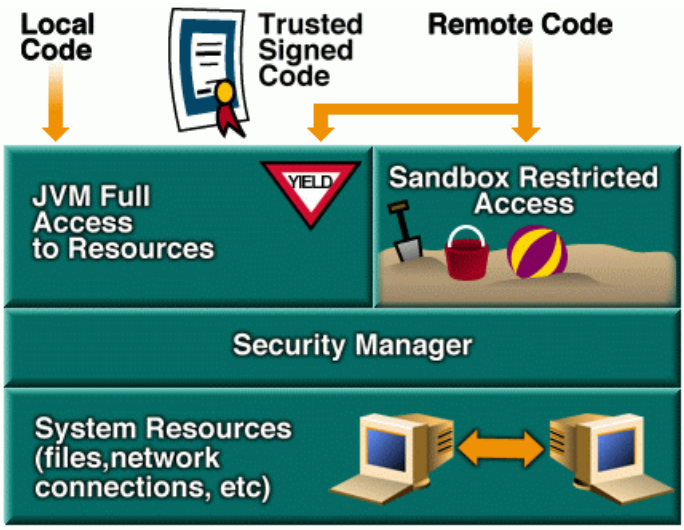

Abbildung 2-2 Sicherheitsmodell im JDK 1.1[4]

2.3. Sicherheitsmodell im JDK 1.2 / Java 2

Die Version 1.2 des Java Development Kits, auch Java 2 genannt, hat einige Verbesserungen zu seinen Vorgängern gebracht.

Mit dieser Version ist das „Alles oder Nichts Prinzip“ der Vorgängerversionen durchbrochen worden. Es besteht jetzt die Möglichkeit Code, egal ob er über das Netz oder lokal geladen wurde, bestimmten Sicherheitsrichtlinien zu unterwerfen. Diese Sicherheitsrichtlinien bestehen aus einer Zusammenstellung verschiedener Rechte, die verschiedenen Autoren und/oder verschiedenen Netzadressen (locations) zugeteilt werden können. So kann z.B. einem bestimmten Applet, das von einer bestimmten Person signiert wurde, die Möglichkeit gegeben werden, sich mit einem anderen bestimmten Rechner über einen bestimmten Port zu verbinden, was im „orginal“ Sandkasten nicht möglich gewesen wäre.

Aber auch in der Version 1.2 ist es möglich einen „orginalen“ Sandkasten zur Verfügung zu stellen.

[3] iX November 1999 “Offene Gesellschaft” von Jörn Heid S. 202ff
[4] Quelle: http://java.sun.com/docs/books/tutorial/figures/security1.2/scrtmdl2.gif

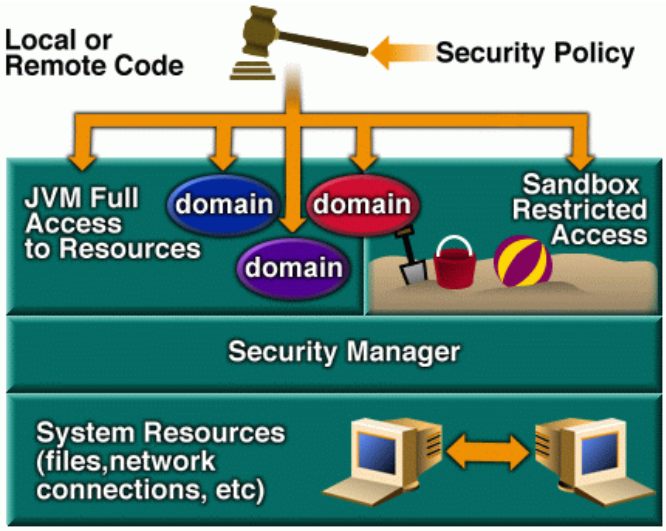

Abbildung 2-3 Sicherheitsmodell in Java 2[5]

Im folgenden Kapitel soll nun auf das Sicherheitsmodell der Java 2 Plattform näher eingegangen werden.

2.4. Sandkastenmodell

In den vorangegangenen Absätzen war immer die Rede vom "Sandkasten". Was genau hinter diesem Sandkastenmodell steht soll an dieser Stelle kurz erläutert werden.
Der "original" Sandkasten stellt, wie schon oft erwähnt, eine eingeschränkte Umgebung für nichtvertrauten Code dar. Welche Operationen Code, der im original Sandkasten ausgeführt wird, **nicht** ausführen darf, zeigt folgende Aufzählung:

- Dateien lesen, schreiben, umbenennen oder löschen
- Den Inhalt eines Verzeichnisses lesen oder erzeugen
- Informationen über eine Datei sammeln (z.B. Größe, Typ, ..)
- Informationen über den Benutzer in Erfahrung bringen (home directory, user name)
- Eine Netzwerkverbindung zu einem anderen Rechner wie dem Ursprungsrechner aufbauen
- Auf Netzwerkverbindungen warten und diese annehmen
- Netzwerkkontrollfunktionen angeben
- System Eigenschaften angeben/verändern (z.B. `os.name` auf einen anderen Wert setzen)
- Ein Programm ausführen, welches die Runtime.exec() Methode oder dynamische Bibliotheken benutzt
- class loaders oder security managers erzeugen (instanziieren)
- threads, die außerhalb der Applet threadgroup liegen, erzeugen bzw. verändern (z.B. einen thread in der Priorität verändern)
- Klassen, die lokal in Paketen auf dem Client liegen, definieren.

[5] Quelle: http://java.sun.com/docs/books/tutorial/figures/security1.2/scrtmdl3.gif

Trotz diesen Einschränkungen bietet der Sandkasten keine 100% Sicherheit gegenüber fremden Programmen. Sie haben noch die Möglichkeit das System auf andere Weise "in die Knie" zu bringen bzw. den Benutzer zu täuschen. So können fremde Programme:

- Viel Rechenleistung "verschlucken". (CPU Zeit verschwenden)
- Eingabemasken fälschen, so daß der Benutzer dazu gebracht wird, vertrauliche Angaben anzugeben (siehe Abbildung 4Abbildung 5)
- Sogenannte "denial of service" Angriffe durchzuführen (z.B. zu viele Fenster zu öffnen)

Abbildung 4 Beispiel einer gefälschten Eingabemaske[6]

Abbildung 5 Original Eingabemaske[7]

[6] http://developer.java.sun.com/developer/onlineTraining/Security/Fundamentals/images/invalidPrompt.gif
[7] http://developer.java.sun.com/developer/onlineTraining/Security/Fundamentals/images/validPrompt.gif

3. Das Sicherheitsmodell in Java 2

Wie schon erwähnt hat die Einführung der Java 2 Plattform im Sicherheitsbereich einige Erweiterungen gebracht. Das Sicherheitsmodell in Java 2 erleichtert Systemadministratoren, Webmastern aber auch allen anderen den Umgang mit Sicherheitseinstellungen. Es spielt hierbei keine Rolle, ob man sich im firmeneigenen Intranet oder im weltumspannenden Internet bewegt.
Sun Microsystems nennt 4 Vorteile, die die Java 2 Plattform in Puncto Sicherheit bietet[8]:

- Eine sehr fein konfigurierbare Zugriffskontrolle
  Es war auch schon in den älteren Versionen des Java Development Kits möglich bestimmte Objekte mit Zugriffskontrollen/ Berechtigungsstrukturen auszustatten. Dies war allerdings mit einem beträchtlichem sicherheitskritischen Programmieraufwand verbunden und es wurden meist nur recht grobe Strukturen erreicht.
- Leicht konfigurierbare Sicherheitsrichtlinien
  Wie schon erwähnt, war es auch schon in früheren Versionen möglich, bestimmten Anwendungen bestimmte Rechte einzuräumen. Dies war allerdings mit relativ viel Aufwand verbunden.
  Dem Anwender und Entwickler soll nun die Möglichkeit gegeben werden, ohne Programmieraufwand Sicherheitsrichtlinien aufzustellen. Hierzu dienen die Policy Dateien, die weiter unten noch besprochen werden.
- Leicht zu erweiternde Zugriffskontrollstrukturen
  Bis zu der Version 1.1 mußte man, wenn man neue Berechtigungen in einem Programm aufnehmen wollte eine neue `check`-Methode in den `security manager` implementieren, dieser Programmieraufwand ist in der neuen Architektur nicht mehr nötig. Auf das Vorgehen bei der Version 1.2 wird weiter unten näher eingegangen.
- Erweiterung von Sicherheitskontrollen auf alle Java Programme
  In der Version 1.2 wird nicht mehr zwischen lokalem Code, dem vertraut wird, und „externen“ Code unterschieden, vielmehr besteht jetzt die Möglichkeit, auch lokalen Code bestimmten Sicherheitsrichtlinien zu unterwerfen. Desweiteren ist es völlig egal, ob es sich um eine Anwendung oder ein Applet handelt. Die Sicherheitsrichtlinien lassen sich individuell anpassen.

3.1. Protection Domains

Grundlegend für die gesamte Sicherheitskonzeption der Java 2 Plattform sind die sogenannten Protection Domains. Wird Code geladen, wird er einer Protection Domain zugewiesen, und er darf nur Aktionen ausführen, die ihm durch die Domain zugestanden werden.

3.1.1. Was ist eine Domain?

Es stellt sich natürlich jetzt die Frage: Was ist eine Domain?
Eine Domain besteht aus einer Zusammenstellung von Objekten, denen bestimmte Berechtigungen zugeteilt werden. Diese Berechtigungen werden im Standardfall über sogenannte policy files (siehe 3.3

[8] http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc1.html

Berechtigungen und Sicherheitsvorgaben (Security Policies)) bestimmt. Eine Domain dient also zur Zusammenstellung und Isolation bestimmter „Berechtigungeseinheiten“.

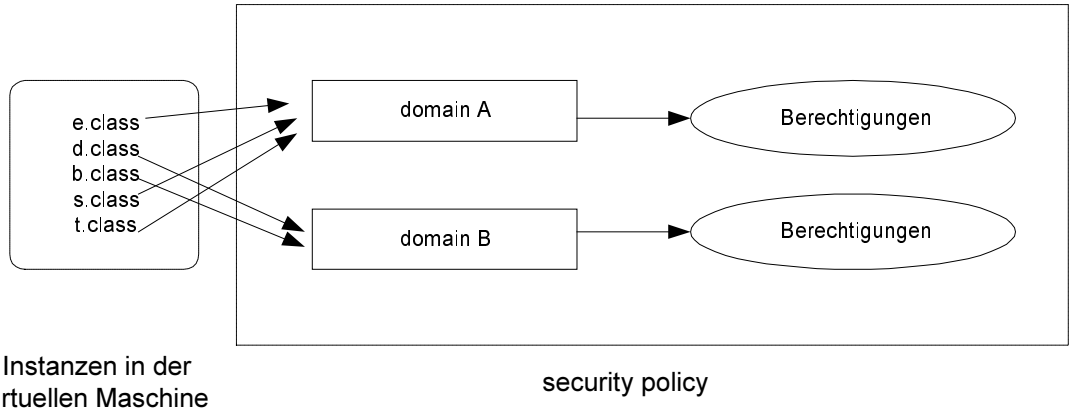

Abbildung 3-1 Protection Domain

In Abbildung 3-1 ist dargestellt, daß die Objekte[9] zur Laufzeit verschiedenen Domains zugeteilt werden, die wiederum verschiedene Berechtigungen besitzen. Berechtigungen einer Domain werden durch zwei Attribute bestimmt:

- Unterzeichner und
- Herkunft des Codes.

Wobei der Unterzeichner die Person[10] ist, die den Code digital unterzeichnet hat. Die Herkunft stellt die URL dar, von welcher der Code geladen wurde.

Außerdem ist möglich, auch wenn das noch nicht durch eine standardmäßig eingebaute Funktion unterstützt wird, Protection Domains von einer Kommunikation untereinander abzuhalten, so daß jede Interaktion entweder über vertrauten Systemcode oder über eine explizite Erlaubnis beider Domains stattfinden muß.

3.1.2. Beispiel für Domains

Um zu zeigen, wie eine solche Domain aussehen kann, sei hier ein Beispiel gegeben:

Es wurde eine security policy so eingerichtet, daß alle Klassen, die von den Rechnern des Rechenzentrum Franken Unterer Neckar heruntergeladen und vom Rechenzentrum unterzeichnet wurden, in die Domain `rzfun` gelangen. Außerdem könnten in der gleichen security policy die Klassen, die von den Webservern von Sun Microsystems heruntergeladen wurden, einer Domain `sun` zugeteilt werden.

Beiden Domains kann man dann dementsprechende Berechtigungen einräumen.
Wenn man dieses Beispiel auf Abbildung 3-1 übertragen möchte, so würde z.B. die domain A aus dem Beispiel die Domain `rzfun` darstellen und die domain B die Domain `sun`. Die Klassen `e`, `s` und `t` würden somit über die Berechtigungen der Domain `rzfun`, und die Klassen `d` und `b` über die Berechtigungen der `sun` Domain verfügen.

[9] Instanzen von Klassen in der virtuellen Maschine
[10] Es kann sich um eine juristische oder eine natürliche Person handeln (Firma, Organisation oder “leibliche” Person)

3.1.3. Kategorien von Domains

Grundsätzlich wird in zwei Kategorien von Domains unterteilt: Die System und die Anwendungs domains.

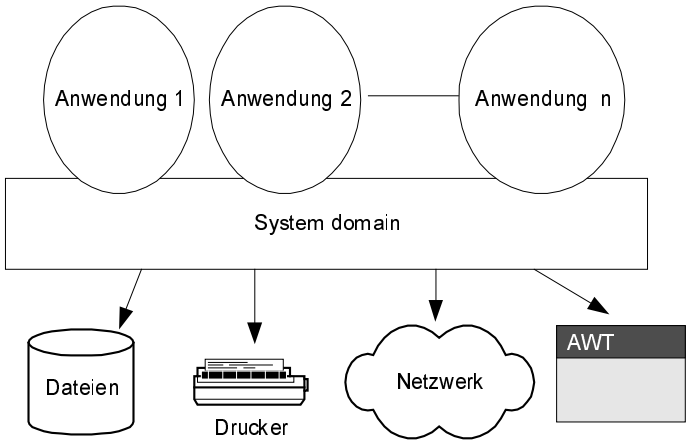

Abbildung 3-2 System und Anwendungs Domains

Zu den System domains zählen alle Klassen, die uneingeschränkt auf die Ressourcen des Rechners zugreifen können. Wie aus Abbildung 3-2 zu ersehen, zählen z.B. die Klassen java.io.File, java.awt und java.net dazu. Aber auch die Klassen die über den CLASSPATH erreicht werden können werden als Systemcode behandelt und zählen somit auch zur Systemdomain. Alle anderen Domains werden als Anwendungsdomain bezeichnet.

3.1.4. Domainübergreifende Methodenaufrufe

Wie weiter oben beschrieben, gehört jeder Code, und somit jedes Objekt, immer zu einer Protection Domain. Allerdings kann es auch vorkommen, daß eine Anweisungseinheit[11] nicht komplett innerhalb einer Protection Domain abläuft, sondern sich über mehrere Domains erstreckt. Dies ist z.B. der Fall, wenn eine Anwendung eine Meldung ausgibt. Hier muß die Anwendungsdomain mit der Systemdomain interagieren, da die Systemdomain der einzige Punkt ist, über den mit dem System kommuniziert werden kann. In diesem Fall ist es enorm wichtig, daß die Anwendungsdomain durch den Aufruf der Systemdomain keine zusätzlichen Rechte erhält. Im umgekehrten Fall darf es natürlich auch nicht möglich sein, daß eine Anwendungsdomain durch einen Aufruf der Systemdomain mehr Rechte bekommt, wie ihr durch die policies zugestanden wurden. Diese Situation (Systemdomain ruft Methode aus der Anwendungsdomain auf) kann z.B. dann auftreten, wenn das System die `paint()` Methode eines Applets aufruft.

Kurz gesagt kann sich keine Domain, die mit weniger Rechten wie eine andere ausgestattet ist, durch Aufruf von Methoden einer anderer Domain mit mehr Rechten ausstatten.

[11] es muß sich in diesem Fall nicht um einen Thread handeln

Eine einfache Regelung für domainübergreifende Methodenaufrufe ist[12]:

- Die Berechtigung für die ausführende Anweisung ist die kleinste gemeinsame Berechtigung, aller von der Anweisung berührter Protection Domains. („kleinster gemeinsamer Nenner“)
- Eine Berechtigung wird erteilt, wenn ein Codestück die Methode `beginPrivileged` (siehe weiter unten) aufruft, und
  - es durch die eigene Protection Domain und
  - alle direkte oder indirekt berührten nachfolgenden Domains erlaubt ist.

Wie schon oben kurz erwähnt, gibt es auch die Möglichkeit mit der `doPrivileged` Methode Codefragmente kurzzeitig mit mehr Rechten auszustatten, als ihrer Anwendung normalerweise zustehen würde. Diese Funktion wird in manchen Situationen nötig. Als Beispiel sei hier die Verwendung von Schriften genannt. Eine Anwendung (Applet) hat z.B. keine Berechtigung direkt von der Festplatte Schriftdateien zu lesen, sie muß allerdings diese Schriftendateien zur Verfügung haben.

Ein Aufruf der `doPrivileged` Methode ermöglicht allerdings keiner Klasse, sich mehr Rechte anzueignen, vielmehr kann sie Berechtigungen, die sie besitzt, anderen Klassen zur Verfügung stellen, d.h. die `doPrivileged` Methode ist nicht in der aufrufenden, sondern in der aufgerufenen Methode implementiert. In unserem Beispiel mit der Schriftdatei, ist die `doPriviliged` Methode nicht im Applet implementiert, sondern in der entsprechenden Klasse der Systemdomain.

Greift ein Objekt einer Protection Domain auf eine Methode zu, die sie nicht ausführen darf, wird an der ersten Stelle, an der es zu einer Verletzung von Zugriffsrechten kommt, eine Exception erzeugt. Wird keine Exception erzeugt, so wurde der Aufruf der Methode erlaubt.

3.2. Warum Protection Domains?

Nach den Ausführungen was eine Domain ist, sollte eine Frage leicht zu beantworten sein: Warum Protection Domains?

Bis zum Domainmodell gab es nur zwei Möglichkeiten: dem Code wurde entweder vertraut und alle Rechte eingeräumt, oder ihm wurde nicht vertraut und somit auch keine Rechte zugestanden. Solche recht einfachen Entscheidungen mag es im Computer geben, aber in der wirklichen Welt wird es schwer sein alles nach diesem Schwarz/Weiß-Prinzip abzuhandeln. Mit den Protection Domains erhält man die Möglichkeit bestimmten Anwendungen bestimmte, sehr fein konfigurierbare, Rechte einzuräumen.
Netzwerkadministratoren sowie Webmaster und alle anderen Benutzer, die Sicherheitsrichtlinien einrichten müssen, können mit dem, in Java 2 neu eingeführtem Modell, Sicherheitseinstellungen vornehmen, ohne Änderungen im Quelltext der Programme vornehmen zu müssen. Es ist recht leicht sich eigene Berechtigungsstrukturen für verschiedene Programme und/oder verschiedene Autoren einzurichten. Man kann selbst entscheiden, ob man bestimmten Programmen Zugriffsberechtigungen erteilen möchte oder nicht. Dies war bis zu Java 2 nur mit relativ viel Programmieraufwand zu bewerkstelligen.

[12] sun educational services – Implementing Java Security 5-34

3.3. Berechtigungen und Sicherheitsvorgaben (Security Policies)

Wie unter 3.1 erwähnt, können Protection Domains bestimmte Rechte eingeräumt werden. Diese Zuweisung erfolgt über die Security Policies. Hier werden Autoren und Codeherkunft[13] mit Berechtigungen und Sicherheitsvorgaben versehen, also Domains gebildet.

Es handelt sich im Normalfall um ASCII Dateien, die an bestimmten Stellen im Verzeichnisbaum abgelegt sind. Man unterscheidet hier drei verschiedene Policy Files:

- System Policy File,
- User Policy File und das
- Anwendungs Policy File.

Wo sich die Dateien befinden wird in der `java.security` Datei, die sich im Unterverzeichnis `lib/security` des JRE Verzeichnisses befindet, definiert. Standardmäßig befindet sich die System Policy Datei auch im `lib/security` Verzeichnis des JREs, und heißt `java.policy`. Hier lassen sich auch noch weitere Policy Dateien angeben, die dann in der angegebenen Reihenfolge eingelesen werden. Berechtigungen werden dann immer „hinzugefügt“.

```
..
policy.url.1=file:${java.home}/lib/security/java.policy
..
```

Quelltext 1 Ausschnitt aus der `java.security` Datei

In der `java.security` Datei werden außerdem weitere sicherheitsrelevante Einstellungen vorgenommen, die aber an dieser Stelle nicht weiter erläutert werden sollen.

Wie in Quelltext 1 zu sehen handelt es sich bei der Definition der Policy Dateien um URLs. Es könnte sich also prinzipiell auch um eine Policy Datei handeln, die sich irgendwo in einem Intranet befindet und von einem Administrator zentral verwaltet wird. Somit ließe sich eine unternehmensweite Sicherheitsstruktur für Java Anwendungen implementieren.

3.3.1. System Policy File

Die System Security Datei definiert das Grundgerüst der Sicherheitseinstellungen der virtuellen Maschine. Eine System Policy Datei ist für jeden Benutzer, der mit der virtuellen Maschine arbeitet gleich. Jedem Programm liegen diese Sicherheitseinstellungen zu Grunde[14]. In Quelltext 2 wird die standardmäßige `java.policy` Datei gezeigt. Das Format der Datei wird weiter unten noch näher erläutert.

[13] locations ☺
[14] nur wenn auch ein security manager aktiviert ist!

```
// default permissions granted to all domains

grant {
       // allows anyone to listen on un-privileged ports
       permission java.net.SocketPermission "localhost:1024-", "listen";

       // "standard" properies that can be read by anyone

       permission java.util.PropertyPermission "java.version", "read";
       permission java.util.PropertyPermission "java.vendor", "read";
       permission java.util.PropertyPermission "java.vendor.url", "read";
       permission java.util.PropertyPermission "java.class.version", "read";
       permission java.util.PropertyPermission "os.name", "read";
       permission java.util.PropertyPermission "os.version", "read";
       permission java.util.PropertyPermission "os.arch", "read";
       permission java.util.PropertyPermission "file.separator", "read";
       permission java.util.PropertyPermission "path.separator", "read";
       permission java.util.PropertyPermission "line.separator", "read";
       permission java.util.PropertyPermission "java.specification.version", "read";
       permission java.util.PropertyPermission "java.specification.vendor", "read";
       permission java.util.PropertyPermission "java.specification.name", "read";
       permission java.util.PropertyPermission "java.vm.specification.version", "read";
       permission java.util.PropertyPermission "java.vm.specification.vendor", "read";
       permission java.util.PropertyPermission "java.vm.specification.name", "read";
       permission java.util.PropertyPermission "java.vm.version", "read";
       permission java.util.PropertyPermission "java.vm.vendor", "read";
       permission java.util.PropertyPermission "java.vm.name", "read";
};
```

Quelltext 2 Die Datei java.policy

3.3.2. Benutzer Policy File

Im Gegensatz zu der System Policy Datei gibt es auch noch Benutzer Policy Dateien, die für jeden Benutzer individuell gelten. Mit ihnen hat jeder Benutzer die Möglichkeit, sich seine eigenen Sicherheitsrichtlinien einzurichten. Wo sich die Benutzer Dateien befinden, wird auch in der `java.security` Datei angegeben. In den Standardeinstellungen wird die Datei `.java.policy` aus dem Homeverzeichnis des Benutzers geladen.

```
..
policy.url.2=file:${user.home}/.java.policy
..
```

Quelltext 3 Ausschnitt aus der java.security Datei

Die zusätzlichen Berechtigungen, die durch eine Benutzer Policy Datei eingeräumt werden, werden einfach zu den bisher bestehenden Berechtigungen „addiert“.

Fehlen beide Dateien, System und Benutzer Policy, gelten die „eingebauten“ Sicherheitsrichtlinien, die die Sandkasten Berechtigung als Richtlinie nehmen.

Standard ist **eine** System- sowie **eine** Benutzer-Richtlinie.

3.3.3. Anwendungs Policy File

Seit der Version 1.2 des JDKs kann auch eine Anwendung eine Policy Datei besitzen. Somit kann man auch einer bestimmten Anwendung bestimmte Rechte geben.

Die Namenskonvention sieht folgendermaßen aus:

```
policy.application-name=policy_name
```

Quelltext 4 Anwendungs Policy Datei

Ein Beispiel hierzu:

```
policy.navigator=navigator.policy
```

Quelltext 5 Beispiel für eine Anwendungs Policy Datei Angabe

3.3.4. Policy File Format

Das Format der Policy Datei ist für System und Benutzer Richtlinien gleich. Hier soll das Format kurz vorgestellt werden.

```
grant [signedBy "signer names"] [,codeBase "URL"] {
       permission permission class name ["target name"]
  [, "action"] [, signedBy "signer_names"];
       permission ...
};
```

Quelltext 6 Format der Policy Dateien

In Quelltext 6 sieht man den prinzipiellen Aufbau der Datei. Die Einträge in den eckigen Klammern sind optional und können somit weggelassen werden, wenn sie nicht benötigt werden. Ein Beispiel einer Datei sieht man in Quelltext 2.

Allgemein gilt für alle Einträge[15]:

- Ein Eintrag beginnt mit dem Schlüsselwort `grant`
- Jeder "Berechtigungsuntereintrag" beginnt mit dem Schlüsselwort `permission`
- Leerzeichen sind vor und nach Kommata erlaubt.
- Der Name der Permission Klasse muß qualifiziert angegeben werden. (z.B. `java.io.FilePermission`)
- Das `action` Feld ist optional, muß aber, wenn es angegeben ist, direkt nach dem `target_name` Feld angegeben werden.
- Das `codeBase` Feld ist optional. Falls es weggelassen wird, gilt die Richtlinie für alle CodeBases.
- Das erste `signedBy` Feld kann eine durch Kommata getrennte Liste von unterzeichnenden Autoren sein. z.B. "Carols,Eric,Nick“, was soviel heißt wie, daß der Code von Carlos **und** Eric **und** Nick unterzeichnet sein muß. Es reicht nicht aus, wenn der Code nur von einem der drei unterzeichnet wurde.
- Das zweite `signedBy` Feld (hinter Permission) bezieht sich auf die Berechtigung an sich. Hier muß die angegebene Person den Bytecode unterzeichnet haben, der die Berechtigung implementiert.

[15] http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc3.html

Außer den eigentlichen Berechtigungen, die angegeben werden, kann auch **ein** „keystore“ Eintrag angegeben werden.

In der jetzigen Version des JDKs kann nur **ein** keystore Eintrag vorkommen. Wenn mehrere Einträge angegeben sind, wird nur der erste Eintrag berücksichtigt, alle anderen werden ignoriert.

Ein „keystore“ ist eine Datenbank, die öffentliche Schlüssel und die dazugehörigen Zertifikate enthält. Wenn also in einem `grant` Eintrag ein Autor als Unterzeichner angegeben ist, wird die keystore Datenbank nach dem passenden öffentlichen Schlüssel durchsucht und anschließend die Datei „freigegeben“. Die Angabe zu einer keystore Datenbank sieht folgendermaßen aus:

```
keystore "some_keystore_url", "keystore_type";
```

Quelltext 7 Angabe der "keystore" Datenbank

Hier spezifiziert das Feld `"some_keystore_url"` eine URL, an der die Datenbank liegt. Auch hier ist es wieder möglich eine Datenbank unternehmensweit zur Verfügung zu stellen, damit nicht jeder Mitarbeiter seine Datenbank selbst pflegen muß. Die Angabe des Types der Datenbank ist optional. Wird kein Typ angegeben, so wird der Standardtyp aus der `java.security` Datei herangezogen. Die angegebene URL ist relativ zur Policy Datei zu betrachten. Bei folgender Angabe der Policy Datei in der `java.security` Datei:

```
policy.url.1=http://ich.weiss.net/sonstwas/irgendeine.policy
```

Quelltext 8 Beispiel zur Angabe einer Policy URL

und folgendem keystore Eintrag:

```
keystore ".keystore"
```

Quelltext 9 Beispiel zur Angabe eines keystores

wird unter folgender URL nach der Datenbank gesucht:

```
http://ich.weiss.net/sonstwas/.keystore
```

Quelltext 10 Beispiel zur keystore URL

Die angegebene URL kann natürlich auch absolut angegeben werden.

Als Typ der Datenbank wird das Format der Datenbank selbst und der Algorithmus, der zum Schutz[16] der öffentlichen Schlüssel verwendet wird, bezeichnet. Der Standardwert steht hier auf `"JKS"` einem properitären Typ von Sun Microsystems.

Bei vielen Berechtigungseinstellungen ist es sinnvoll, sogenannte "Wildcards" zu verwenden. Zum Beispiel möchte man ein ganzes Verzeichnis zum Schreiben freigeben und nicht nur eine einzelne Datei.

[16] auch Integritätsschutz

Für dieses Beispiel gilt:

- `Verzeichnis/*` steht für alle Dateien im angegebenen Verzeichnis.
- `Verzeichnis/-` steht für alle Dateien und Unterverzeichnisse ausgehend vom angegebenen Verzeichnis.
- `-` steht für das komplette Dateisystem.

"Wildcards" können auch bei der Berechtigung `java.net.SocketPermission` sinnvoll sein. Auch hier ein kleines Beispiel:

- `*.domain` steht für alle Rechner in einer Domäne[17]
- `*.subdomain.domain` steht für alle Rechner einer Unterdomäne
- `*` steht für alle Rechner.

Wie die einzelnen Berechtigungen mit den "Wildcards" umgehen ist weitgehend ihnen überlassen. Dies muß in der jeweiligen Permission Klasse implementiert werden.

3.3.4.1. "Variablen" in der Policy und der Security Properties Datei

Es besteht die Möglichkeit "Variablen" in der Policy und der Security Properties Datei zu verwenden. Diese verwendeten "Variablen" verhalten sich wie Variablen auf der Kommandozeilenebene. Hier stellt eine Variable, z.B. PATH, einen individuellen Wert für einen Benutzer dar. Gleiches gilt für die "Variable" in den Policy oder Security Properties Dateien. Ein Wert wie "${some.policy}", der in einer der Dateien auftaucht, wird durch den entsprechenden Wert der Systemeigenschaft ersetzt.

Hierzu ein Beispiel:

```
permission java.io.FilePermission "${user.home}","read";
```

Quelltext 11 Beispiel zu Variablen in Policy bzw. Security Properties Dateien

Der Ausdruck "`${user.home}`" wird durch den entsprechenden Ausdruck der Systemeigenschaft "`user.home`" ersetzt. Ist der Wert z.B. "`/home/wi97/kkoehler`" würde der Ausdruck unter Quelltext 11 zu folgendem äquivalentem Ausdruck führen:

```
permission java.io.FilePermission "/home/wi97/kkoehler","read";
```

Quelltext 12 Beispiel 2 zu Variablen in Policy bzw. Security properties Dateien

Weiterhin besteht die Möglichkeit für plattformunabhängige Dateien mit "`${/}`" den sogenannten File Separator anzugeben. Der File Separator wird für jede Plattform in einer anderen Weise interpretiert. So wird in einem Windows System der Backslash (\) und in einem UNIX System der normale Slash (/) verwendet. Auch hier ein kleines Beispiel:

```
permission java.io.FilePermission "${user.home}${/}*","read";
```

Quelltext 13 Beispiel zum File Separator

[17] die hier genannte Domäne hat nichts mit der Protection Domain von weiter oben zu tun!

Das Beispiel würde zu folgenden Interpretationen führen:

```
Unter Windows:
permission java.io.FilePermission "c:\users\kkoehler\*","read";

Unter UNIX:
permission java.io.FilePermission "/home/wi97/kkoehler/*","read";
```

Quelltext 14 Beispiel zur File Separator Interpretation unter Windows und UNIX

Eine Ausnahme, bei der man keine File Separatoren verwenden sollte, stellen die codeBase Angaben dar. Hier werden alle Separatoren automatisch zu normalen Slashs (/) umgewandelt. Dies liegt darin begründet, daß alle codeBase Angaben URLs sind.
Beispiel:

```
grant codeBase "file:/${java.home}/lib/ext/"
```

Quelltext 15 Beispiel codeBase Separator

Unter Windows würde, auch wenn `java.home` auf `c:\jdk1.2.2` steht, die Interpretation unter Quelltext 16 herauskommen.

```
grant codeBase "file:/C:/jdk1.2.2/lib/ext/"
```

Quelltext 16 Beispiel codeBase Separator unter Windows

Die Verwendung dieser Variablen ist überall dort erlaubt, wo durch Anführungszeichen hervorgehobener Text angegeben werden kann. Also auch in den `signedBy`, `target names`, und `action` Feldern.

In der Security Properties Datei kann angegeben werden, ob die Verwendung von Variablen erlaubt sein soll oder nicht. Dazu muß der Eintrag "`policy.expandProperties`" entweder auf `true` (Standardwert) oder `false` gesetzt werden. `True` steht hierbei für die Erlaubnis, diese Variablen verwenden zu dürfen.

Das Schachteln von Variablen ist nicht erlaubt!
So z.B. "`${user.${wasanderes}}`".

Zum Schluß muß noch erwähnt werden, daß unter Windows für einen Backslash immer 2 Blackslashes angegeben werden müssen!
"`c:\\users\\kkoehler\\startTheEngine.bat`" wird so zu "`c:\users\kkoehler\startTheEngine.bat`" und wird auch so interpretiert[18].

[18] Strings werden mit Hilfe eines sogenannten StreamTokenizer verarbeitet, der den Backslash als Escape Zeichen ansieht. Nur doppelt angegebene Backslashes werden als einfacher Backslash interprepiert!

3.3.5. Policytool

Da es bei der Eingabe der Policy Dateien im "reinen" ASCII Format sehr leicht zu Fehlern kommen kann, liefert Sun seit der Version 1.2 des JDKs ein grafisches Werkzeug zum editieren der Policy Dateien mit. Das `policytool` unterstützt den Anwender dabei, die Dateien zu erzeugen bzw. zu verwalten. In Abbildung 3-3 sieht man das Policytool mit der standardmäßigen System Policy Datei. Es sind hier zwei Berechtigungen eingerichtet.

Klickt man auf den Edit Button[19] gelangt man zu Abbildung 3-4, in der die Berechtigungseinträge angezeigt werden. Diese Einträge im ASCII Format werden in Quelltext 2 gezeigt.

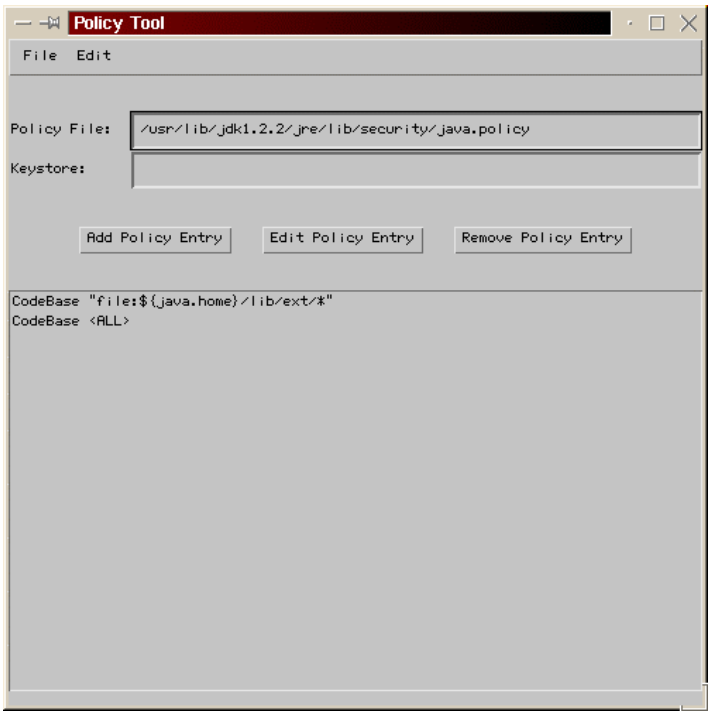

Abbildung 3-3 Policytool im KDE unter Linux

[19] die Edit Schaltfläche ☺

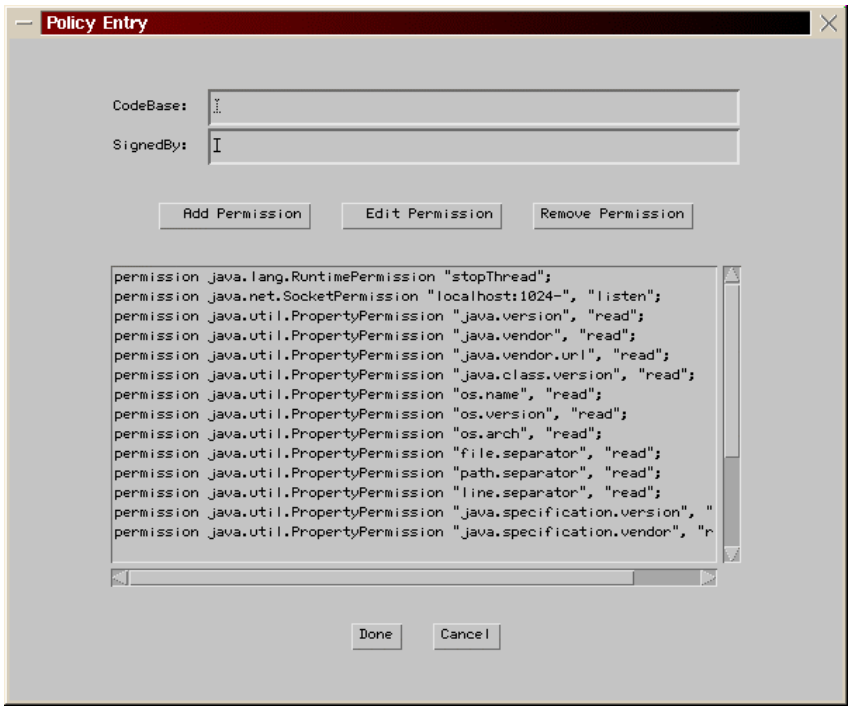

Abbildung 3-4 Policytool - Policy Entry - KDE unter Linux

Die Masken zum Neuanlegen von Einträgen ist genauso intuitiv gestaltet wie die gezeigten Masken zur Ausgabe der Policy Dateien. Aus diesem Grund soll darauf an dieser Stelle nicht weiter eingegangen werden. In Abbildung 3-5 sieht man die leere Eingabemaske. Die Werte links entsprechen den im vorherigen Abschnitt vorgestellten Feldern[20] in der Policy Datei. Bei der Eingabe wird man durch "Vorschlagswerte" unterstützt.

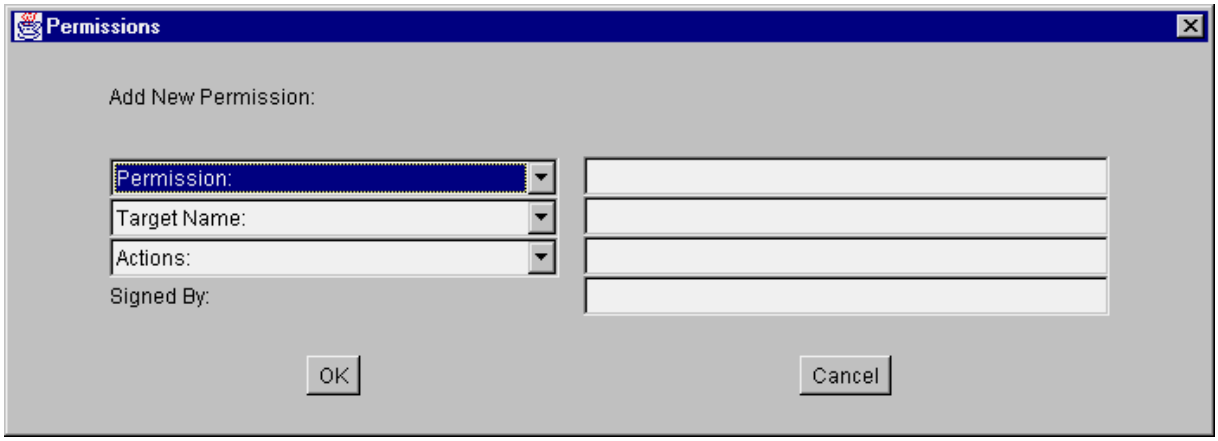

Abbildung 3-5 Eine neue Berechtigung aufnehmen[21]

[20] Permission, target_name und actions
[21] http://developer.java.sun.com/developer/onlineTraining/Security/Fundamentals/images/ptAddPerm.gif

Abbildung 3-6 zeigt ein Beispiel für eine ausgefüllte Makse. Sie würde wie folgt in der Policy Datei wieder auftauchen[22]:

```
grant {
       permission java.util.PropertyPermission "java.io.tmpdir", "read";
};
```

Quelltext 17 ASCII Entsprechung zum Beispiel

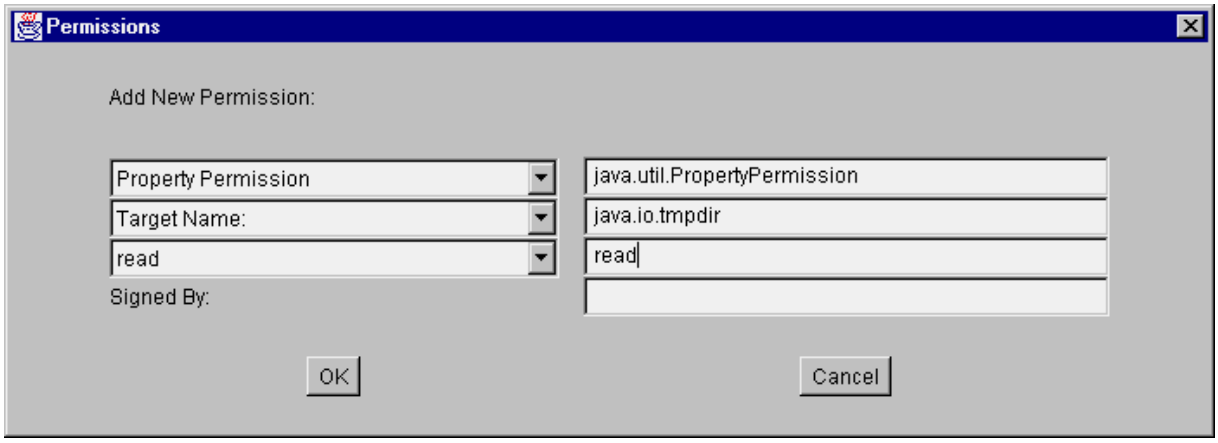

Abbildung 3-6 Beispiel einer ausgefüllten Maske[23]

3.3.6. Vorgehen beim Bestimmen von Berechtigungen

Unter 3.1.4 wurde bereits auf domainübergreifende Methodenaufrufe eingegangen. Es wurde hierbei vorausgesetzt, daß es bestimmte Berechtigungen für die verschiedene Domains gibt. Wie kommen diese aber zustande? Wie ist das Vorgehen beim Bestimmen von Berechtigungen?

Es ist prinzipiell möglich, daß ein Objekt zu mehreren Einträgen in der Policy Datei "paßt". Dies ist im besonderen dann der Fall, wenn man mit Wildcards ("*") arbeitet. Eine Klasse, die sich z.B. auf dem Rechner java.subdomain.domain befindet, kann, wenn es einen Policyeintrag für *.domain und *.subdomain.domain gibt, zu beiden Domains gehören.

Wichtig ist in diesem Zusammenhang, daß die Überprüfung, zu welcher Domain eine Klasse gehört, für jede Klasse individuell durchgeführt wird.
Folgender Algorithmus wird für die Überprüfung verwendet[24]:

- Überprüfe die öffentlichen Schlüssel, wenn der Code unterzeichnet ist.
- Wenn der Schlüssel in der Policy Datei nicht mit dem gegebenen übereinstimmt, ignoriere den Schlüssel
  Wenn jeder Schlüssel ignoriert wurde, behandle den Code als nicht unterzeichnet.
- Wenn der/die Schlüssel anerkannt wird/werden, oder kein Unterzeichner angegeben wurde
  {
  Versuche alle URLs in der Policy Datei zu überprüfen
  }
- Wenn weder die Schlüssel noch die URL mit einem Wert in der Policy Datei übereinstimmt, verwende die Standardberechtigung ("original" Sandkasten).

[22] ein Programm könnte jetzt den Ort des temporären Verzeichnis erfragen
[23] http://developer.java.sun.com/developer/onlineTraining/Security/Fundamentals/images/ptAddPermFilled.gif
[24] http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc3.html

Die genaue Bedeutung der angegebenen codeBase hängt stark vom letzten Zeichen der Zeichenkette ab. Endet die Zeichenkette mit einem "/" sind, außer JAR Dateien, alle Dateien im Verzeichnis angesprochen. Alle Dateien, also Class und JAR Dateien, sind angesprochen, wenn die Zeichenkette mit einem "*" endet. Eine abschließendes "/-" spricht alle Dateien in allen Unterverzeichnis, ausgehend vom angegebenen Verzeichnis, an.

3.3.7. Beispiele und Anwendungen

Im folgenden sollen kurze Beispiele gezeigt werden, wie Policy Files verwendet werden können und welche Berechtigung sich daraus ergeben.

Wichtig ist, daß bei einem standardmäßigen Aufruf der JVM aus der Kommandozeile die Sicherheitsrichtlinien **nicht** greifen! Hierzu muß ein Security Manager installiert sein, der auf die entsprechenden Policy Files zugreift.

Dies erreicht man durch folgenden Kommandozeilenargumente bei Aufruf des Programmes:

```
java -Djava.security.manager AppName
```

Quelltext 18 Installieren des SecurityManagers über die Kommandozeile

Beim gezeigten Aufruf unter Quelltext 18 wird der mitgelieferte SecurityManager für dieses Programm (*AppName*) eingerichtet. Welche Policy Dateien verwendet werden wird in der Datei `java.security` definiert. Es besteht aber auch die Möglichkeit weitere Policy Dateien entweder zusätzlich oder ausschließlich beim Aufruf zu definieren.
Dies erreicht man über folgende Kommandozeilenargumente:

```
java -Djava.security.manager -Djava.security.policy=polURL AppName

                                        oder

java -Djava.security.manager -Djava.security.policy==polURL AppName
```

Quelltext 19 Definition von Policy Dateien in der Kommandozeile

Der Unterschied der beiden in Quelltext 19 gezeigten Kommandozeilenargumente liegt in der Zuweisung der Policy Datei. Verwendet man nur ein Gleichheitszeichen, so wir die genannte Policy Datei zusätzlich zu der bereits installierten (standardmäßigen) Policy Datei verwendet. Ruft man hingegen das Kommandozeilenargument mit doppelten Gleichheitszeichen auf, so wird nur noch die genannte Policy Datei verwendet. Alle anderen werden ignoriert.

Möchte man im Appletviewer Policy Dateien verwenden, erreicht man das durch folgenden Aufruf:

```
appletviewer -J-Djava.security.policy=polURL AppletName
```

Quelltext 20 Definition einer Policy Datei für den Appletviewer

3.4. Sicherheitsmanagement

In den letzten Abschnitten wurde näher darauf eingegangen, wie man Zugriffe auf die verschiedensten Ressourcen unterbinden bzw. einschränken kann. Dies war aber nur eine Seite des Sicherheitsmanagement der Java Plattform. Wie diese Sicherheitseinstellungen zur Laufzeit "durchgesetzt" werden, wird im folgenden näher beschrieben.

3.4.1. Security Manager

Jede virtuelle Maschine besitzt einen Security Manager, der für die Einhaltung der Sicherheitsrichtlinien verantwortlich ist. Allerdings kann jede virtuelle Maschine nur einen Security Manager installieren/ laden. Es spielt dabei keine Rolle, ob es sich dabei um einen selbstgeschrieben[25] oder um den standardmäßigen handelt.
Ist ein Security Manager einmal installiert, kann er nicht mehr durch einen anderen ersetzt werden. Wäre dies der Fall, wäre jede Anwendung in der Lage, ihren eigenen, nicht ganz so strengen, Security Manager zu installieren und somit die Sicherheit des Systems, indem sie die gegebenen Sicherheitsrichtlinien unterläuft, zu gefährden.
Wie schon unter 3.3.7 erwähnt wird bei einem standardmäßigen Aufruf der JVM aus der Kommandozeile kein Security Manager installiert. Dies muß "per Hand" erfolgen. Den entsprechenden Aufruf zeigt Quelltext 18. Weiterhin besteht die Möglichkeit einen eigenen Security Manager über das Kommandozeilenargument anzugegeben (vgl. Quelltext 21).

```
java -Djava.security.manager=de.rzfun.MySecurityManager
```

Quelltext 21 Angabe eines eingenen Security Managers

Lädt man keinen Security Manager, hat die Anwendung volle Kontrolle über die System Ressourcen. Um diese lockere Handhabung zu ändern, muß die Anwendung ihren eigenen Security Manager implementieren (laden).
Jeder heute verfügbare Internetbrowser, der die Ausführung von Java Programmen ermöglicht (java-enabled) benutzt einen (eigenen) Security Manager, der den Sandkasten zur Verfügung stellt, in dem die Applets ausgeführt werden. Dieser Security Manager wird beim Start der virtuellen Maschine des Browsers gestartet.
Führt ein Java Programm eine Operation, die eventuell beschränkt sein könnte, aus, prüft die virtuelle Maschine mit Hilfe des Security Managers, ob diese Operation ausgeführt werden darf oder nicht. Wird die Operation verweigert wird eine `SecurityException` erzeugt. `SecurityExceptions` sind `RuntimeExceptions` und müssen aus diesem Grund nicht durch ein `try/catch` Konstrukt „aufgefangen“ werden.

Ein typischer Aufruf einer Sicherheitsüberprüfung mit Hilfe eines Security Managers ist in Quelltext 22 dargestellt.

```
..
SecurityManager security = System.getSecurityManager();
if (security != null) {
    security.checkXXX(argument,  . . . );
}
..
```

Quelltext 22 typischer Aufruf einer Sicherheitsüberprüfung mit Hilfe des Security Managers

Ein Security Manager stellt eine Vielzahl von `checkXXX`[26] Methoden zur Verfügung, mit denen die Zugriffsberechtigungen geprüft werden. Um den Security Manager zu erweitern, kann man beliebig viele `checkXXX` Methoden hinzufügen. Man ist dann

[25] `SecurityManger` ist eine Klasse im `java.lang` Paket, von der Unterklassen gebildet werden können
[26] die XXX müssen durch einen individuellen Namen ersetzt werden!

allerdings an einen Security Manager gebunden, in dem alle Methoden implementiert sind, die man zur Überprüfung der Sicherheitsrichtlinien benötigt.

Ein Beispiel soll zeigen, was geschieht, wenn eine Anwendung Daten aus einer Datei lesen möchte (Die Anwendung führt eine `read()` Methode der `FileInputStream` Klassen aus). Der Sourcecode der `read()` Methode geht folgendermaßen vor:

- Prüft, ob ein Security Manager geladen ist.
- Wenn ein Security Manager geladen ist, ruft er (der Sourcecode) die `checkRead()` Methode auf.
- Wenn die checkRead() Methode "ohne Fehler zurückkehrt"[27], dann wird der Lesevorgang fortgesetzt.
- Wenn die `checkRead()` Methode die Berechtigung nicht erteilt, wird eine SecurityException erzeugt.

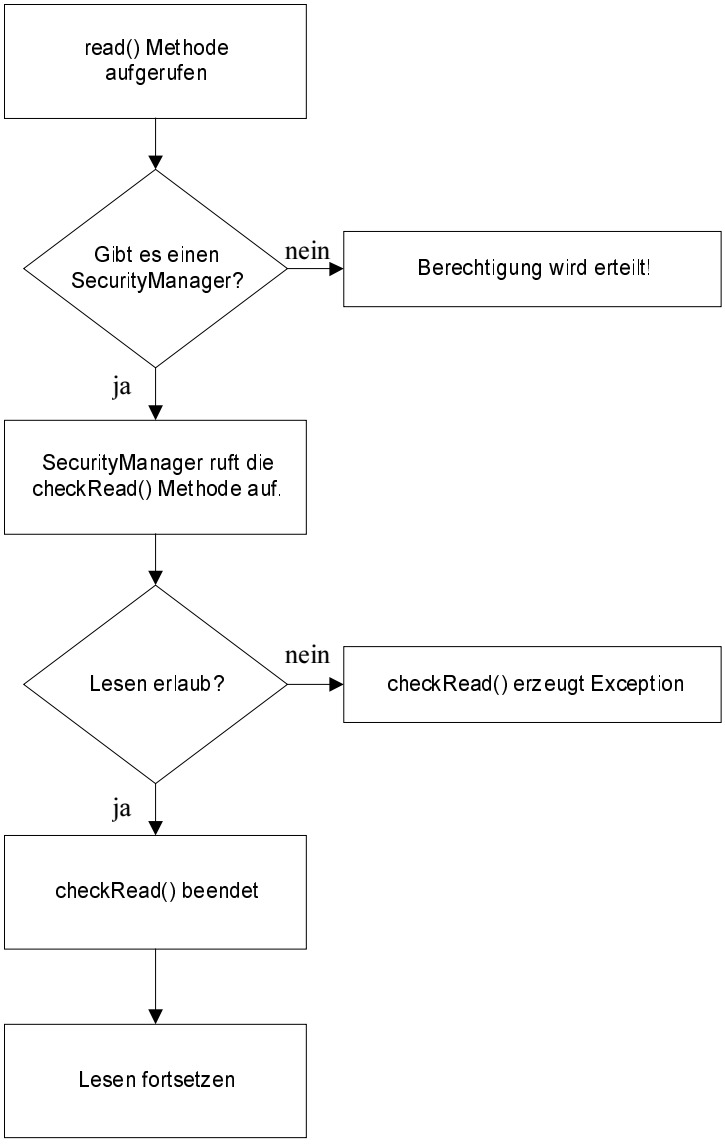

Abbildung 3-7 Prüfen von Berechtigungen

[27] Es wurde keine Exception erzeugt

3.4.2. Access Controller

Die hier beschriebenen Sicherheitsmechanismen mit dem Security Manager sind auch in Java 2 noch einsetzbar. Allerdings ist dies nur aus Gründen der Abwärtskompatibilität zu den älteren JDK Versionen der Fall.
In der Java 2 Plattform trifft der Access Controller alle Zugriffsberechtigungsentscheidungen. Er besitzt im Grunde drei Aufgaben[28]:

- Entscheidung darüber, ob der Zugriff auf bestimmte System Ressourcen erlaubt werden kann oder nicht. (Diese Entscheidung wird mit Hilfe der Policy Dateien getroffen).
- Codeausschnitte als "privileged" zu markieren und alle darauf aufbauende Zugriffe dementsprechend zu beeinflussen.
- Einen "Schnappschuß" des gerade aktuellen Kontextes ("Umgebung"/ Protection Domain) zu erhalten, um damit Zugriffsentscheidungen anderer Kontexte in bezug auf den gesicherten Kontext zu bilden.

Wie der AccessController seine drei Aufgaben erfüllt, wird weiter unten näher erläutert.

Der Aufruf zur Überprüfung von Sicherheitsrichtlinien sieht z.B. folgendermaßen aus:

```
..
FilePermission perm = new FilePermission("path/file","read");
AccessController.checkPermission(perm);
..
```

Quelltext 23 Aufruf der Sicherheitsüberprüfung mit Hilfe des AccessControllers

Auch hier gilt:
Wenn die Berechtigung nicht erteilt wird, wird eine Exception erzeugt (AccessControllerException[29])!

3.4.2.1. Algorithmus zum Überprüfen von Berechtigungen

Wie schon unter 3.1.4 besteht die Möglichkeit, daß sich eine Anweisungseinheit über mehrere Protection Domains erstreckt. Der AccessController hat in diesem Fall dafür zu sorgen, daß die entsprechende Anwendung genau die Rechte bekommt, mit denen sie ausgestattet wurde. Hier noch einmal kurz die Faustregel, nach der in solch einem Fall vorgegangen wird:

> Wenn (mindestens) eine Methode in der Anweisungskette nicht die jeweilige Berechtigung besitzt, wird eine `AccessControllerException` erzeugt ("kleinster gemeinsamer Nenner"). Dies ist nicht der Fall, wenn die Methode als "privileged" gekennzeichnet ist und alle anderen Methoden, die von ihr im folgenden aufgerufen werden, die jeweilige Berechtigung besitzen.

[28] http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc4.html
[29] AccessControllerException ist eine vererbte Klasse von java.lang.SecurityException

Für die Vorgehensweise beim Bestimmen von Berechtigungen gibt es zwei Implementationsstrategien:

- Die "eager evaluation"[30] Implementation.
  Wie der Name schon sagt, handelt es sich hierbei um eine "eifrige" Vorgehensweise. Kommt ein Thread zu einer Protection Domain oder stammt er aus ihr, wird die Berechtigung, die sich daraus ergibt, sofort dynamisch bestimmt. Jeder Thread ist somit immer mit der jeweiligen Berechtigung ausgestattet.
- Die "lazy evaluation"[31] Implementation.
  Auch hier spricht der Name Bände. Bei der "faulen" Berechnung wird die Berechtigung eines Threads erst dann bestimmt, wenn es zu einer Berechtigungsanfrage kommt.

Beide Implementationen haben Vor- und Nachteile. Bei der "eifrigen" Vorgehensweise ist die Berechtigungsbestimmung relativ einfach und in vielen Situationen recht schnell. Der Nachteil dieser Methode besteht allerdings darin, daß domainübergreifende Methodenaufrufe, für die zur Zeit des Aufrufs noch Rechte "berechnet" werden müssen, öfter stattfinden, als die schnellen Berechtigungsbestimmungen, die schon "im Vorfeld" geklärt sind, und somit ein großer Prozentsatz der dynamischen Berechtigungszuordnungen nutzlos ist.

Der Nachteil der "faulen" Variante besteht in der Geschwindigkeit bei "normalen" Berechtigungsbestimmungen, also der nicht domainübergreifenden, da diese erst zum Zeitpunkt der Anfrage berechnet werden.

Sun Microsystems sieht aber in der zweiten Variante den ökonomischeren Ansatz und verwendet aus diesem Grund bei der Bestimmung von Berechtigungen die "lazy evaluation" Implementation.

Folgender Quelltext stellt den Algorithmus (Pseudocode) dar, der in der `checkPermission` Methode verwendet wird um Berechtigungen zu bestimmen.

```
i = m;
while (i > 0) {
   if (die Domain der Methode i verfügt nicht über die Berechtigung)
       throw AccessControlException
   else if (Methode i ist als "privileged" gekennzeichnet)
       return;
   i = i - 1;
};
```

Quelltext 24 Pseudocode Algorithmus zum Bestimmen von Berechtigungen

Die zu prüfende Anweisung in Quelltext 24 erstreckt sich über m Methoden. Methode m ruft in diesem Beispiel die `checkPermission` Methode auf. In der `checkPermission` Methode wird jetzt nacheinander geprüft, ob die jeweiligen Berechtigungen vorliegt. Liegt keine Berechtigung vor, wird eine Exception erzeugt.

3.4.2.2. Umgang mit "privileged"

Unter 3.1.4 wurde auch schon kurz auf die `doPrivileged` Methode eingegangen. Hier soll der Umgang mit „`privileged`“ näher betrachtet werden. Mit ihr ist es möglich, Codefragmente kurzzeitig mit mehr Rechten auszustatten, als ihrer Anwendung normalerweise zustehen würden. Diese Methode ist neu im Sicherheitsmanagement der Java Plattform. Es handelt sich um eine `static method` in der AccessController Klasse, mit der der AccessController darüber informiert wird, daß der Code als "`provileged`" gekennzeichnet ist, und daß, egal von welcher Methode er aufgerufen wurde, der Code auf seine zugestandenen Ressourcen zugreifen kann.

[30] frei übersetzt "eifrige Wertberechnung"
[31] frei übersetzt "faule Wertberechnung"

Um eine Ausführungseinheit als `privileged` zu kennzeichnen muß man die `doPrivileged` Methode verwenden. Ist eine Anweisungseinheit auf diese Weise markiert, und besitzt sie die nötigen Berechtigungen, bricht der AccessController die weiteren Sicherheitsüberprüfungen für diese Methode ab, und gibt somit die „Ausführung frei“. Diese Vorgehensweise sieht man auch recht gut im Pseudocode des Quelltext 24.

Quelltext 25 zeigt den prinzipellen Aufbau einer Methode, die ein Codefragment als `privileged` kennzeichnen möchte.

```
eineMethode() {
       ...normaler Code ...
       AccessController.doPrivileged(new PrivilegedAction() {
           public Object run() {
               // hier kommt als privileged gekennzeichneter Code:
               System.loadLibrary("awt");
               return null;
           }
       });
       ...und hier kommt wieder normaler Code...
  }
```

Quelltext 25 Aufruf der doPrivileged Methode

Der Methode `doPrivileged` muß ein Argument des Typs `PrivilegedAction` übergeben werden. `PrivilegedAction` ist ein Interface mit einer einzigen (`run()`) Methode[32]. Beim Aufruf der `doPrivileged` Methode wird die `run` Methode der Instanz des `PrivilegedAction` Objekts ausgeführt.

Die Methode, die als privileged gekennzeichnet ist, **muß** über die Berechtigung verfügen, die die Anweisungen im `doPrivileged` Teil zur Ausführung benötigen.

Der Rückgabewert der `run` Methode stellt gleichzeitig den Rückgabewert der `doPrivileged` Methode dar. Da im obigen Beispiel die `run` Methode keinen Rückgabewert besitzt, wird der Rückgabewert der `doPrivileged` Methode vernachlässigt. Falls die doPrivileged Methode einen Rückgabewert besitzt, kann dieser wie jeder andere Rückgabewert einer Methode behandelt werden. Es wird aber voraussichtlich ein `cast` nötig sein, bevor man mit dem Objekt weiterarbeiten kann, da es sich immer um einen Rückgabewert des Typs `Object` handelt.

Besteht die Möglichkeit, daß die Methode eine „checked“[33] Exception erzeugt, muß anstatt des Interfaces PrivilegedAction das Interface PrivilegedExceptionAction verwendet werden. Quelltext 26 zeigt hierzu ein Beispiel.

[32] Im obigen Beispiel handelt es sich um eine anonyme innere Klasse, die dieses Interface implementiert.
[33] Eine Exception die bei der Methoden Deklaration unter throws aufgeführt wird.

```
somemethod() throws FileNotFoundException {
        ...normal code here...
    try {
         FileInputStream fis = (FileInputStream)
         AccessController.doPrivileged( new PrivilegedExceptionAction() {
               public Object run() throws FileNotFoundException {
                     return new FileInputStream("eineDatei");
               }
         } );
    } catch (PrivilegedActionException e) {
        //  hier muß eine FileNotFoundException erzeugt werden!
        throw (FileNotFoundException) e.getException();
     }
         ...normal code here...
}
```

Quelltext 26 Anwendung des Interfaces PrivilegedExceptionAction

Das Konzept eines doPrivileged Methodenaufrufs existiert nur innerhalb eines Threads. Ist der Aufruf beendet, werden die Berechtigungen, die kurzzeitig ausgesprochen worden sind, wieder zurückgenommen.

Wichtig ist auch, daß, wenn innerhalb der `doPrivileged` Methode nicht vertrauter und mit weniger Rechten ausgestatteter Code ausgeführt wird, dieser nicht mehr Rechte erlangt, als ihm zustehen. Nur im Fall, daß die `doPrivileged` Methode die Rechte besitzt und alle darauf folgenden Methoden (in ihr aufgerufenen Methoden) auch, wird keine Exception erzeugt („die Berechtigung erteilt“).

3.4.2.3. Vererbung des AccessController Kontextes

Wenn ein neuer Thread erzeugt wird, wird gleichzeitig auch ein neuer Stack erzeugt. Würde jetzt der aktuelle AccessController Kontext nicht vererbt werden, würde bei einem Aufruf von checkPermission innerhalb des neuen Threads, eine Sicherheitsentscheidung nur auf Grund des Kontextes des neuen Threads und nicht auf Grundlage des Kontext des erzeugenden Threads, dem Eltern Thread, getroffen werden.

Dieser Ansatz ist eigentlich nicht weiter schlimm. Er stellt bis an diese Stelle kein Sicherheitsproblem dar. Das Entwickeln von sicherheitsrelevantem Code, besonders System Code, ist in solch einen Fall nur wesentlich fehleranfälliger. Ein kleines Beispiel hierzu:

Ein Entwickler nimmt an, daß ein „Kind Thread“ den Sicherheitskontext, also alle Berechtigungseinstellungen, seines „Eltern Threads“ erbt. Im „Kind Thread“, dem in diesem Fall vertraut werden würde, wird dann auf Ressourcen zugegriffen und für alle anderen zugreifenden Methoden freigegeben. Jetzt könnte der „Eltern Thread“, dem in diesem Beispiel nicht vertraut werden würde, ohne Berechtigung auf die Ressource zugreifen. Diese Konstellation würde „unbeabsichtigte“ Sicherheitslöcher öffnen.

Aus diesem Grund wird beim Erzeugen eines neuen Threads immer der Sicherheitskontext des „Eltern Threads“ mit vererbt, um sicherzustellen, daß keine Methode durch Erzeugen eines neuen Threads mehr Rechte erlangen kann, als ihr normalerweise zustehen würde.

Folge daraus ist, daß der logische Kontext des Threads, die Kontexte beider Threads enthält, den Kontext des Eltern Threads und des neuen Threads, die dann auch überprüft werden müssen.

Es ergibt sich daraus die Berechtigungsüberprüfung in Quelltext 27.

```
i = m;
while (i > 0) {
   if (die Domain der Methode i verfügt nicht über die Berechtigung)
       throw AccessControlException
   else if (Methode i ist als "privileged" gekennzeichnet)
       return;
   i = i - 1;
};

 // Als nächstes wird der Kontext, der beim Erzeugen vererbt wurde überprüft.
 // Wenn ein neuer Thread erzeugt wird, wird in "inherited"
 // der AccessControlContext des Ausführungszeitpunktes des Eltern Thread im
 // Kind Thread gespeichert.

 inheritedContext.checkPermission(permission);
```

Quelltext 27 Erweiterung des Pseudocode Algorithmus zum Bestimmen von Berechtigungen

Zu Beachten ist:
Die Vererbung endet nicht bei der ersten Vererbung. Sie wird weitergeführt. Es gibt sozusagen auch Enkel.
Der vererbte Kontext wird zum Zeitpunkt der Erzeugung des Threads und nicht bei der ersten Ausführung "aufgenommen".

3.4.2.4. AccessControlContext

Jetzt ergibt sich ein weiteres Problem. Normalerweise führt die checkPermission Methode Sicherheitsüberprüfungen im aktuell ausführenden Thread aus. Was passiert aber, wenn die Sicherheitsüberprüfung nicht im aktuellen Kontext stattfinden kann, sondern in einem anderen Kontext ausgeführt werden muß? Als Beispiel: Ein Thread schickt ein Ereignis an einen anderen Thread. Der angesprochene Thread benötigt jetzt Sicherheits-informationen, um entscheiden zu können, ob die angefragte Aktion ausgeführt werden kann oder nicht. Es handelt sich beim zweiten Thread um einen komplett anderen Kontext.

Um genau diese Problematik zu lösen, gibt es die Klasse `AccessControlContext`, die einen Schnappschuss der Berechtigungsstrukturen darstellt. Die Klasse `AccessControll` stellt eine Methode `getContext` zur Verfügung, mit der solche Schnappschüsse erstellt werden können. Mit Hilfe dieses Schnappschusses können Berechtigungsentscheidungen eines anderen Threads überprüft werden. Man muß hierzu dem aufgerufenen Thread nur noch den `AccessControlContext` als Argument mit übergeben.

Auch die Klasse `AccessControlContext` besitzt eine Methode `checkPermission`, mit der die eigentliche Überprüfung ausgelöst wird und somit als Argument übergebene „Kontexte überprüft“ werden können.

Aus diesen Überlegungen ergibt sich der Pseudocode der `checkPermission` Methode, der im AccessController verwendet wird.

```
i = m;
while (i > 0) {
   if (caller i's domain does not have the permission)
       throw AccessControlException
   else if (caller i is marked as privileged) {
           if (a context was specified in the call to doPrivileged)
              context.checkPermission(permission);
           return;
   }
   i = i - 1;
};


        // Next, check the context inherited when // the thread was created. Whenever a new
thread is created, the // AccessControlContext at that time is // stored and associated
with the new thread, as the "inherited" //

       context. inheritedContext.checkPermission(permission);
```

Quelltext 28 Verwendeter Pseudocode in der checkPermission Methode

3.4.3. SecurityManager im Vergleich zum AccessController

Der in der Java 2 Plattform neu eingeführte AccessController ist voll zum „alten“ SecurityManager abwärtskompatibel. Alle check Methoden sind auch im AccessController weiter einsetzbar. Allerdings führt der neue SecurityManager die Sicherheitsüberprüfung mit Hilfe des AccessController aus (vgl. Quelltext 23).

Um die Beziehung zwischen dem SecurityManager und dem AccessController zu verstehen, muß man sich im klaren sein, daß der SecurityManager die zentrale Rolle der Sicherheitsüberprüfungen darstellt, und der AccessController „nur“ besondere Zugriffskontrollalgorithmen darstellt. Hier sind Methoden wie die `doPrivileged` implementiert.

Mit dem AccessController lassen sich relativ leicht eigene Berechtigungen einführen und später dann auch sehr gut verwalten.

Wollte man im SecurityManager neue Berechtigungen einführen, mußte man einen neuen SecurityManager implementieren und seine eigenen `checkXXX` Methoden hinzufügen. Diese relativ umständliche und zum Teil auch sehr sicherheitskritische Vorgehensweise ist im Mechanismus mit dem AccessController nicht mehr nötig. Hier implementiert man eine neue Permission Klasse und kann diese dann recht einfach einsetzen und verwalten. Es sind keine sicherheitskritischen Änderungen der eigentlichen „Überprüfungsinstanz“ mehr nötig.

4. Kommentar

Möchte man sämtliche Sicherheitsmechanismen der doch recht komplexen Plattform darstellen, wäre der hier vorgeschriebene Rahmen recht schnell gesprengt. Aus diesem Grund habe ich nur einen relativ kleinen, aber meiner Ansicht nach recht wichtigen Bereich, herausgenommen.

In meinen Ausführungen bin ich nur darauf eingegangen, daß sich die Programme schon in der virtuellen Maschine befinden. Bis sie dort sind, müssen sie auch einige Sicherheitsinstanzen überwinden. An dieser Stelle soll, um den Umfang dieses Bereichs darzustellen, noch einmal dargestellt werden, welche Schritte das im groben sind:

Sicherheit in der Java Plattform beginnt schon bei der Sprache an sich. So sind z.B. Pointer wie sie aus C/C++ bekannt sind in Java kein Thema. Durch das Unterbinden solcher Anweisungen besitzt Java schon als Sprache ein sehr hohes Sicherheitsniveau. Daß diese Sprachkonstrukte auch eingehalten werden ist wiederum die Aufgabe des Compilers. Auch hier muss auf Sicherheit geachtet werden. Durch Einsatz eines „bösartigen“ Compilers kann man auch Java Bytecode erzeugen, der ein System gefährden kann. Aber auch hier bietet die Java Plattform eine Möglichkeit, sich gegen solche Angriffe zu schützen. Der sogenannte Bytecode Verifier prüft beim Laden des Codes durch den ClassLoader den kompletten Code, ob er auch den Konventionen der Sprache Java genügt.

Es sind also bis ein Programm ausgeführt werden kann viele einzelne Schritte notwendig, bei denen wiederum Sicherheitsmechanismen implementiert sind bzw. sein müssen.
An dieser Stelle kann nur angesprochen werden, daß man bei der Berechtigungsfreigabe über die Policy Dateien genau überlegen soll, was man denn gerade freigibt. Wie erwähnt, können auf den ersten Blick recht unbedeutende Berechtigungen, wie unter Quelltext 29 zu sehen, zu großen Sicherheitsproblemen führen.

```
java.lang.RuntimePermission "createClassLoader"
```

Quelltext 29 createClassloader Permission

Weitere Themen wie Verschlüsselung oder digitale Signaturen (Unterzeichnen von Applets) wurden hier nicht erwähnt.

Meiner Ansicht nach, bietet Java eine sichere Plattform, die sich in näherer Zukunft noch weiter verbreiten wird. Gerade die Möglichkeit Programme über das Internet zu laden und jedes Programm individuelle Sicherheitsrichtlinien zu unterwerfen, stellt eine perfekte Möglichkeit dar, Programme von fremden Personen nur mit Rechten auszustatten, die die Systemsicherheit nicht gefährden. Andererseits kann man auch Programmen, die aus „gesicherter“ Quelle kommen mehr Rechte einräumen. So steht in Zukunft der Verbreitung von „größeren“ Programmen über das Internet nichts mehr im Wege.
Eine sichere Plattform steht zur Verfügung.

Literaturverzeichnis

Heid, Jörn	Offene Gesellschaft, iX, November 1999, Seite 202 ff
Gong, Li	Java Security Architecture „Online im Internet“, Sun Microsystems 1998 http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc1.html http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc2.html http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc3.html http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc4.html http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc5.html http://developer.java.sun.com/developer/onlineTraining/Security/Fundamentals/Security.htmlh ttp://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc7.html http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc8.html http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc9.html http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc10.html http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc11.html Abfragedatum: 24.12.1999
MageLang Institute	Fundamentals of Java Security „Online im Internet“, Sun Microsystems 2000 http://developer.java.sun.com/developer/onlineTraining/Security/Fundamentals/Security.html Abfragedatum: 1.2.2000
Oaks, Scott:	JAVA Security Erste Ausgabe mit kleineren Korrekturen, O´Reilly Februar 1999
(ohne Verfasser)	sun educational services – Implementing Java Security Revision A, SunService, Februar 1998