Java-Security

Vorlesung *Firewalls und andere IT-Sicherheitsmechanismen*

Kai Schmitz-Hofbauer André Große Bley

7. Februar 2003

Inhaltsverzeichnis

(Kapitel 1-6 wurden von Kai Schmitz-Hofbauer verfasst.

Kapitel 7-11 wurden von André Große Bley verfasst)

1 Einleitung

1.1 Applets und Applikationen

Bei Java handelt es sich um eine von der Firma Sun Microsystems entwickelte, objektorientierte, plattformunabhängige Programmiersprache, die seit 1995 auf dem Markt ist. In Java geschriebene Programme lassen sich grob in zwei Arten von Programmen unterteilen, nämlich in Applikationen und Applets. Java-Applikationen sind Computer-Programme mit dem vollen Funktionsumfang, wie er auch bei anderen Programmiersprachen gegeben ist. Applikationen können als lokale Programme auf dem Rechner des Benutzers oder als verteilte Client-Server-Systeme in einem Netzwerk ausgeführt werden. Sie sind ein Pendant zu den bekannten C++-, Delphi- oder Basic-Programmen. Java-Applets werden innerhalb einer HTML-Seite dargestellt und unter der Kontrolle eines Web-Browsers ausgeführt. Sie werden in der Regel über das Internet von einem Server geladen. Da der Autor einer solchen HTML-Seite nicht von vornherein als bekannt und vertrauenswürdig angesehen werden kann und der Datentransport über das Internet generell relativ unsicher ist, müssen für Java-Applets besondere Sicherheitsvorkehrungen getroffen werden. Es muss gewährleistet werden, dass ein Java-Applet auf dem ausführenden Rechner weder Schaden anrichten noch sicherheitsrelevante Daten ausspionieren kann. [FLA99],[KR02]

1.2 Die virtuelle Maschine

Java-Quellcode-Dateien (Endung `.java`) werden mit Hilfe eines Java-Compilers nicht in den Maschinencode des jeweiligen Prozessors, sondern in sogenannten Bytecode (Dateiendung `.class`) übersetzt. Dieser Bytecode ist vergleichbar mit dem Maschinencode eines virtuellen Prozessors, der Anweisungen, arithmetische Operationen, Sprünge und Weiteres kennt. Damit aber der Programmcode des virtuellen Prozessors ausgeführt werden kann, muss ein Interpreter die Befehlsfolgen dekodieren und ausführen. Dieser wird virtuelle Maschine genannt. Der Bytecode ist portabel und plattformunabhängig. Einmal erstellter Bytecode kann auf jeder Plattform ausgeführt werden, für die es eine solche virtuelle Maschine gibt. Mittlerweile sind für alle gängigen Plattformen solche virtuellen Maschinen verfügbar. Java-fähige Browser verfügen ebenfalls über eine solche virtuelle Maschine. Da die VM für die Ausführung des Bytecodes verantwortlich ist, sind in ihr auch zahlreiche Sicherheitsmechanismen enthalten. [FLA99],[KR02]

2 Sicherheitsphilosophie

Die Firma SUN verfolgt bei Java die Sicherheitsphilosophie *Sicherheit durch Offenheit*. Das Sicherheitsmodell sowie Konzepte, Spezifikationen, Implementierungsdetails und sogar die Quellen der VM und der Java-Bibliotheken wurden veröffentlicht. Durch die so geschaffene Transparenz können Sicherheitslücken schnell lokalisiert und beseitigt werden. Von *Security by Obscurity* (Sicherheit durch Verschleierung) kann hier keine Rede sein.

3 Sprachsicherheit

Java wurde von Anfang an mit hohen Sicherheitsanforderungen entworfen. Diese Sicherheitsanforderungen wurden vor allem bei Implementierung der Sprache berücksichtigt. Im Gegensatz zu C++ stellt Java keine Möglichkeit zur Verfügung, direkt auf den Hauptspeicher zuzugreifen. Um unberechtigten Speicherzugriff zu vermeiden, wurde auf eine Pointerarithmetik verzichtet. Das gesamte Memory-Managment arbeitet voll automatisch. Die Grenzen der Speicherbereiche werden zur Laufzeit überwacht. Insbesondere bei Zugriffen auf Array-Elemente und Strings wird grundsätzlich die Einhaltung der Bereichsgrenzen geprüft. Zugriffe, die außerhalb des erlaubten Bereichs liegen, führen nicht zu undefiniertem Verhalten, sondern werden definiert abgebrochen und lösen eine Exception aus. Sicherheitslücken, die durch (provozierte) Speicherüberläufe (*BufferOverflow*) verursacht werden, sind damit nicht ohne weiteres möglich. Java ist außerdem eine streng typisierte Sprache. Typ-Umwandlungen (*type casting*) dürfen nur beschränkt eingesetzt werden. Alle Typkonvertierungen werden zur Compile- und Laufzeit geprüft und unerlaubte Umwandlungen von vornherein ausgeschlossen. Somit würde beispielsweise der Versuch, eine ganzzahlige Variable vom Typ Integer als Speicheradresse zu missbrauchen, fehlschlagen. Final deklarierte Klassen und Methoden können keine Unterklassen bilden bzw. überschrieben werden. Dies schützt schon überprüften Code vor evtl. böswilligen Veränderungen. Java verlangt die explizite Initialisierung von Variablen mit einem gültigen Defaultwert vor ihrer Verwendung und eine explizite Instanzierung von Objekten über den vorgesehenen Konstruktor. Damit wird verhindert, dass auf zufällige Speicherbereiche zugegriffen wird (unabsichtlich oder um zu spionieren). Die *Garbage Collection* (automatische Speicherbereinigung) trägt ebenfalls zur Erhöhung der Sicherheit bei. Die Garbage Collection gibt den Speicher von nicht mehr referenzierten Objekten wieder frei. So muss ein Entwickler nicht entscheiden, wann es sicher ist, Speicher wieder freizugeben. Die durch falsche Freigaben entstehenden Si-

cherheitsrisiken werden vermieden. [CT00],[OAK98]

4 Sicherheitskomponenten der VM

4.1 Der Bytecode-Verifier

Der Bytecode ist die Maschinensprache der VM. In der Spezifikation der VM ist genau festgelegt, wie ordnungsgemäßer und sicherer Bytecode auszusehen hat. Im Normalfall wird der Java-Bytecode (.class-Dateien) durch Übersetzen von Java-Quelldateien (.java-Dateien) erzeugt. Es gibt aber auch andere Möglichkeiten, Bytecode zu erzeugen. So existieren z. B. bereits Compiler (sog. Crosscompiler), die Bytecode aus Ada-, C-, Cobol- oder Delphi-Programmen erzeugen können. Weiterhin besteht die Möglichkeit, Bytecode von Hand zu erstellen. Wenn die Sprachspezifikationen von Java nur beim Compilieren mit einem Java-Compiler überprüft würden, könnte man leicht Code erstellen, der die in der Java-Spezifikation festgelegten Sicherheitsmechanismen umgeht bzw. aushebelt. Somit muss vor der Ausführung des Bytecodes sichergestellt werden, dass es sich bei dem Code um ordnungsgemäßen Bytecode handelt, er muss also verifiziert werden. Auf diese Weise wird u. a. sichergestellt, dass nur gültige Opcodes verwendet werden, dass alle Sprunganweisungen auf den Anfang einer Anweisung zeigen, dass alle Methoden mit korrekten Signaturen versehen sind und dass Ausdrücke korrekt typisiert sind. Diese Aufgabe erledigt der Bytecode-Verifier. Kann der Bytecode nicht verifiziert werden, so wird eine Fehlermeldung ausgegeben. [OAK98],[KR02]

4.2 Der Classloader

Der Classloader ist für das Lokalisieren und Laden von Java-Klassen zuständig. Außerdem führt der Classloader noch einige sicherheitsbezogene Pflichten aus. So stellt er zum einen sicher, dass Klassen aus den Paketen java.* nicht über das Netzwerk geladen werden. Dies stellt sicher, dass die VM nicht ausgetrickst werden kann, indem falsche Repräsentanten von den Standard-Java-Klassen geladen werden, die das Sicherheitsmodell durchbrechen können. Zum anderen stellt der Classloader für Klassen unterschiedlicher Herkunft (z. B. Applets von verschiedenen Servern) jeweils einen eigenen Namensraum zur Verfügung. Somit kommt es bei Klassen, die zwar den gleichen Namen haben, aber von unterschiedlichen Rechnern stammen, nicht zu Konflikten, was Einfluss auf die Sicherheit des Systems haben würde. Klassen unterschiedlicher

Herkunft werden von der VM so gegeneinander abgeschottet. Sie können nicht miteinander kommunizieren oder sich gegenseitig beeinflussen. Dies ist besonders dann von großer Wichtigkeit, wenn eines der geladenen Applets als vertrauenswürdig eingestuft wird und ihm Zugriff auf Ressourcen des Gastrechners gewährt wird. [OAK98],[CT00]

4.3 Der SecurityManager

In Java darf nicht jedes Programm auf jede Ressource (Betriebsmittel) des ausführenden Rechners zugreifen. Applets haben z. B. nur eingeschränkte Zugriffsrechte. Der SecurityManager überwacht den Zugriff auf kritische Ressourcen. Darf auf eine gewünschte Ressource nicht zugegriffen werden, so löst der SecurityManager eine SecurityException aus und verweigert den Zugriff auf das Betriebsmittel, ansonsten gestattet er Zugriff auf das Betriebsmittel. Applikationen verwenden standardmäßig keinen Sicherheitsmanager. Sie dürfen daher auf sämtliche Ressourcen zugreifen. Es besteht aber auf für Applikationen die Möglichkeit, einen SecurityManager explizit zu setzen und die Rechte der Anwendung einzuschränken. Java-Applets hingegen benutzen einen Sicherheitsmanager, der standardmäßig nur eingeschränkten Zugriff auf die Ressourcen des ausführenden Rechners gestattet. Es ist aber auch möglich, den eingeschränkten Zugriff für Applets etwas zu lockern. [SUN02a]

5 Das Sandbox-Modell und dessen Weiterentwicklung

Da das Sicherheitsmodell von Java historisch gewachsen ist, ist zum Verständnis des Sicherheitsmodells der aktuellen Java-Version *Java 2* [1] Kenntnis des ursprünglichen Sicherheitsmodelles von Vorteil. Daher wird im Folgenden zunächst das Sicherheitsmodell von Java 1.0 erklärt und anschließend dessen Weiterentwicklung aufgezeigt. Im ursprünglichen Sicherheitsmodell von Java wird zwischen vertrauenswürdigem und nicht vertrauenswürdigem Programmcode unterschieden. Im Sicherheitsmodell von Java 1.0 wurde jeder Programmcode, der von einer entfernten Netzwerkadresse stammt -wie z. B. ein Java-Applet aus dem Internet-, als nicht vertrauenswürdig und jeder lokale Code als vertrauenswürdig angesehen. Lokalem Code wird voller Zugriff auf die Systemressourcen gewährt und er kann ohne Einschränkungen ausgeführt werden. Nicht lokaler Code wird innerhalb eines abgeschotteten Bereiches, der sog. Sandbox,

[1] Java 2 entspricht Java 1.2 und höher

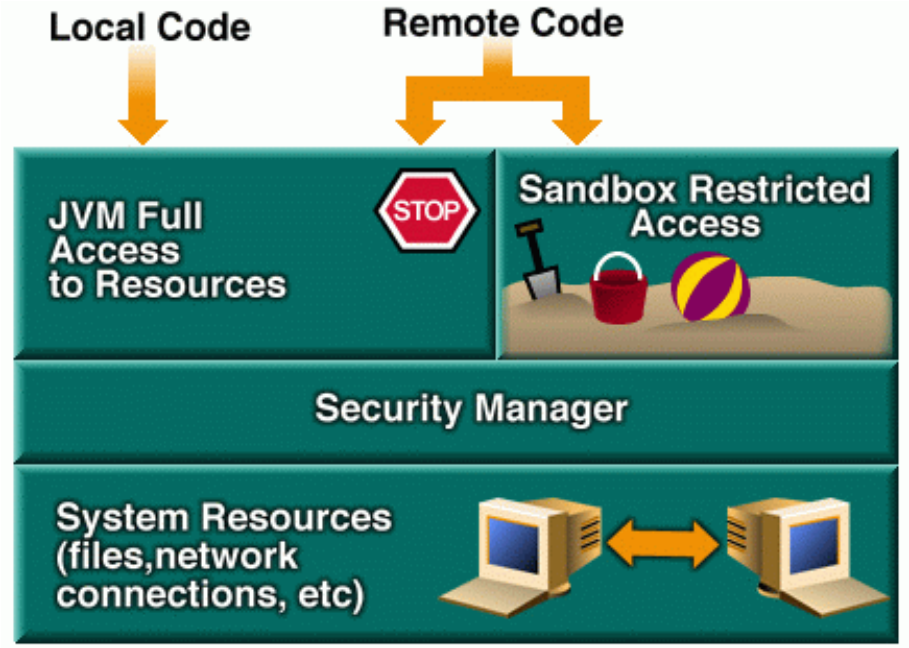

Abbildung 1: Das Sandboxmodell in Java 1.0

ausgeführt. Aus dieser Sandbox ist nur eingeschränkter Zugriff auf die Ressourcen des Gastrechners möglich (vgl. Abbildung 1). Folgendes ist entferntem Code nicht gestattet :

- Zugriff auf lokale Dateien und Verzeichnisse des Gastrechners
- Aufnehmen von Netzwerkverbindungen zu anderen Computern außer dem Heimatserver
- Ausführen von Programmen, die sich auf dem lokalen Rechner befinden
- Laden von nativen Bibliotheken
- Beenden der virtuellen Maschine
- Zugriff auf Informationen über das Java-Home-Verzeichnis, den Java-Klassenpfad, den User-Namen, das Home- und das Arbeitsverzeichnis des Anwenders

Für Applets, die von einer lokalen Quelle stammen (genauer gesagt, von einem Verzeichnis, das sich im `CLASSPATH` [2] befindet), gibt es keine Einschränkungen. Als Weiterentwicklung des ursprünglichen Sandbox-Modells besteht seit Java 1.1 die Möglichkeit, Java-Applets zu signieren. Applets mit einer bekannten Signatur gelten als vertrauenwürdig. Im Gegensatz zu normalen Applets wird ein signiertes Applet wie lokaler Code behandelt und hat daher Zugriff

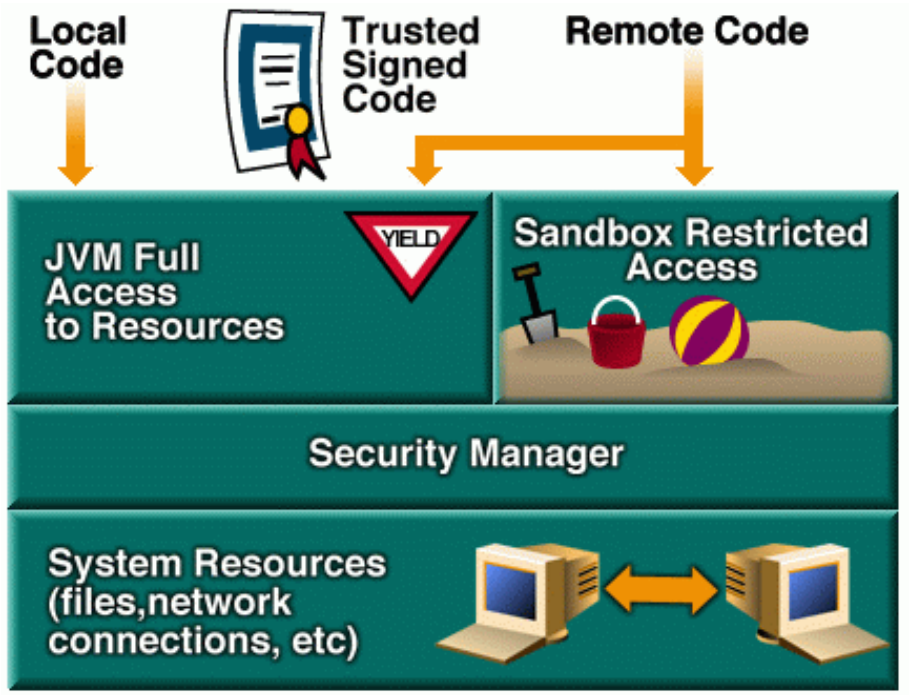

Abbildung 2: Das Sandboxmodell in Java 1.1

auf sämtliche Systemressourcen (vgl. Abbildung 2). In Java 2 wurde das Sicherheitskonzept sehr stark überarbeitet. Ein gravierender Nachteil des Sicherheitskonzeptes von Java 1.1 war, dass ein Applet entweder auf gar keine Systemressourcen Zugriff hatte oder, falls es signiert war, auf sämtliche Systemressourcen Zugriff hatte. Es gab keine Möglichkeit, einem Applet Zugriff nur auf bestimmte, ausgewählte Systemressourcen zu gewähren. Deshalb wurde das Sicherheitskonzept weiter verfeinert. Es besteht nun die Möglichkeit, Applets und Anwendungen Zugriffsrechte je nach Bedürfnissen und Anforderungen zuzusichern (vgl. Abbildung 3). Die Security Policy definiert eine Reihe von Rechten (Permissions) für Code von unterschiedlichen Unterzeichnern (mit unterschiedlicher Signatur) und/oder Herkunftsorten. Sie kann vom Benutzer oder von einem Administrator konfiguriert werden. Jede Permission spezifiziert einen Zugriff auf eine bestimmte Ressource, wie z. B. Lesen oder Schreiben einer Datei oder eines Verzeichnisses. Das Laufzeitsystem führt den Code in Schutzbereichen (`Protection Domain`) aus. Dabei handelt es sich um eine individuell abgeschirmte Umgebung. Die Rechte des in einer Protection Domain ausgeführten Codes entsprechen den in der Policy angegebenen Rechten. Eine Domäne kann, falls dies gewünscht wird, wie eine Sandbox konfiguriert werden, so dass ein Applet weiterhin keinen Zugriff auf Systemressourcen hat. In der VM von Java 2

[2]Umgebungsvariable `CLASSPATH` definiert die Verzeichnisse, in denen nach Java-Klassen gesucht wird

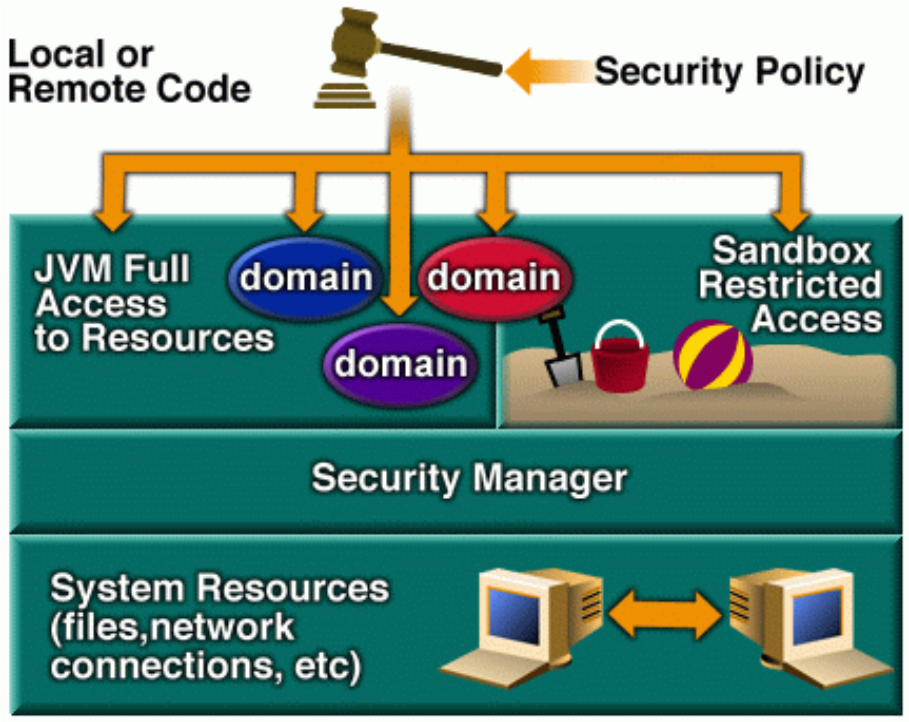

Abbildung 3: Das Sandboxmodell in Java 2

gibt es eine neue Sicherheitskomponente, den `AccessController`. Dieser arbeitet eng mit dem SecurityManager zusammen und stellt sicher, dass die in der Policy angegebenen Rechte eingehalten werden. Der SecurityManager ist aber nach wie vor der Schlüssel zu den Ressourcen, er benutzt lediglich den AccessController zur Durchführung seiner Schutzaufgaben. Der SecurityManager entscheidet jetzt nicht mehr selber, ob ein Zugriff auf eine bestimmte Ressource erlaubt ist, sondern er erfragt dies beim AccessController. Auf diese Art und Weise wird eine Abwärtskompatibilität zu älteren Java-Versionen gewahrt.[SUN02a], [OAK98]

6 Security-API und Kryptographie

Die in den vorangegangenen Kapiteln beschriebenen Sicherheitsmechanismen beziehen sich alle auf ein sehr grundlegendes Niveau. Bei der Entwicklung eigener Java-Programme ist es unter Umständen notwendig, weitere Sicherheitsmechanismen auf höheren Ebenen hinzuzufügen. Das Java-Security-API (*API - Application Programmer Interface* - Programmierschnittstele) stellt dazu einen Satz von Klassen und Schnittstellen zur Verfügung. Bestandteil der Security-API ist die JCA (*Java Cryptography Architecture*). Sie ist im Wesentlichen ein Grundgerüst für die Benutzung und Entwicklung von kryptographischen Verfahren. Außerdem bietet sie eine

Klasse	Aufgabe
`MessageDigest`	Berechnung von Hash-Werten
`Signature`	Erzeugung und Verifikation digitaler Signaturen
`SecureRandom`	Generierung von kryptographischen Zufallszahlen
`KeyPairGenerator`	Generierung von (public- und private-) Schlüsselpaaren
`KeyGenerator`	Generierung von geheimen Schlüsseln
`KeyStore`	Zugriff auf eine Schlüsseldatenbank
`Cipher`	Ver- und Entschlüsselung
`Mac`	Berechnung eines MAC (Method Authentication Code)
`X509Certificate`	Zugriff auf ein X.509-Zertifikat

Tabelle 1: ausgewählte Klassen der JCA und JCE

einheitliche Schnittstelle zu verschiedenen kryptographischen Funktionalitäten, die unabhängig von den verwendeten Algorithmen und der zu Grunde liegenden Implementierung ist. Somit bleibt zum einen eine Unabhängigkeit von der Implementierung gewahrt. Eine (nicht zufriedenstellende) Implementierung kann ohne weitere Probleme durch eine andere ersetzt werden. Ein Paket, das Implementierungen von bestimmten Algorithmen enthält, wird auch als *Cryptography Service Provider* (CSP) bezeichnet. Zum Lieferumfang von Java gehört auch ein CSP `SUN`, der Implementierungen der wichtigsten kryptographischen Verfahren zur Verfügung stellt. Zum anderen bleibt durch die gewählte Architektur eine Unabhängigkeit von dem verwendeten Algorithmus gewahrt. So kann z. B. mit digitalen Signaturen gearbeitet werden, losgelöst vom Algorithmus, der diese tatsächlich berechnet. Außerdem ist JCA so entwickelt worden, dass sie ohne Probleme um weitere Funktionalität ergänzt werden kann. Die JCA API wurde um die *Java Cryptography Extension* (JCE) erweitert. Diese enthält u. a. Implementierung von verschiedenen Verschlüsselungsalgorithmen wie z. B. RSA oder DSA. Die JCE gehören auf Grund der geänderten US-Exportbeschränkungen für starke Kryptographie erst seit dem JDK 1.4 zum Standardlieferumfang von Java. Die zugehörigen Klassen und Schnittstellen befinden sich im Paket `javax.crypto` und `javax.security` und deren Unterpaketen. Eine vertiefende Behandlung dieser Thematik würde an dieser Stelle zu weit führen und den Rahmen dieser Ausarbeitung sprengen. Der Vollständigkeit halber sollen dennoch einige Klassen sowie deren Aufgaben kurz erwähnt werden (vgl. Tabelle 1). Für eine ausführlichere Dokumentation sei hier

auf [SUN02c], [SUN02d] und [SUN02b] verwiesen. Im JDK 1.4 sind weitere Sicherheits-APIs enthalten. Neben JCE sind nun sowohl die *Java Secure Socket Extension* (JSSE), die Java-Implementierungen der SSL- und TLS- Protokolle enthält, als auch der *Java Authentication and Authorization Service* (JAAS), der Authentikations- und Autorisations- Dienste zur Verfügung stellt, fester Bestandteil des JDK. Auch hier sei aus Gründen der Komplexität nur auf [SUN02e] und [SUN01a] verwiesen.

7 Signierte Klassen

Seit Java Version 1.1 können JAR-Archive signiert werden. JAR-Archive können neben den benötigten Java-Klassen noch weitere Dateien wie Bild- und Audiodateien enthalten. Dies verkürzt allgemein die Ladezeit über ein Netzwerk und ist für den Nutzer einfacher zu handhaben als getrennte Dateien.

Die Sprachdefinition wurde dafür um Klassen für Schlüsselverwaltung, Verschlüsselung und digitale Signaturen erweitert. Die genaue Vorstellung der Klassen würde den Rahmen dieser Ausarbeitung spregen. Genaueres ist z.B. in [OAK98] nachzulesen. Sun liefert bei seinem *Java Development Kit* Tools für die Schlüsselverwaltung und JAR-Signierung mit, die auf diesen Klassen basieren.

Zur Signierung wird die von SSL[3] /TLS[4] bekannte ITU-T X.509 Empfehlung benutzt. Dabei signiert eine vertrauenswürdige Institution (CA[5]) das öffentliche Zertifikat des Nutzers, nachdem dieser z.B. durch Vorlage des Personalausweises oder ähnlichen amtlichen Dokumenten identifiziert wurde. Die öffentlichen Schlüssel der verschiedenen CAs müssen natürlich der überprüfenden Software bekannt sein, damit eine Überprüfung der Signatur überhaupt erst möglich wird. Solange die verwendete ROOT-CA der Software bekannt ist, ist das Verfahren für den Nutzer einfach, da er nichts machen muss.

Die Überprüfung der Zertifikate in Java wird von einer Java Klasse, dem *SecurityProvider* vorgenommen. Dies ist in der Voreinstellung die Klasse *sun.security.provider.Sun*. Sie kann vom Benutzer durch einen eigenen SecurityProvider ersetzt werden.

Eine signierte JAR-Datei bedeutet nicht automatisch, dass der Inhalt unproblematisch ist. Es wird nur sichergestellt, dass der Inhalt mittels des verwendeten Zertifikates signiert und nicht von einem dritten manipuliert wurde. Eine Prüfung des Codes seitens der CA oder einer anderen Stelle wird *nicht* vorgenommen!

Es ist auch denkbar, dass ein privater Schlüssel gestohlen wird. Somit kann ein dritter Signaturen mit einer falschen Identität ausstellen. Ein sicherheitsbewusster Nutzer kann die Integrität einer CA anzweifeln. Es ist schon vorgekommen, dass eine CA unbefugten Zertifikate ausgehändigt hat [VER01]. Das JDK 1.3.1 liefert Zertifikate von Thawte und Verisign mit. Inzwischen wurde Thawte von Verisign aufgekauft, so dass - zumindest beim JDK 1.3.1 - keine

[3] SSL: secure socket layer

[4] TLS: transport layer security

[5] CA: certification authority

voneinander unabhängigen CAs verfügbar sind.

Dennoch werden entfernten signierten Klassen im Allgemeinen gleiche Zugriffsrechte gewährt wie lokalen Klassen. Es ist sinnvoll, die Rechte auch für signierte Klassen einzuschränken und nur vertrauenswürden Klassen die von ihnen mehr benötigten Rechte einzuräumen. Bei Java 1.1 war dies aufwendig, da ein eigener *SecurityMananger* programmiert werden musste. Java2 bietet über den *AccessController Security Policies*, die eine feinere Kontrolle der Rechte ermöglichen. Ein eigener *SecurityMananger* wird nur noch in Ausnahmefällen benötigt.

Beispiele für signierte, aber nicht unbedingt vertrauenswürdige Inhalte finden sich in der Welt der ActiveX-Controls zur Genüge [IKS97].

Als weiteres Problem stellen sich die Kosten für ein Zertifikat heraus. Sie betragen knapp 200 US$ pro Jahr und das Zertifikat muss jedes Jahr verlängert werden. Hobbyprogrammierer und kleine Firmen sehen nicht unbedingt ein, diese Summen auszugeben. Auch mag es für größere Firmennetzwerke sinnvoll erscheinen, eine eigene CA aufzusetzen. Damit der Nutzer nicht jedes mal das Ausführen bestätigen muss, empfiehlt es sich, den Schlüssel der CA in der verwendeten Software zu hinterlegen. Denkbar wäre eine firmeneigene JDK-Distribution, soweit dies mit den Lizenzbedingungen von Sun vereinbar ist.

7.1 keytool

Das Keytool ist der Nachfolger des Tools *javakey* der Java-Version 1.1. Es wird mit Java2 mitgeliefert.

Das keytool ermöglicht das Erstellen und Verwalten von Schlüsseln.

Hier ein Beispiel, das ein Zertifikat erstellt.

```
$ keytool -genkey -alias agb
Enter keystore password: geheim
What is your first and last name?
[Unknown]: Andre Grosse Bley
What is the name of your organizational unit?
[Unknown]:
What is the name of your organization?
[Unknown]: Privat
What is the name of your City or Locality?
[Unknown]: Bochum
What is the name of your State or Province?
[Unknown]:
What is the two-letter country code for this unit?
[Unknown]: DE
Is <CN=Andre Grosse Bley, OU=Unknown, O=Privat, L=Bochum,
```

```
ST=Unknown, C=DE> correct? [no]: yes

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

Aus diesem Schlüssel wird nun ein Certificate Request generiert, der an eine CA geschickt werden kann.

```
$ keytool -certreq -alias agb -file CAcertreq.out
Enter keystore password: geheim
$ cat CAcertreq.out
---BEGIN NEW CERTIFICATE REQUEST---
(...)
---END NEW CERTIFICATE REQUEST---
```

Nachdem die CA das Zertifikat signiert zurückgeschickt hat, muss es noch importiert werden.

```
$ keytool -import -file 00.pem
Enter keystore password: geheim
Owner: CN=Andre Grosse Bley, OU=Unknown, O=Privat, L=Bochum,
ST=Unknown, C=DE
Issuer: EmailAddress=grossagk@rub.de, CN=Andre Grosse Bley,
O=Purgatory CA, L=Bochum, ST=Some-State, C=DE
Serial number: 0
Valid from: Thu Feb 06 12:58:41 CET 2003
until: Fri Feb 06 12:58:41 CET 2004
Certificate fingerprints:
MD5: 83:A6:D8:E4:0F:28:6C:0F:22:88:53:97:ED:4A:18:09
SHA1: 93:5C:D0:06:6D:88:2F:F5:47:9E:34:74:C3:82:6F:67:BE:BA:9D:D9
Trust this certificate? [no]: yes
Certificate was added to keystore
```

7.2 jarsigner

Auch Jarsigner ist ebenfalls der Java2-Nachfolger von javakey. Es dient zum signieren und Überprüfen der Signaturen von JAR-Archiven.

Hier ein Beispiel zum Signieren eines JAR-Files, welches TicTacToe.class aus dem JDK enthält.

```
$ jarsigner -signedjar /tmp/ttt_sign.jar /tmp/ttt.jar agb
Enter Passphrase for keystore: geheim
```

jarsigner kann auch zum Überprüfen von Signaturen eingesetzt werden:

```
$ jarsigner -verify -verbose /tmp/ttt_sign.jar

139 Thu Feb 06 13:25:30 CET 2003 META-INF/MANIFEST.MF
192 Thu Feb 06 13:25:34 CET 2003 META-INF/AGB.SF
```

```
1048 Thu Feb 06 13:25:34 CET 2003 META-INF/AGB.DSA
0 Thu Feb 06 13:10:58 CET 2003 META-INF/
smk 3900 Fri Jun 14 17:57:32 CEST 2002 TicTacToe.class

s = signature was verified
m = entry is listed in manifest
k = at least one certificate was found in keystore
i = at least one certificate was found in identity scope

jar verified.
```

7.3 java.security

Die Datei *JAVAROOT/jre/lib/security/java.security* ist das *master security properties file* und legt somit den verwendeten SecurityProvider und PolicyProvider fest. Es ist möglich, mehrere Provider anzugeben.

Auch ist es möglich, den Suchpfad für die Policies einzustellen, um so z.B. dem Nutzer die Möglichkeit der Änderung der Policies zu verweigern. Auch kann das master policy file von einem Netzwerklaufwerk geladen werden, um so netzwerkweit gleiche Policies zu haben.

Als Beispiel sind die wichtigsten Einstellungen aus dem java.security File des JDK 1.3.1 aufgeführt.

```
security.provider.1=sun.security.provider.Sun
security.provider.2=com.sun.rsajca.Provider

policy.provider=sun.security.provider.PolicyFile

policy.url.1=file:$java.home/lib/security/java.policy
policy.url.2=file:$user.home/.java.policy

keystore.type=jks
```

8 Policies

Die Policies sind eine wesentlichte Erweiterung in Java2. Sie ermöglichen eine Feineinstellung der Zugriffsrechte. Sie werden von einer Klasse, *der Policy Class* realisiert. Die Standard-Klasse von Sun, *sun.security.provider.PolicyFile* liest die globalen Berechtigungen aus der Datei *java.policy*. Zusätzlich kann jeder Nutzer des Systems[6] in der Datei *.java.policy* in seinem Homeverzeichnis weitere Policies vorgeben. In diesen Dateien ist es möglich, Netzwerkverbindungen nur zu bestimmten Rechner zuzulassen, Dateioperationen auf bestimmte Pfade beschränken und vieles mehr. Bestimmten Klassen, die durch die Codebase identifiziert werden, können andere Privilegien erhalten. Dabei ist zu beachten, dass die Codebase vom Autor einer Klasse frei gewählt werden kann. Zur Bildung der Codebase gibt es eine Konvention, die die Benutzung der Internet-Domain in umgekehrter Reihenfolge erfordert, z.B. *com.sun.java*.

Mehr Sicherheit verspricht das Einbeziehen eventueller Signaturen. Es ist möglich, Klassen, die von einem bestimmten Schlüssel signiert wurden, andere Rechte zuzuweisen. Natürlich können beide Mechanismen kombiniert werden.

Sun liefert bei seinem JDK ein in Java geschriebenes Tool mit, mit dem die Policies editiert werden können. Vom Prinzip her ist das Editieren mit einem Texteditor ebenfalls möglich.

Die Zahl der Möglichkeiten, Rechte zu vergeben wird mit einer komplexen Konfiguration bezahlt. Der Endnutzer mag in vielen Fällen mit der Konfiguration überfordert sein. Auch das *policytool* erleichtert die Konfiguration nicht sehr viel, hier ist dringend eine überarbeitete Version erforderlich.

Abbildung 4, 5 und 6 zeigen eine Sitzung mit dem Policy Tool, angewandt auf das folgende Policy-File.

```
keystore ".keystore";

grant codeBase "file:/usr/local/share/apps/kjava/-" {
permission java.security.AllPermission;
};

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

grant {
permission java.lang.RuntimePermission "stopThread";
permission java.net.SocketPermission "localhost:1024-", "listen";
```

[6] zumindest bei mehrbenutzerfähigen Betriebssystemen wie z.B. Unix

```
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";
};

grant signedBy "Duke", codeBase "http://java.sun.com/security/
signExample12/signedWriteFile.jar" {
permission java.io.FilePermission "/tmp/foo", "write";
};

grant codeBase "http://192.168.150.101/titan.jar" {
permission java.security.AllPermission;
};

grant codeBase "http://192.168.150.104/titan.jar" {
permission java.security.AllPermission;};
```

Abbildung 4: Startbildschirm des Policytools

9 Sicherheitsprobleme

Auch bei der Veröffentlichung von Sicherheitsproblemen bleibt Sun dem Motto „Sicherheit durch Offenheit" treu. So werden alle existierenden und zusätzlich potientielle Sicherheitslücken veröffentlicht [SUN02].

Bis jetzt ist kein Fehler im Konzept von Java bekannt. Allerdings gibt es Fehler in der Implementiertung der einzelenen Sicherheitsmechanismen. Für all diese Fehler waren Patches schnell verfügbar und wurden von Sun an alle Lizenznehmer weitergeleitet.

So konnten in einigen JVM-Versionen Applets doch die virtuelle Maschine beenden oder mehr Rechte erlangen. Wie schon gesagt handelt es sich dabei nicht um Designfehler, sondern nur um Fehler in der Implementierung. Zum Teil wies auch der *bytecode verifier* Schwachstellen auf, so dass er nicht alle bösartigen Codestücke erkennen konnte. Genaueres über alle Schwachstellen findet sich in [SUN02]. Beispiele für *hostile Applets*, die Schwachstellen ausnutzen, finden sich im WWW (vgl. [MLD02], [GLC02]).

Unter Laborbedingungen sind zwei Java-Viren bekannt, die aber nicht aus der Sandbox-Umgebung her tätig werden können, solange der Nutzer ihnen nicht manuell mehr Rechte gewährt

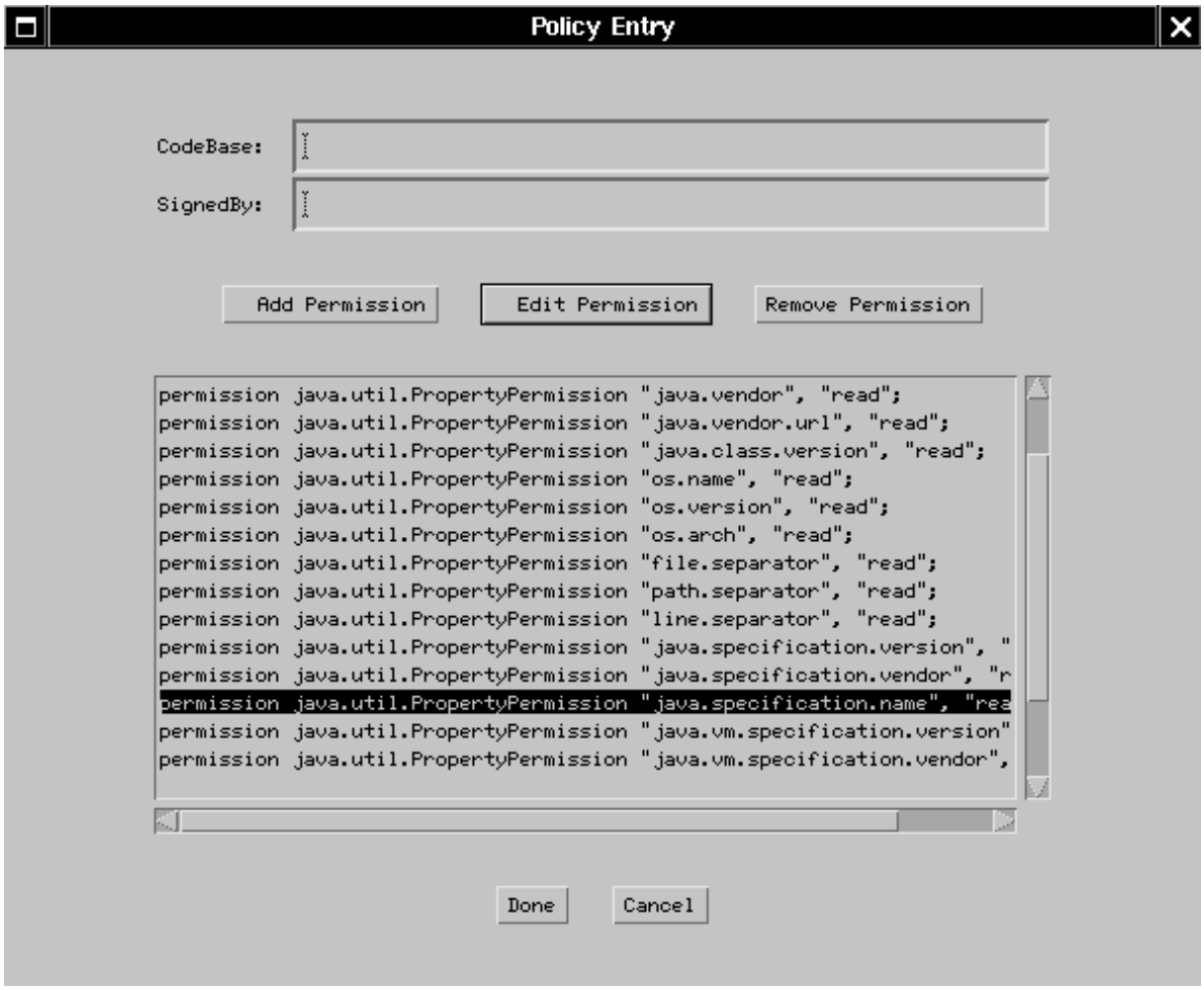

Abbildung 5: Auswahl der Codebase *ALL*

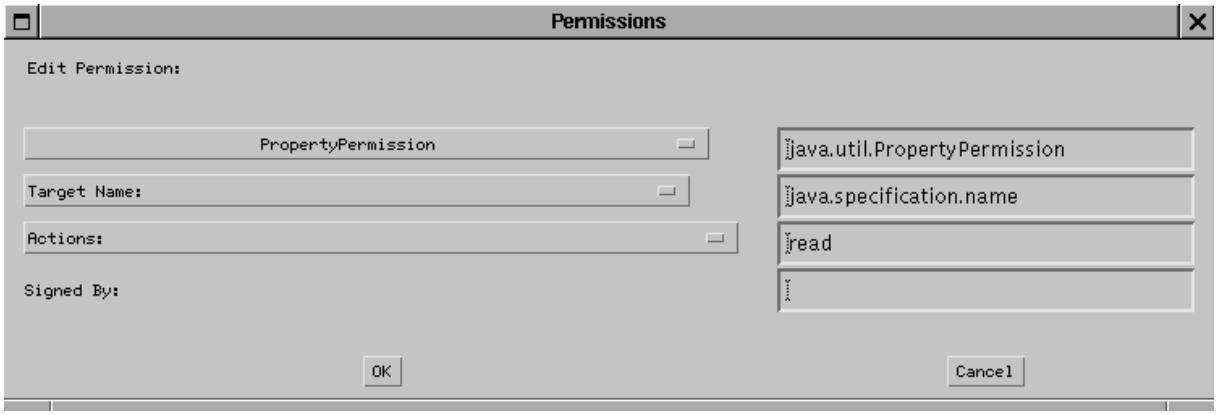

Abbildung 6: Ändern einer Berechtigung

(vgl. [SOP98] und [SYM99]). Beide Viren gelten als Demonstrationsviren, sie haben keine nennenswerte Verbreitung gefunden (Symantec gibt weniger als 49 Infektionen an).

Wie bei jeder Software kann nur empfohlen werden, die Sicherheitshinweise des Herstellers ernstzunehmen und gegebenenfalls die entsprechenden Patches einzuspielen. Dies alles schützt nicht vor unbekannten Sicherheitslücken, die aber bei Java aufgrund des Konzeptes und der Sicherheitsphilosophie von Sun nicht so häufig auftreten wie in anderer Software.

Sun stellt den gesamten Quellcode von Java inklusive virtueller Maschine zur Verfügung. Dies ermöglicht eine eigene Sicherheitsprüfung, die aber zeit- und kostenintensiv ist.

Vom Prinzip her können Applets, wie jedes andere Programm auch, durch Belegung von Resourcen die Leistung des Systems einschränken (Denial of Service).

Das Konzept von Java verhindert zuverlässig *Buffer Overflows*, die heute für die meisten Sicherheitslücken in Programmen verantwortlich sind.

10 Anwendungen für Java

10.1 Applets

Java Applets sind den meisten Nutzers des WWW bekannt. Das sind Programme, die auf dem Rechner des Nutzers ablaufen und dort z.B. grafische Effekte abspielen, für die Kommunikation mit dem Homebanking-Server sorgen oder einen Börsenticker darstellen. Durch Weiterentwicklung von HTML nimmt die Verbreitung von Applets für aktive Inhalte allerdings ab.
Da die Applets über das Netz geladen werden und ohne Bestätigung des Nutzers auf dem Rechner des Nutzers, der vertrauliche Daten enthalten kann, ausgeführt werden, ist es sehr wichtig, dass ein Applet nicht uneingeschränkt auf die Resourcen zugreifen kann.

10.2 Applikationen

Anwendungen, die in Java geschrieben wurden haben den Vorteil, ohne Neuübersetzung auf vielen verschiedenen Plattformen abzulaufen.
Standardmässig besitzen *Applikationen* volle Zugriffsrechte auf den lokalen Rechner. Diese können jedoch eingeschränkt werden, in dem ein *SecurityMananger* beim Aufruf der JVM per Kommandozeile installiert wird (z.B. durch `java -Djava.security.mananger -Djava.security.policy=policyfile PROGRAMM`). Dies ist bei Applikationen aus nicht vertrauenswürdiger Quelle ratsam.
Damit ist Java in diesem Punkt vielen anderen Systemen überlegen, da normalerweise ein Programm immer die Rechte des Nutzers erbt, der es startet. Eine Einschränkung der Zugriffsrechte ist meist nur durch zusätzliche Software möglich, die sehr tief in das Betriebssystem eingreift und unter Umständen nicht alle Zugriffe abfangen kann.

10.3 Servlets

Aufgrund des Sicherheitskonzeptes gewinnen Java Servlets immer mehr Beliebtheit im Bereich der aktiven Inhalte von WWW-Servern (Java Server Pages, .jsp). Im Gegensatz zu Microsofts' ASP (Active Server Pages) legt sich der Entwickler nicht auf ein Betriebssystem fest, sondern kann dies bei wachsenden Anforderungen ersetzen, ohne Änderungen an den Seiten und Programmen vornehmen zu müssen. Desweiteren bietet Java eine flexible Anbindung an verschiedene Datenbanken. Die Sprachsicherheit von Java, vor allem die strenge Typprüfung, hilft

hierbei sichere Anwendungen zu programmieren.

Interessanterweise sah Sun diese Anwendung für Java am Anfang überhaupt nicht.

11 Ausblick

Das Java-Konzept hat sich in der Praxis bewährt, auch wenn die Implementierungen zum Teil Sicherheitsprobleme aufweisen. Immerhin kann Java-Bytecode nur in einer virtuellen Maschine Schaden anrichten, der durch eine geschickte Konfiguration begrenzt werden kann.
Microsoft ist Java ein Dorn im Auge, da sie C#, .net und ActiveX als Plattform für aktive Inhalte genutzt sehen möchte. Dabei ist es mit der Plattformunabhängigkeit vorbei, da die ActiveX Controls für die entsprechende CPU des Zielsystemes kompiliert werden müssen. Beim Entwurf wurde der Sicherheitsaspekt aussen vor gelassen, so dass ein ActiveX-Control vollen Zugriff auf den Rechner des Nutzers, der es wohlmöglich unbewusst ausführt, hat. Nach grosser Kritik wurden Sicherheitsaspekte angeflickt, die aber sehr viele Lücken offen lassen.
Java setzt auf Kompatibilität. Selbst alte Klassen aus den Anfangszeiten von Java sind auf den heute aktuellen JVMs ablauffähig. Auf der anderen Seite sorgt diese Kompatibilität für gewisse Redundanzen, wie sie z.B. bei den Klassen *Security Mananger* und *AccessController* auftritt.
Ein neuer Markt für Java-Bytecode sind portable Geräte, wie z.B. PDAs und neuere Mobiltelefone. Für sie stellt Sun extra eine abgespeckte Version von Java (*Java2 Micro Edition, J2ME*) bereit, da das Standard-Java (*Java2 Standard Edition, J2SE*) inzwischen zu groß für Geräte mit kleinem Speicher ist. Ein mobiles Gerät benötigt z.B. kaum sämtliche fähigkeiten der Windows GUI. Durch die Vielfalt der Geräte gewinnt die Plattformunabhängigkeit von Java stark an Gewicht bei der Auswahl der Programmiersprache.
Für größere Anwendungen auf PC-Basis hat sich Java als Programmiersprache ebenfalls etabliert. Meistens werden aber native Compiler zur Erzeugung des Programmcodes benutzt. Ein native Compiler erzeugt keinen Bytecode für die JVM, sondern direkt Maschinencode für die verwendete Prozessorarchitektur. Natürlich gibt es dann keine Sicherheitsmechanismen während der Laufzeit mehr.
Bei der Vorstellung von Java war oft die Rede davon, dass kleine, laufwerkslose Javastations den PC ersetzen sollten, da sie zu der Zeit bedeutend billiger wären als komplette PCs. Die Daten sollten dabei zentral in einem Netzwerk gesichert werden. Wie jedermann heute weiss, hat sich diese Idee nicht durchsetzen können, obwohl sie sicher diverse Sicherheitsprobleme und Viren hätte vermeiden können.
Abschliessend ist zu sagen, dass trotz bekannter Sicherheitslücken wenig böswilliger Bytecode bekannt ist, der einen realen Schaden verursacht. Daher ist es anzunehmen, dass Java heute die sicherste verfügbare Plattform, vor allem im Vergleich zu Systemen aus Redmond, ist [FLA99].

Literatur

[CT00] Karsten Sohr: *Sandkastenspiele : Java-Sicherheitsmodelle, ct Ausgabe 11/00*

[FLA99] David Flanagan: *Java in a Nutshell 3rd Edition*, O'Reilly & Associates 1999, ISBN 1-56592-487-8

[GLC02] Global Conspiracy: *HOSTILE JAVA APPLETS*, http://www.geocities.com/Hollywoord/2900/hostile.html

[IKS97] IKS-Jena: *ActiveX - Konzeptionelle Sicherheitslücke*, http://www.iks-jena.de/mitarb/lutz/security/activex.en.html

[KR02] Guido Krüger: *Handbuch der Java-Programmierung*, Addison Wesley 2002, ISBN 3-8273-1949-8

[MLD02] Mark LaDue: *A Collection of Increasingly Hostile Applets*, http://www.cigital.com/hostile-applets/

[OAK98] Scottt Oaks: *Java Security*, O'Reilly & Associates 1998, ISBN 1-56592-403-7

[SOP98] Sophos: *The first Java virus*, http://www.sophos.com/virusinfo/articles/java.html

[SUN01a] Sun Microsystems: *JAAS Reference Guide*, http://java.sun.com/j2se/1.4/docs/guide/security/jaas/JAASRefGuide.html

[SUN02] Sun Microsystems: *Chronology of security-related bugs and issues, 11/19/02*, http://java.sun.com/sfaq/chronology.html

[SUN02a] Sun Microsystems: *The Java Tutorial, 11/25/02*, http://java.sun.com/doc/books/tutorial/

[SUN02b] Sun Microsystems: *Java 2 Platform, Std. Ed. v 1.4.1 API-Specification*, http://java.sun.com/j2se/doc/api

[SUN02c] Sun Microsystems: *JCA API Specification Reference, 02/08/02* , http://java.sun.com/j2se/1.4/docs/guide/security/CryptoSpec.html

[SUN02d] Sun Microsystems: *JCE Reference Guide , 01/15/02* , http://java.sun.com/j2se/1.4/docs/guide/security/jce/JCERefGuide.html

[SUN02e] Sun Microsystems: *JSSE Reference Guide , 01/15/02* , http://java.sun.com/j2se/1.4/docs/guide/security/jsse/JSSERefGuide.html

[SYM99] symantec: *JavaApp.BeanHive*, http://www.symantec.com/avcenter/venc/data/javaapp.beanhive.html

[VER01] VeriSign: *VeriSign Security Alert Fraud Detected in Authenticode Code Signing Certificates March 22, 2001*, http://www.verisign.com/developer/notice/authenticode/index.html

Alle Links im WWW sind Stand Januar 2003.