Die Sicherheitsaspekte von Java

Joachim Posegga
Deutsche Telekom AG
Technologiezentrum Darmstadt,
Forschungsbereich IT Sicherheit/FE34a
D-64276 Darmstadt
posegga@tzd.telekom.de

Zusammenfassung

Java stellt, insbesondere hinsichtlich des Applet-Konzepts, eine äußerst interessante Technologie dar, die darauf beruht, Programme über ein Netz zu transportieren und auszuführen. Aus der Sicht der IT–Sicherheit ist ein solches Vorgehen jedoch nicht unproblematisch.

Im folgenden wird diese Problematik näher beleuchtet, das Java zugrundeliegende Konzept erläutert und eingeordnet. Das Ziel ist, den Leser in die Lage zu versetzen, die tatsächlich mit dem Einsatz von Java verbundenen Sicherheitsrisiken bei Anwendungen einschätzen zu können.

Abstract

Java is a very promising emerging technology, in particular because of the concept of downloadable code behind Java applets. The underlying idea is to provide smallish applications (applets) in a network, which can be downloaded and executed as needed. Unfortunately, such a procedure is quite problematic from the perspective of IT security.

This paper investigates the problems involved with using such downloadable code, and reviews the solutions proposed in the Java runtime environment. The aim of the paper is to support the reader in understanding the technology and help him/her to asses the actual risks of Java applications.

CR Classification: **C.2.1, C.2.4, D.2.0, D.2.4, D.3.4, D.4.6, F.1.1,F.3.1**

1 Einleitung

Selten stieß die Einführung einer neuen Technologie im Internet auf mehr Echo als bei dem von Sun Microsystems eingeführten Konzept von Java: Mitte 1995 wurde die Newsgruppe `comp.lang.java` ins Leben gerufen und bereits ein halbes Jahr später erschienen darin 300-400 neue Nachrichten pro Tag –weit jenseits dessen, was überschaubar ist. Mitte 1996 konnte Java als etabliert betrachtet werden –interessanterweise bevor es tatsächlich „ernsthafte“, geschweige denn kommerzielle Java-Anwendungen gab.

Selbst Microsoft wurde von der Java-Welle regelrecht überrollt und sah 1996 seine eigene, damals gültige Internet-Strategie (im wesentlichen bestehend aus Blackbird, Visual Basic und OLE/COM) vor dem Scheitern. Wer jedoch den Wintel-Standard bereits fallen sah, hatte die Flexibilität von Microsoft unterschätzt: Mit geradezu unglaublicher Geschwindigkeit sprang man auf den Internet-Zug auf, und versucht nun, in Form von ActiveX mit einverleibtem Java, die eigenen Vorstellungen als de-facto Standard zu etablieren.

Bei allen rosigen Zukunftsaussichten der neuen Technologie sollte jedoch ein wichtiger Aspekt nicht übersehen werden: Die fortschreitende Kommerzialisierung des Internet bedingt, daß die dort eingesetzten Technologien auch neuen Sicherheitskriterien genügen müssen: Solange das Internet im wesentlichen eine akademische Angelegenheit war, erschienen die immer wieder auftauchenden Sicherheitslücken zwar ärgerlich, aber im wesentlichen doch akzeptabel und kontrollierbar; die Expertise aller Beteiligten (incl. der Endbenutzer) war hoch, und die eventuellen Konsequenzen und Gefahren des eigenen Agierens im Netz allen Beteiligten mehr oder weniger bewußt.

Mit der fortschreitenden privaten Nutzung des Internets sind obige Voraussetzung nicht mehr erfüllt, da nun auch technisch weniger bedarfte Nutzer zur anvisierten Zielgruppe gehören. Weiterhin setzt der Umschlag von „echtem Geld“, wie er im Rahmen des Electronic Commerce vorgesehen ist, neue Maßstäbe, da der Anreiz zu Mißbrauch durch die Präsenz von Zahlungsmitteln erheblich erhöht wird. Gerade in der letzten Zeit mehren sich die Stimmen, die auf Mängel und ungelöste Probleme im Sicherheitsbereich hinweisen, insbesondere bei Technologien wie ActiveX.

Java wird hier oft als sicherere Alternative angesehen, der technische Hintergrund wird jedoch nur selten im Detail beleuchten. Genau dies soll im Folgenden versucht werden, mit dem Ziel, dem Leser zu helfen, die Technologie hinter Java einordnen, und deren Risiken abschätzen zu können.

2 Das Konzept von Java–Applets

Auf die Frage *Was ist Java?* gibt es viele Antworten. Zunächst kann Java als eine Programmiersprache angesehen werden, in der u.a. sogenannte Applets programmiert werden können, dies sind kleine Programme die über ein Netz geladen und dann ausgeführt werden. Wichtiger als die Programmiersprache selbst ist für unsere Betrachtungen das Konzept solcher Java-Applets:

Aus „historischer“ Sicht kann man die Idee dahinter als Erweiterung der Funktionalität des World Wide Web (WWW) verstehen: die Intention hinter dem WWW war es, die Möglichkeit zu schaffen, von einem ins Internet eingebundenen Rechner aus Dokumente (Text und Graphik) zu betrachten, die sich auf einem anderen Rechner befinden. Technisch ist dies realisiert, indem über entsprechende Protokolle (HTTP) das Layout und der Inhalt eines Dokumentes (beschrieben in HTML) übertragen wird und diese beim Nutzer mittels eines Browsers dargestellt wird. Zusätzliche Hyperlinks verknüpfen beliebige Dokumente miteinander. Im Prinzip handelt es sich also um die Vermittlung von statischer Information, mit der Möglichkeit sehr einfach darin zu navigieren.

Java trat nun an, um nicht nur Information, sondern auch Funktionalität zu übertragen, d.h., eine im wesentlichen statische Angelegenheit um Dynamik zu erweitern. Durch Java werden die in

HTML formulierten WWW-Seiten um ausführbare Programme ergänzt, und der WWW-Browser wird zur virtuellen Maschine dieser Programme. Somit kann man nicht nur Daten, sondern lauffähige Programme, also im Prinzip jede beliebige Funktionalität, übertragen.

2.1 Technische Realisierung

Eine solche Übermittlung von Funktionalität kann mit Java-Applets recht einfach realisiert werden. Java selbst ist zunächst eine objektorientierte, an C++ angelehnte Programmiersprache [Gosling & Steele 1996], die in sog. *Java Bytecode* übersetzt werden kann; wird ein solcher Bytecode über ein Netz geladen und ausgeführt, dann bezeichnet man dieses Programm als *Applet*. Ein solches Applet benötigt eine *Java Virtual Machine (Java VM)*, also eine spezielle Laufzeitumgebung, um ablaufen zu können. Zur Laufzeit wird der Bytecode dann interpretiert.

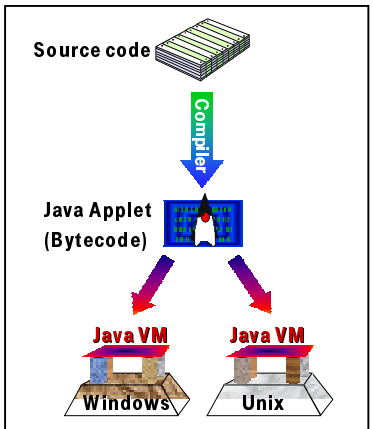


Abbildung 1: Java Platform

Hier liegt auch der Schlüssel zu der propagierten plattformunabhängigen Entwicklung von Java (vgl. **Fehler! Verweisquelle konnte nicht gefunden werden.**): der Bytecode kann auf jedem System ablaufen, das eine Java Virtual Machine zur Verfügung stellt, d.h. lediglich der Bytecode-Interpreter und die Funktionen, die dieser benutzt, müssen auf der jeweiligen Plattform implementiert sein[1].

Beispiele für solche Laufzeitumgebungen sind die gängigen WWW-Browser wie *Netscape*, der *Microsoft Internet Explorer*, oder auch *HotJava* von Sun. Neuere Betriebssysteme unterstützen Java direkt, d.h. die Verwendung eines besonderen Browsers ist nicht mehr notwendig.

Java bietet aus Sicht der Informatik keine prinzipiell neue Funktionalität, auch die Plattformunabhängkeit mittels eines interpretierten Bytecodes ist nicht neu. Auch die Möglichkeit, Programme (Applets) über ein Netz zu übertragen, ist technisch längst etabliert und eröffnet keine prinzipiell neuen Möglichkeiten. Denkt man sich die Idee hinter Java-Applets jedoch einmal konsequent verwirklicht, wird allerdings klar, daß das konkrete Arbeiten mit einem Java-Computer anders aussieht, als man dies bisher gewohnt ist, und hier liegt auch das Potential des Ansatzes:

Das Java-Szenario ist eine konsequente Realisierung eines verteilten, plattformunabhängigen, objektorientierten Systems: Der eigentliche Rechner, der benutzt wird, verliert an Bedeutung, in den Mittelpunkt rückt statt dessen das Netz. Anwendungssoftware ist keine statische Angelegenheit mehr, die (aufwendig) installiert und genutzt wird, sondern besteht in der im Netz zur Verfügung gestellten Funktionalität; bei Bedarf werden die gewünschten Programme einfach über das Netz transportiert, geladen und benutzt.

Damit diese Vorgehensweise funktioniert, sollten die Anwendungsprogramme selbst natürlich von anderer Qualität als bisher sein: große, monolithische Systeme passen nicht mehr ins Bild: Applets sind eher kleine Programmeinheiten, die wenig Funktionalität bieten; die Gesamtfunktionalität wird aus kleinen Teilen zusammengesetzt[2] —aus der Sicht der Informatik eine sinnvolle und lange überfällige Entwicklung.

[1] Alternativ dazu können auch sog. Just-in-time-Compiler eingesetzt werden, die den Java Bytecode quasi in Echtzeit während des Ladens eines Applets in plattformspezifischen Maschinencode übersetzen. Auf diesen Aspekt soll jedoch hier nicht näher eingegangen werden, da er aus sicherheitstechnischer Sicht weniger relevant ist.

[2] Im Prinzip ist hier eine Komponentenarchitektur wie JavaBeans gemeint auf diesen Aspekt soll aber nicht näher eingegangen werden.

Exkurs 1: Kann Java die Welt der Informationstechnik verändern?

Java hat einige interessante „industriepolitische“ Aspekte, hinterfragt es doch den bisherigen “Wintel’’-Standard im PC-Bereich: Java-Applets sind plattformunabhängig, somit nicht mehr an eine konkrete Architektur oder ein bestimmtes Betriebssystem gebunden. Microsofts Monopolstellung in diesem Bereich, basierend auf proprietären Standards, ist dadurch ernsthaft bedroht.

Weiterhin kämen Rechner selbst mit weniger Funktionalität aus: große Massenspeicher für Anwendungssoftware wären z.B. nicht mehr unbedingt notwendig. Somit könnten auch die Systeme der Endbenutzer anders aussehen: Außer einer virtuellen Java-Maschine und der Netzeinbindung würde lediglich ein –wie auch immer gearteter– Bildschirm und eine (ev. reduzierte) Tastatur benötigt. Insbesondere die geplanten Java-Umgebungen in Form von Chips oder Smartcards könnte vollkommen neue Produkte ermöglichen. Ohne weiter auf diesen Aspekt einzugehen, sei hier das Schlagwort „ubiquitious computing“ genannt.

Letztlich paßt das bisherige Konzept zur Lizensierung von Software nicht mehr in das Bild: man könnte statt eines Pauschalbetrages für Software (wie dies bisher der Fall ist) über die tatsächliche Benutzung abrechnen.

Da dies recht spekulativ klingen mag, sei an dieser Stelle auf die Parallelen zum Aufkommen des WWW gegen Ende der 80er Jahre verwiesen: die in Mosaic/HTTP implementierte Technik war ebenfalls weitgehend bekannt, grundlegend neue Konzepte waren nicht zu erkennen; die bekannten Ideen wurden “lediglich’’ konsequent realisiert. Viele der damit konfrontierten Fachleute waren der Ansicht, daß es sich im wesentlichen um eine am Spieltrieb bestimmter Internet-Enthusiasten orientierte Entwicklung handelt, die niemanden “ernsthaft’’ interessiert. Diese Einschätzung beruhte auf der irrigen Annahme, daß die Technik als Ganzes durch Betrachtung der Summe der Einzeltechniken abschätzbar ist.

Anfang der 90er Jahre war WWW aus weiten Bereichen des Informatik-Wissenschaftsbetriebes nicht mehr wegzudenken: wer keine Homepage hat, existiert in der Wissenschaft kaum noch. Nun dringt die Entwicklung mit Vehemenz in den kommerziellen und privaten Bereich vor.

Java zielt in eine ähnliche Richtung, bietet jedoch deutlich mehr Potential, somit auch wesentlich mehr Anwendungsmöglichkeiten. Insofern sind die möglichen Auswirkungen von Java tatsächlich enorm, und die von Sun propagierte „Revolution“ könnte tatsächlich stattfinden

3 Sicherheitsaspekte von Java

Java bietet neue Möglichkeiten, ist aber auch mit einer inhärenten Sicherheitsproblematik behaftet: das Prinzip, ausführbaren Code über ein Netz zu transportieren, ist aus sicherheitstechnischer Perspektive hoch problematisch. Nicht umsonst gilt in den meisten Firmennetzten die Regel, daß Software nur dann benutzt werden darf, wenn sie durch offizielle Kanäle (etwa den Einkauf) gekommen ist. Die Nutzung von Java-Applets aus dem Internet ist damit erst einmal nicht kompatibel.

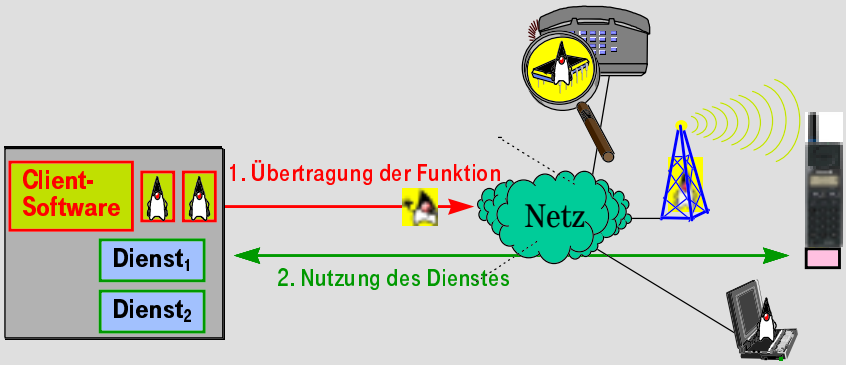
Die damit in Zusammenhang stehende Sicherheitsproblematik muß analysiert und verstanden werden, bevor an “ernsthafte’’ Anwendungen von Java zu denken ist. Dies gilt insbesondere für einen professionellen Einsatz, bei dem besondere Anforderungen an Software gestellt werden (sollten). Aber auch private Nutzer sollten sich bewußt sein, welche Risiken damit verbunden sein können, Java-fähige Browser (wie etwa Netscape oder MS Internet Explorer) zu verwenden.

Exkurs 2: Das Potential von Java am Beispiel der Telekommunikation

Das Telefonnetz wurde originär als ein Netz konzipiert um Sprache (später auch Daten) zu Endgeräten zu übertragen. Ziel war es, knappe Ressourcen effizient zu nutzen, dabei spielten Endgeräte nur eine untergeordnete Rolle, die „Intelligenz“ steckte im Netz (Stichwort: „IN“, Intelligent Network).

Diese Situation wandelt sich jedoch seit einiger Zeit: die Endgeräte übernehmen immer mehr Funktionen und das Netz selbst wird mehr und mehr zur reinen Übertragungsplattform. Eines der wesentlichen Problem dabei ist die starre Funktionalität der TK-Endgeräte. Diese sind eine deutliche Innovationsbremse, denn jeder neue Dienst, der es erforderte, die benutzten Endgeräte anzupassen, steht –aufgrund des dafür notwendigen Aufwandes– vor kaum unüberwindlichen Hürden.

Java bietet nun genau an dieser Stelle eine Lösung an: kann man über ein Netz nicht nur Information, sondern auch Funktionalität zu den Endgeräten übertragen, ist es möglich, ein Endgerät vor der Nutzung eines Dienstes genau an diesen Dienst anzupassen, indem es mit der entsprechenden Software versehen wird. Etwas „informatischer“ ausgedrückt: Java erlaubt es, die Client-Software aus dem Netz auf den Client zu bringen und somit dessen Funktionalität an den Dienst anzupassen; die folgende Abbildung verdeutlicht das:



Mehrere Hersteller von TK-Endgeräten haben für 1998 Java-fähige Modelle angekündigt. Die sich damit bietenden Möglichkeiten sind, gerade für Betreiber großer TK-Netze, enorm. Wäre diese Technologie in der Fläche verfügbar, hätten TK-Netze eine Größenordnung an Flexibilität gewonnen und man könnte von einer geradezu revolutionären Umgestaltung der zugrundeliegenden Technik sprechen.

Diese Aussage gilt natürlich analog auch für „klassische“ Rechnernetze, der eigentlichen „Heimat“ von Java. Der tatsächliche Gewinn an Flexibilität ist aber vergleichsweise geringer: Installation oder Updates von Soft- oder Hardware ist zwar schwierig, aber doch (kostspielige) Routine. Ein TK-Netz dagegen ist in dieser Hinsicht wesentlich inflexibler: selbst ein unverändertes Telefon aus den 50er Jahren funktioniert heute noch. Für Rechnernetze ist das automatische Verteilen von Software also vergleichsweise unspektakulär, der interessantere Aspekt hier ist die Plattformunabhängigkeit von Java .

3.1 Die Grundproblematik

Java-Technolgie besteht –vereinfacht ausgedrückt– darin, nicht nur Information, sondern auch Funktionalität über ein Netz zu transportieren, wobei Ursprung und tatsächlicher Umfang der Funktionalität zunächst unklar sind. Hierin liegt das wesentliche Problem aus der Perspektive der Systemsicherheit.

Aus der Nutzerperspektive formuliert heißt das: Java automatisiert den Prozeß der Übertragung, Installation und Ausführung von Software aus einem Netz. Mehr noch: Java erhebt dies zum grundlegenden Arbeitsprinzip.

Exkurs 3: Computerviren

Wieviel Schaden ein Virus anrichten kann, hängt zum großen Teil von dem Übertragungsweg, und der dadurch bestimmten Geschwindigkeit, mit der er sich verbreitet ab. „Klassische" Viren verbreiten sich über neu installierte Software, wodurch es Monate oder Jahre dauert, bis ein solcher Virus eine signifikante Verbreitung findet. Dadurch läßt sich das Problem relativ gut in den Griff bekommen, wenn entsprechende Virenscanner eingesetzt werden. Deren Schutzwirkung beruht darauf, daß sie neue Viren, bzw. Virustypen erkennen können, bevor diese eine signifikante Verbreitung finden.

Finden Viren aber einen anderen, schnelleren Verbreitungsweg, können diese wesentlich mehr Schaden anrichten, bevor sie erkannt und bekämpft werden können. Ein Beispiel dafür sind die 1996 aufgekommenen WinWord Macro-Viren [CIAG 1996], die sich über Makro-Programme in Dokumenten verbreiten: Da der Austausch von Dokumenten sehr viel schneller stattfindet, wurden Macro-Viren sehr schnell verbreitet und brachten enorme Probleme für größere Unternehmen mit sich.

Daß dies ein Problem ist, läßt sich leicht verstehen, wenn man z.B. Computerviren betrachtet, deren Gefährlichkeit wesentlich durch die Ausbreitungsgeschwindigkeit definiert wird: je schneller sich Viren verbreiten, desto geringer sind die Chancen, diese in den Griff zu bekommen (siehe auch Exkurs 3).

Im Java-Szenario werden solchen Viren praktisch ideale Bedingungen geboten: Das Laden und Ausführen von Applets (also lauffähigen Programmen) von externen Rechnern ist das zugrundeliegende Arbeitsprinzip und der Normalfall. Gäbe es tatsächlich Java-Viren, so könnten diese sich extrem schnell ausbreiten. Im Prinzip ist eine signifikante Verbreitung über das Internet sogar innerhalb weniger Stunden oder Tage möglich –eine bisher nicht gekannte Dimension des Problems. Weiterhin steht, durch die notwendige Einbindung einer Java-Umgebung in ein Netz, auch wesentlich mehr Funktionalität, die mißbraucht werden kann, zur Verfügung als auf einem klassischen PC: ein Java-Interpreter kann nicht nur, sondern er soll in der Regel auch aktiv Netzverbindungen aufbauen und damit arbeiten.

Selbstverständlich sind neben Viren auch alle anderen Formen unerwünschter Funktionalität denkbar (trojanische Pferde, Würmer, etc.). Mögliche "Nebenwirkungen'' von Programmen wären beispielsweise *Integritätsangriffe*, bei denen Daten oder Prozesse geändert oder zerstört werden, *Belegung von Ressourcen*, die eine normale Weiterverwendung eines Rechners erschweren oder unmöglich machen (z.B. das Öffnen tausender Fenster, oder das Erzeugen vieler Prozesse). Auch das *Ausspähen* von Daten stellt ein sehr ernstes Problem dar; dies kann vom Weiterleiten der Password-Datei, bis zum Suchen nach Kreditkarteninformationen im Dateisystem reichen. Auch sonstige *nicht autorisierte Aktionen*, wie etwa das heimliche Verschicken von Email im Namen des Benutzers, können sehr unangenehme Folgen haben.

Prinzipiell können Gegenmaßnahmen zunächst in zwei Hauptbereiche unterteilt werden:

1. *Präventive Maßnahmen*, wie die Sicherstellung der Integrität der benutzten Java-Laufzeitumgebung. Dies ist relativ einfach zu erreichen, indem der Interpreter von einer sicheren Quelle beschafft wird; weiterhin sollte auch sichergestellt sein, daß der Interpreter nicht im nachhinein manipuliert werden kann, dies ist jedoch auf den gängigen Windows-Plattformen nur schwer erreichbar (siehe auch Exkurs 5).

2. *Maßnahmen während des Betriebs*, die verhindern, daß Java-Applets unerwünschte Funktionen ausüben. Dieser Aspekt bringt erheblich mehr Probleme mit sich, denn i.a. läßt sich nicht charakterisieren, was eine solche "unerwünschte Funktionalität'' ist. Erschwerend kommt hinzu, daß Java-Applets oft auch ohne Zutun des Benutzers (oft sogar ohne daß dies auch nur bemerkt wird) geladen und auf dem lokalen Rechner ausgeführt werden. Hier ist ein tragfähiges Konzept zur Verhinderung von Mißbrauch, sowie eine praxistaugliche Realisierung davon, absolut unverzichtbar.

Sun ist diese inhärente Sicherheitsproblematik von Java-Applets natürlich nicht verborgen geblieben [Sun Microsystems 1995, Java Security FAQ 1997]; im Folgenden wird der von Sun vorgeschlagene Ansatz, dem Problem zu begegnen, näher erläutert.

3.2 Das Sicherheitsmodell von Java

Das Sicherheitskonzept bei Java kann zunächst in zwei Bereiche unterschieden werden, das *Sprachdesign* von Java, bzw. des Java-Bytecodes, sowie die *Sicherheitsarchitektur* der Laufzeitumgebung.

3.2.1 Das Sprachdesign

Die wesentliche Komponente in Bezug auf Sicherheit beim Sprachdesign von Java ist die Typsicherheit [Lindholm & Yellin 1996, Dean 1997]; dies soll garantieren, daß Java-Programme nicht auf unautorisierte Bereiche im Speicher zugreifen können, indem verhindert wird, daß der Inhalt von Variablen nicht ihrer Deklaration entspricht und Funktionen nicht mit inkorrekten Argumenten aufgerufen werden. Im Gegensatz zu C++ ist es bei Java nicht möglich, direkt auf Speicherstellen zuzugreifen; die Speicherverwaltung ist dem Java-Laufzeitsystem überlassen, das u.a. einen Garbage-Collector enthält.

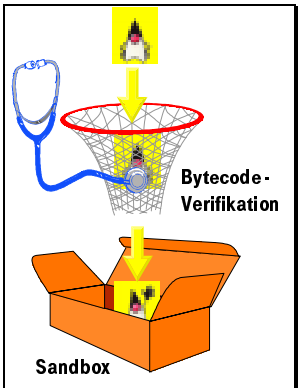


Abbildung 2: Bytecode-Verifikation

Dadurch wird Java zu einer wesentlich sichereren Sprache als beispielsweise C++, in dem Sinne, daß Programme zur Laufzeit nicht in undefinierte Zustände geraten können indem etwa auf nicht allokierten Speicher zugegriffen wird. Diese Vorkehrungen allein garantieren zwar noch kein „gutmütiges“ Verhalten eines Programmes, sind aber eine Voraussetzung, um Programme „unter Kontrolle“ halten zu können. Dies soll durch die nachfolgend beschriebene Sicherheitsarchitektur, die im wesentlichen auf einer sog. „Sandbox“ beruht, erreicht werden. Vereinfacht gesagt ist eine Sandbox ein abgeschotteter Bereich auf einem Rechner, der keinen Zugriff auf „kritische“ Ressourcen zulaßt.

3.2.2 Die Sicherheitsarchitektur

Die Aufgabe der Sicherheitsarchitektur des Laufzeitsystems von Java ist es, zu verhindern, daß aus einem Netz heraus geladene, potentiell unsichere Java-Applets in der lokalen Umgebung Schaden anrichten können. Die Vorgehensweise läßt sich dabei strukturell in drei Phasen einteilen: Die Phase der *Übertragung* eines Applets, die *Lade-* und die *Ausführungsphase*. Diese Phasen sind allerdings nicht strikt trennbar, da Java dynamisches Nachladen von Klassen erlaubt.

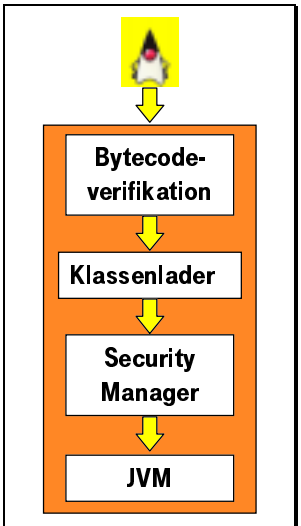


Abbildung 3: Sicherheitsarchitektur

Die wichtigsten funktionalen Komponenten der Sicherheitsarchitektur sind in Abbildung 3 dargestellt: Ankommender Java-Bytecode durchläuft zunächst den *Bytecode-Verifier*, der Applets auf „Gutmütigkeit“ untersucht. Die nächste Komponente ist der *Klassenlader*, der das Laden von Java-Klassen kontrolliert und Konsistenz sicherstellt.[3]

[3] De facto stößt der Klassenlader auch die Bytecode-Verifikation an, dies ist jedoch für unsere Betrachtung sekundär.

Geladene Applets laufen dann in einer sogenannten „*Sandbox*“, einer in der Funktionalität beschränkten Laufzeitumgebung ab. Innerhalb der Sandbox kommt dem *Security Manager* („Sicherheitsmanager“) eine zentrale Rolle zu, da er alle sicherheitsrelevanten Aktionen eines Applets auf Zulässigkeit überprüft. Die Java Virtual Machine (JVM) ist die eigentliche Stackmaschine, die den Java-Bytecode ausführt.

Betrachten wir diese Schritte im einzelnen:

3.2.2.1 Die Bytecode-Verifikation

In der Phase der Übertragung (vgl. Abbildung 2) wird ein Applet u.a. einer sogenannten „Bytecode-Verifikation“ [Yellin 1995] unterzogen, in der verschiedene statische Tests durchgeführt werden. Dieser Verifikationsprozeß ankommender Applets führt –im strengen Sinne– keine Verifikation, sondern lediglich eine automatische Validierung des übertragenen Codes durch:

Zunächst wird der Bytecode auf syntaktische Korrektheit überprüft. Dabei wird beispielsweise festgestellt, ob die in der übertragenen Datei enthaltene Klassendefinition syntaktisch richtig ist, ob alle Argumente von Methodenaufrufen vorhanden und typgerecht sind, usw. Dies garantiert im wesentlichen, daß der übertragene Bytecode dem entspricht, was ein korrekt arbeitender Java-Compiler an Code erzeugen würde.

Neben diesen, im Prinzip einfachen Tests wird auch versucht sicherzustellen, daß zur Laufzeit keine undefinierten Zustände erreicht werden können; dies geschieht, indem überprüft wird, ob alle Kontrollflußoperationen legal sind (also nicht etwa in die Mitte von Schleifen oder nach außerhalb des Codes gesprungen wird), ob nur auf definierte Register zugegriffen wird, usw.

Letztlich wird eine Art Datenflußanalyse durchgeführt, die sicherstellen soll, daß zur Laufzeit keine Typverletzungen vorkommen und keine Über- oder Unterläufe des Laufzeitstacks vorkommen können. Das Vorgehen dabei läuft auf eine Fixpunktberechnung hinaus; [Yellin 1995] erwähnt auch einen Theorembeweiser, über den jedoch keine genaueren Angaben gemacht werden.

3.2.2.2 Der Klassenlader

Hat ein Applet die Bytecode-Verifikation erfolgreich durchlaufen, kommt es in die Ladephase. Die wichtigste Komponente hierbei ist der *Klassenlader*, dieser sucht und lädt Oberklassen, initialisiert die Objekte und verwaltet den Namensbereich der geladenen Klassen, wobei jedes Applet einen eigenen Namensbereich erhält. Dadurch können Applets nur die eigenen Methoden, sowie die zur Verfügung gestellten Systemmethoden benutzen. Weiterhin stellt der Klassenlader sicher, daß keine Systemklassen überschrieben oder manipuliert werden. Wichtig ist dabei, daß der Klassenlader von Applets nicht umgangen werden kann, insbesondere kann ein Applet auch keine Methoden des Klassenladers direkt aufrufen.

3.2.2.3 Der Security Manager („Sicherheitsmanager“)

Der Security Manager implementiert die Kontrolle über die Funktionen, die Applets ausführen dürfen, bestimmt also die „Grenzen“ der Sandbox. Alle potentiell gefährlichen Operationen können durch den Security Manager blockiert werden. Dies sind u.a. Dateizugriffe, Netzwerkverbindungen, die Definition neuer Klassenlader oder der Zugriff auf Betriebssystemfunktionen.

3.2.2.4 Die Funktionalität der Sandbox

Die Idee hinter einer Sandbox ist sehr einfach: einem „fremden“ Programm wird nicht die volle Funktionalität der Umgebung, in der es läuft, zur Verfügung gestellt, sondern lediglich eine eingeschränkte Auswahl. In diesem Bereich, der Sandbox, gibt es keine Ressourcen oder Funktionen, die dazu benutzt werden könnten, Schaden anzurichten.

Schwierig ist die Entscheidung, welche Funktionalität die Sandbox einem Programm zur Verfügung stellen soll: wird die Umgebung zu sehr eingeschränkt, sind keine sinnvollen Anwendungen mehr möglich, steht zu viel zur Verfügung, kann dies aber wieder zu Sicherheitsproblemen führen.

Als einfaches Beispiel betrachte man den Zugriff auf das lokale Dateisystem: erlaubt man den Zugriff, so kann ein Applet eventuell auf sicherheitskritische Informationen zugreifen, verbietet man ihn, sind viele nützliche Anwendungen (z.B.: ein Editor) nicht als Applet realisierbar.

Im Standardfall, also dem Laden eines unbekannten Java-Applets aus dem Netz, realisiert die Java-Sandbox folgende Einschränkungen:[4]

1. *Kein Zugriff auf das Dateisystem,*
2. *nur Netzwerkverbindungen zu dem Host, von dem das Applet geladen wurde,*
3. *kein Zugriff auf kritische Systemressourcen wie das Starten von Kommandos, Prozessen, etc.,*
4. *Nur sehr eingeschränkter Zugriff auf Systemeigenschaften.*

Diese Unterscheidung hat sich jedoch in der Praxis als zu grob erwiesen: viele sinnvolle Anwendungen sind daran gescheitert, daß über ein Netz geladene Applets nicht auf Funktionalität außerhalb der Sandbox zugreifen können; ernsthaften Anwendungen stoßen sehr schnell an Grenzen.

Sun hat dieses Manko erkannt und Mitte 1997 auf der großen Java-Konferenz JavaOne eine wesentliche Flexibilisierung angekündigt, die insbesondere auf der Einbindung digitaler Signaturen beruht. Näheres dazu im nächsten Kapitel.

4 Bewertung des Java-Sicherheitskonzeptes

Die Frage, ob das von Java angebotene Sicherheitskonzept hinreichend ist, läßt sich natürlich in ihrer Allgemeinheit nicht beantworten. Private Nutzer, die Java im wesentlichen als WWW-Erweiterung nutzen, haben (zur Zeit noch) wesentlich geringere Anforderungen als für einen professionellen Einsatz gelten müssen. Denkt man aber daran, Java in einem sicherheitskritischen Bereich wie dem TK-Netz oder im Bankenbereich einzusetzen, in dem Mißbrauch sehr leicht erheblichen finanziellen Schaden anrichten kann, müssen noch rigidere Kriterien gelten.

Wenden wir uns jedoch der prinzipiellen Kritik am Sicherheitskonzept von Java zu, wie sie im wesentlichen im Laufe des Jahres 1996 aufgekommen ist (vgl. z.B. [Dean & Wallach 1995, Sterbenz 1996, McGraw & Felten 1996]); hier sind zwei wesentliche Punkte zu nennen:

Das Sicherheitskonzept ist nicht hinreichend formal. Gemeint ist damit, daß das Sicherheitskonzept im wesentlichen auf einer informellen (textuellen) Beschreibung basiert, deren tatsächliches Zusammenspiel aus formaler Sicht unklar ist. Dies sei an zwei Beispielen erläutert:

- Der Prozeß der Bytecode-Verifikation erweist sich bei näherer Betrachtung als in seinen Auswirkungen nicht scharf faßbar: es gibt keine expliziten Aussagen darüber, welchen Bedingungen der Bytecode eines Applet tatsächlich genügt, wenn es diesen Prozeß erfolgreich durchlaufen hat. Solche Aussagen scheitern u.a. daran, daß Java bisher noch keine formale Semantik besitzt.

[4] Diese Beschränkungen gelten für „fremde“ Java-Applets, die aus dem Internet geladen wurden, lokal vorhandene Java-Anwendungen fallen nicht unter diese Restriktionen, können also auf alle Ressourcen, die die Java VM zur Verfügung hat, zugreifen.

Exkurs 4: Die wichtigsten Java-Fehler 1996

Februar: Der DNS-Spoofing Bug, entdeckt von Steve Gibbons und (unabhängig davon) von der Princeton-Gruppe, erlaubt es mit einem Java-Applet, Verbindungen zu beliebigen Rechnern im Internet aufzubauen. Insbesondere betrifft dies auch Rechner hinter einem Firewall.

März: David Hopwood beschreibt einen Implementierungsfehler im Klassenlader, der es erlaubt, alle Sicherheitsbeschränkungen für Java-Applets zu umgehen.

März: Die Princeton-Gruppe identifizierte einen Implementierungsfehler im Bytecode-Verifier, durch den Java-Applets beliebigen Maschinencode in der Laufzeitumgebung ausführen können.

Mai: Mit Hilfe eines von Tom Cargill beschriebenen Angriffs können Applets beliebigen Maschinencode in der Laufzeitumgebung ausführen.

Juni: David Hopwood beschreibt, wie das Java-Typsystem unterlaufen, und dadurch beliebiger Maschinencode ausgeführt werden kann.

August: Der „Array Name Bug“ (Ed Felten) erlaubt es, alle Sicherheitsbeschränkungen zu umgehen, indem das Typsystem von Java ausgeschaltet wird.

Auch 1997 setzte sich die Reihe der Fehler fort: es wurden weitere Fehler im Bytecode-Verifier entdeckt, und u.a. auch ein Fehler bei der Implementierung im Zusammenhang mit digitalen Signaturen. Insofern ist die Geschichte der Java-Sicherheitsfehler noch nicht abgeschlossen, und sie wird es auch voraussichtlich so schnell nicht sein. Der interessierte Leser sei auf die WWW-Seite von JavaSoft verwiesen [JavaSoft 1997], die weitere Informationen (auch bezüglich neuerer Fehler) enthält.

- Die äußerst sicherheitskritische Funktion des Klassenladers basiert –wie im Endeffekt das Sicherheitskonzept der ganzen Laufzeitumgebung– auf der Konsistenz des Typsystems von Java. Auch hier sind Arbeiten zwar im Gange [Drossopoulou & Eisenbach 1996], aber bisher existiert noch keine formale Beschreibung des kompletten Java-Typsystems, insofern entbehrt das Konzept des Klassenladers bisher eine theoretische Basis.

Das Sicherheitskonzept beruht auf einer „single line of defense“. Das Sicherheitskonzept beschränkt sich im wesentlichen auf eine statische Analyse des Bytecodes ankommender Applets und auf die Sandbox. Weitere Maßnahmen, die Angriff abwehren können, falls diese Hürde genommen wird, sind nicht vorgesehen.

Diese Defizite bei dem Sicherheitskonzept stellen nicht nur Bedenken aus theoretischer Sicht dar: Seit Java eingeführt wurde, sind eine ganze Reihe von Fehlern aufgetaucht, mit deren Hilfe man die Sicherheitsbeschränkungen umgehen konnte (vgl. Exkurs 4).

Die aufgetauchten Fehler wurden zwar i.d.R. schnell durch die Bereitstellung entsprechender Patches behoben, sie zeigen jedoch ein prinzipielles Problem auf, wenn Java tatsächlich in kritischen Bereichen wie einem TK-Netz genutzt werden sollte: Ist eine fehlerhafte Laufzeitumgebung erst einmal bis zu den Endkunden verteilt, gestaltet sich ein Update dieser Software wesentlich schwieriger als in einem System von vernetzten Rechner, bei dem das Einspielen neuer Software eine Routineangelegenheit ist. Unter diesen Voraussetzungen ist es kaum akzeptabel, mit potentiellen Sicherheitslöchern leben zu müssen, und ein wesentlich besserer Schutz vor den damit verbundenen Problemen ist gefragt.

4.1 Ausblick: neuere Entwicklungen

Die beschriebenen Defizite des Sicherheitskonzeptes sind natürlich nicht unbemerkt geblieben und in diversen Arbeiten wurde versucht, den Problemen prinzipiell zu begegnen. Dies geschieht zur Zeit im wesentlichen in zwei Bereichen, mit Hilfe kryptographischer Ansätze und mit dem Einsatz formaler Methoden:

4.1.1 Integration kryptographischer Methoden

In einem Szenario wie bei Java, in dem Programme über ein offenes geladen werden, ist es naheliegend mit Hilfe digitaler Signaturen sicherzustellen, daß die transferierte Information authentisch ist. Dabei wird wie folgt vorgegangen:

1. Ein Applet wird durch eine "trusted (third) party" zertifiziert und mit einer digitalen Signatur versehen, bevor es im Netz zur Verfügung gestellt wird.
2. Nachdem auf Client-Seite ein Applet geladen wurde, wird dessen digitale Signatur geprüft. Dies garantiert, daß das Applet tatsächlich von der vermuteten Stelle stammt, und daß es nicht manipuliert (verändert) wurde.

Wichtig ist dabei, daß die Authentizierung erst einmal *nichts* über die tatsächliche Funktion des Codes aussagt. Ein signiertes Applet kann im Prinzip immer noch Schaden anrichten, man weiß allerdings (im Idealfall) wer dafür verantwortlich ist. Aber auch dies ist nicht notwendigerweise hilfreich, denn ein solches Applet könnte ja (unbewußt) selbst ein Sicherheitsloch enthalten, das von Dritten ausgenutzt werden kann (sog. transitive trojanische Pferde). Um dem entgegenzuwirken, müßte die Zertifizierungsstelle eine Codeinspektion oder gar eine Verifikation durchführen.

Trotz dieser Einschränkungen machen digitale Signaturen Sinn: damit lassen sich beispielsweise die üblichen in Firmen geltenden Regeln, daß Software nur eingesetzt werden darf, wenn diese zentral eingekauft wurde, in ein Java-basiertes Intranet übertragen. In diesem Fall zertifiziert eine Stelle die zur Verfügung gestellten Applets, und diese können dann (mit den selben Risiken wie andere Software) im Netz genutzt werden.

Problematisch ist so ein Konzept jedoch, wenn es auf das ganze Internet skaliert wird, wie etwa im Falle von ActiveX: hier stellt sich die Frage, inwieweit eine zertifizierende Instanz überhaupt eine effektive Kontrolle über Programme ausüben kann.

Ein gutes Beispiel für die Problematik ist „Exploder", ein ActiveX control, das 1996 dadurch Schlagzeilen machte, daß es in der Lage war, einen Windows 95 PC buchstäblich auszuschalten [McLain 97]:

> *Exploder performs a clean shutdown of Windows 95 from a web page. On „Green Machines", particularly those with a power conservative BIOS, (mostly laptop computers) it also turns the power off after shutdown [...] it's the same thing as the „Shut Down" menu item on the „Start" button, but with the power off feature added.*

Besonders problematisch war dabei, daß Exploder digital signiert war, und somit eine gewisse „Sicherheit" suggeriert wurde [McLain 97]:

> *Code Signing simply attempts to identify who signed the control. [...]. You go to a web site, give them a name, address, credit card number and some other stuff (none of which have to be yours), click „I Agree" on a page full of legal jargon, and pretty soon you get an e-mail with the information you need to sign the control in it. Once you have your Digital ID, you can sign any unsigned ActiveX control. Nobody reviews these controls!*

Digitale Signaturen besitzen durchaus das Potential, ein Sicherheitskonzept wie das von Java sinnvoll zu ergänzen; ein Allheilmittel sind sie jedoch nicht.

In neueren Versionen von Java (ab JDK 1.1.1) kann mit signierten Applets gearbeitet werden, im wesentlichen um die Laufzeitbeschränkungen von Applets aufzuheben: eine als vertrauenswürdig eingestufte Java-Klasse wird dann genauso behandelt wie eine lokal gespeicherte Klasse und hat somit Zugriff auf alle Systemressourcen. Zusätzliche Erweiterungen in Richtung auf die Nutzung expliziter Sicherheitspolitiken („security policies") sind geplant, bzw. möglich [Gong 1996, Wallach et.al. 1997], zum Zeitpunkt der Verfassung dieses Artikels lagen hierzu aber nur rudimentäre Informationen von Sun dazu vor.

4.2 Formale Methoden

Eine vollständig fehlerfreie Java-Laufzeitumgebung, die garantiert keine Sicherheitslöcher mehr enthält, ist zwar wünschenswert, in der Praxis aber kaum erreichbar. Ein Schritt in diese Richtung könnte durch den Einsatz formaler Methoden zur Validierung oder Verifikation gemacht werden.

Als ersten Schritt kann man die formale Behandlung der Bytecode-Verifikation sehen: der bisher nur informell beschrieben Prozeß läßt sich z.B. zum Teil als Aufgabe für einen Model-Checker formulieren, was den Vorteil hat, daß die zu überprüfenden Bedingungen an den Bytecode explizit werden. Arbeiten in diese Richtung finden derzeit am Technologiezentrum der Deutschen Telekom in Darmstadt statt.

Eine andere Alternative zur Bytecode-Verifikation ist *Proof-Carrying Code* (PCC) [Necula & Lee 1996, Feigenbaum & Lee 1997] und basiert auf folgender Idee: Bestimmte, sicherheitsrelevante Eigenschaften von Applets werden mit Hilfe eines formalen Beweises zugesichert, der dann als Ausdruck im typisierten λ-Kalkül mit dem Applet zum Empfänger übermittelt wird. Der Nutzer kann dann durch Type-Checking den Beweis nachvollziehen und ist somit sicher, daß der übertragene Code die behaupteten Eigenschaften besitzt. Die Übertragung des Beweises könnte zwar auch dadurch ersetzt werden, daß eine Art „Verifikationsserver" den Beiweis einmal nachvollzieht, und dann die Eigenschaft mittels einer digitalen Signatur an das Applet bindet; der Vorteil von Pcc liegt jedoch darin, daß auf die für digitale Signaturen notwendige Infrastruktur verzichtet werden kann und das Verfahren etwas flexibler ist –um den Preis des Mehraufwandes für das Proof-Checking.

Die wesentlichen offenen Fragen im Zusammenhang mit PCC liegen auf der Hand*: Woher kommen die Beweise? Wie genau sehen die zugesicherten Eigenschaften aus? Wozu nutzt der Empfänger diese?* [Feigenbaum & Lee 1997] argumentieren, daß bestimmte Eigenschaften (wie z.B. Typsischerheit) automatisch von speziellen Compilern geführt werden könnten, diese Technologie ist aber zur Zeit noch nicht verfügbar. Klar ist jedoch, daß dadurch nur entscheidbare Eigenschaften nachgewiesen werden können, zumindest solange die Beweissuche vollautomatisch vonstatten gehen soll. Trotz aller gesunden Skepsis deutet sich hier eine sehr interessante Entwicklung an, um aus den gut abgesicherten Ergebnissen im Bereich formaler Methoden in einem hoch aktuellen Bereich Nutzen ziehen zu können.

Die erste formal verifizierte Umgebung für Java-Programme dürfte wahrscheinlich auf einer Smartcard zu finden sein: Sicherheitslücken sind bei Smartcards hoch problematisch, und ein Austausch der Software auf einmal verteilten Chipkarten ist kaum machbar. Entsprechend hoch sind die Sicherheitsanforderungen an die Software auf Smartcards. Da die virutelle Java-Maschine auf einer Smartcard („Java Card VM") sehr klein ist (ca. 4 KB) läßt sich eine vollständige, formale Modellierung der VM hier tatsächlich mit vertretbarem Aufwand durchführen. Diese Arbeiten werden zur Zeit von der in Deduktionskreisen wohlbekannten Firma Computational Logic Inc. in den USA angegangen [Cohen 97].

Neben einer verifizierten Java-VM sind auch formal verifizierte Applets selbst interessant, da die tatsächliche Funktionalität des übertragenen Programms, bzw. dessen Eigenschaften, nachgewiesen werden könnte. Leider sind wir jedoch noch einiges davon entfernt, Java Programme formal verifizieren zu können, u.a. fehlt dazu noch die formale Semantik der Sprache. Arbeiten in dieser Richtung sind jedoch im Gange [Drossopoulou & Eisenbach 1996, Syme 97].

Betrachtet man die Möglichkeiten der Einbindung von Spezifikationen der Funktion von Applets genauer, ergeben sich äußerst interessante Fragestellungen für weitere Forschungsarbeiten:

- Der derzeitige Prozeß der Bytecode-Verifikation arbeitet mehr oder weniger auf der Syntax des übertragenen Bytecode. Ließe sich hier Information über die Semantik des zu überprüfenden Applets mit einbringen?
- Es hat sich gezeigt, daß eine „one-size-fits-all" Konfiguration der Java-VM nicht ausreichend ist, da bestimmte Anwendungen auch auf Funktionalität außerhalb der Sandbox zugreifen müssen. Will man nun flexibler werden, und „maßgeschneiderte"

Beschränkungen für ankommende Applets einrichten, wäre auch hier eventuell Information über die tatsächliche, bzw. zu erwartende Funktion eines Applets hilfreich.

- Konsequent fortgedacht, bieten Java-Applets eine Umgebung, in der jede Funktionalität aus dem Netz geladen wird, und lokal keine Software mehr installiert ist (sog. Network-centric computing). Wie findet man jedoch in einer solchen Umgebung die Applets, die das ausführen, was man möchte? Man sucht hier nicht (wie derzeit meist im WWW) nach *Information*, sondern nach *Funktion*. Evenuell sucht man gar nach einer Funktion, die ein einzelnes Applet nicht allein liefert, sondern nur eine Kombination verschiedener Applet. Auch bei dieser Problematik (sog. „Trading“ [Puder & Geihs 96]) wären funktionale Spezifikationen interessant.

Zusammenfassend läßt sich festhalten, daß der Einsatz formaler Methoden im Kontext von Java sehr vielversprechend erscheint; erste Ansätze, wie im Fall der Java Card VM oder Proof Carrying Code, sind bereits vorhanden. Java ist für den Einsatz formaler Methoden insbesondere dadurch interessant, daß die Programme tendenziell eher klein, und somit potentiell eher handhabbar sind als „klassische“ Software.

5 Schlußfolgerungen

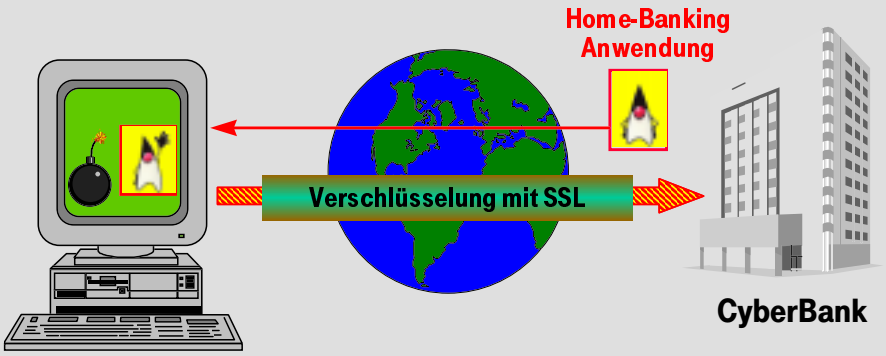
Java bietet äußerst interessante Perspektiven, birgt jedoch auch ein inhärentes Sicherheitsproblem, denn das Laden und Ausführen von Anwendungen über ein Netz ist hoch problematisch. Java verspricht hier ein sicheres Gesamtkonzept, das im wesentlichen auf einem Sandbox-Ansatz beruht, bei dem potentiell unsicheren Programmen nur eine eingeschränkte Funktionalität zur Verfügung stellt.

Zusammenfassend läßt sich feststellen, daß das Sicherheitskonzept der Java-Umgebung zwar vielversprechende Ansätze zeigt, aber zur Zeit für sicherheitskritische Anwendungen noch unzureichend ist. Zu kritisieren sind der fehlende formale Unterbau, und die Beschränkung auf ein einziges Prinzip um die Sicherheit zu garantieren („single line of defense“). Eine vollständige und allgemeine Lösung für die inhärent mit Java verbundene Sicherheitsproblematik erscheint –auch langfristig– unwahrscheinlich: auf der einen Seite muß die Java-Umgebung genügend Funktionalität für Applets zur Verfügung stellen, damit sinnvolle Anwendungen realisiert werden können, auf der anderen Seite ist aus der Sicherheitsperspektive eine möglichst geringe Funktionalität –idealerweise ohne Seiteneffekte– wünschenswert. Trotzdem ist eine formalere Herangehensweise sinnvoll und notwendig, da nur so definitive Aussagen über das Sicherheitskonzept möglich sind.

Die aus sicherheitstechnischer Sicht interessanteste, aktuelle Entwicklung ist die Einbindung kryptographischer Methoden in das Konzept von Java, die es möglich macht, Ursprung und Authentizität von Applets mit einzubeziehen. Auch diese Technologie ist jedoch kein „Selbstläufer“, sondern erfordert einiges an Infrastruktur, wie z.B. ein Trust-Center. Auch sollte die tatsächliche Aussagekraft einer digitalen Signatur nicht überschätzt werden: i.d.R. kann dadurch keine „Gutmütigkeit“ eines Programms „bescheinigt“ werden, sondern lediglich Verantwortung zugewiesen werden.

Bei der Anwendung von Java-Technologie sollte man sich der inhärenten Sicherheitsproblematik explizit bewußt sein, und im Einzelfall prüfen, inwieweit die derzeitige Technologie akzeptabel ist. Dabei darf nicht von einer fehlerfreien Implementierung ausgegangen werden, d.h. mit Sicherheitslöchern muß bis auf weiteres gerechnet werden. Weiterhin ist es unabdingbar, immer auch das System als Ganzes zu analysieren, denn eine noch so sichere Komponente allein garantiert keinen Schutz. Ein gutes Lehrbeispiel dafür sind die kürzlich aufgetretenen Probleme im Bereich Online-Banking (vgl. Exkurs 5). Die von vielen erwartete kommerzielle Nutzung von Internet-Technologien erfordert explizites Know-how im Sicherheitsbereich und eine dauernde Beobachtung der Entwicklung. Beides ist auch unumgänglich für kommerzielle Anwendungen von Java in einem offenen Netz.

Exkurs 5: Virenangriff auf eine Java-basierte Home-Banking Anwendung.



Eine Bank bot Ihren Kunden die Möglichkeit Java-basiert Überweisungen zu tätigen. Dazu wird ein Applet, mit dem der Kunde die Transaktion spezifiziert, übertragen und die Transaktion schließlich über eine verschlüsselte SSL-Verbindung [Freier et.al 1995] zur Bank übertragen.

Der Angriff auf diese, auf den ersten Blick sehr sichere Vorgehensweise, bestand in einem Virus, der (auf anderen Wegen verbreitet) Netscape infizierte, dort auf die typische Sequenz einer Transaktionsübertragung wartete, und die Transaktion noch vor der Verschlüsselung veränderte.

Moral: ein System ist nur so sicher wie die schwächste Komponente, in diesem Fall die Windows-Plattform. Die Schwachstelle konnte jedoch sehr schnell durch Austausch des Applets beseitigt werden: ein klares Plus für Java, den ein Software-Update bei allen Kunden hätte wesentlich länger gedauert.

6 Literatur

[CIAG 1996] CIAG: *Winword Macro Viruses (Concept, DMV, Nuclear, Colors, FormatC, Hot).* The U.S. Department of Energy, Computer Incident Advisory Capability, Information Bulletin, G-10, 1996. (Archiviert auf: http://ciac.llnl.gov/)

[Cohen 97]: RICHARD M. COHEN*: The Defensive Java Virtual Machine Specification Version 0.5*. Computational Logic, Inc., Austin, Texas, 1997. http://www.cli.com/software/djvm/

[Dean & Wallach 1995] DEAN, D, & WALLACH, D. S. 1995. *Security Flaws in the HotJava Web Browser*. http://www.cs.princeton.edu/sip/, 1996.

[Dean 1997] DREW DEAN: *The Security of Static Typing with Dynamic Linking*. Proceedings of the Fourth ACM Conference on Computer and Communications Security, April 1997.

[Drossopoulou & Eisenbach 1997] DROSSOPOULOU, S., EISENBACH, S. *Is the Java type system sound?* Proceedings of the Fourth International Workshop on Foundations of Object-Oriented Languages, Paris, Jan. 1997.

[Feigenbaum &Lee 1997] FEIGENBAUM, J., LEE, P.: *Trust Management and Proof-Carrying Code for Mobile Code Security*. DARPA Workshop on Foundations of Mobile Code Security, Monterey, California, USA, 26-28 March 1997.

[Freier et.al. 1995] FREIER, A. O., KARLTON, P., KOCHER, P. C. *The SSL Protocol: Version 3.0*. Internet draft, Mar. 1996. ftp://ietf.cnri.reston.va.us/internet-drafts/draft-freier-ssl-version3-01.txt.

[Gong 1996] GONG, LI: *New Security Architectural Directions for Java*. Proc. IEEE COMCON, San José, CA:, pp. 97-102, Feb. 1997.

[Gosling & Steele 1996] GOSLING, J., STEELE, G. The Java Language Specification. Addison-Wesley, 1996.

[Java Security FAQ 1997] SUN MICROSYSTEMS: Java Security FAQ.

http://java.sun.com/sfaq, 1997.

[JavaSoft 1997] JAVASOFT: *Java Security*. http://java.sun.com/security/, 1997.

[Lindholm & Yellin 1996] LINDHOLM, T., YELLIN, F. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[McGraw & Felten 1996]: GARY MCGRAW AND EDWARD W. FELTEN: *Java Security: Hostile Applets, Holes and Antidotes*. John Wiley and Sons, New York, 1996.

[McLain 1997] FRED MCLAIN: *ActiveX or how to put nuclear bombs in web pages*. http://www.halcyon.com/mclain/ActiveX/welcome.html, 1997.

[Microsoft 1996] MICROSOFT CORP.*: Proposal for Authenticating Code via the Internet*. http://www.microsoft.com/intdev/security/authcode/, Apr. 1996.

[Necula & Lee 1996] NECULA, G., LEE, P. *Proof-Carrying Code*. CMU Technical Report CMU-CS-96-165, school of Computer Science, Carnegie Mellon Univ., Pittsburg, PA, Sept. 1996. http://www.cs.cmu.edu/people/petel/papers/pcc/pcc.html

[Puder & Geihs 1996] A. PUDER AND K. GEIHS: *System support for knowledge-based trading in open service markets*. 7th ACM SIGOPS European Workshop, Connemara, Ireland. 1996.

[Sterbenz 1996] ANDREAS STERBENZ: *An Evaluation of the Java Security Model*. Twelfth Annual Computer Security Applications Conference, 1996.

[Sun Microsystems 1995] SUN MICROSYSTEMS: *HotJava™: The Security Story*. http://java.sun.com/1.0alpha3/doc/security/security.html, 1995.

[Syme 1997] DON SYME: *Proving Java Type Soundness*. Technical report, Univ. of Cambridge Computer Laborator, Automated Reasoning Group, Combridge, Englan, 1997. http://www.cl.cam.ac.uk/users/drs1004/

[Wallach et.al. 1997]DAN S. WALLACH, DIRK BALFANZ, DREW DEAN, AND EDWARD W. FELTEN: *Extensible Security Architectures for Java*. Technical Report 546-97, Department of Computer Science, Princeton University, April 1997. http://www.cs.princeton.edu/sip/pub/extensible.html

[Yellin 1995] YELLIN, F.: *Low Level Security in Java*. In: WWW4, 1995. http://www.w3.org/pub/Conferences/WWW4/Papers/197/40.html.