DuD
Datenschutz und Datensicherheit

Sicherheitsaspekte von Java-Applets

Holger Mack

Java ist in den vergangenen Jahren immer wieder als eines der größten Sicherheitsrisiken beim Zugriff auf das WWW ins Gerede kommen. Im Unterschied zu ActiveX besitzt Java jedoch ein ausgeklügeltes Sicherheitsmodell, wie der folgende Beitrag zeigt. In der Vergangenheit sind immer wieder Sicherheitslücken in den gängigen Implementierungen von Java bekannt geworden. Trotz dieser Lücken und einiger Probleme mit dem Java-Sicherheitsmodell, sind derzeit keine Fälle bekannt, bei denen Angriffe unter Ausnutzung dieser Lücken Erfolg hatten.

Dipl.Ing(FH)
Holger Mack

IT Security Consultant, Secorvo Security Consulting GmbH, Schwerpunktgebiete: Netzwerk- und Internet Sicherheit, PKI, E-Mail Sicherheit.
E-Mail: mack@secorvo.de

Einleitung

Seit seiner Einführung im Sommer 1995 ist Java oft als eine potentielle Quelle von Sicherheitsproblemen im World Wide Web bezeichnet worden. Viele Unternehmen und Benutzer haben sich daher entschieden, die Ausführung von Java-Applets innerhalb von WWW-Seiten aus dem Internet unternehmensweit zu sperren bzw. unterbanden das Herunterladen von Java-Applets durch entsprechende Filter auf der Firewall.

Der Verlust an Funktionalität bzw. an Information durch den Verzicht auf Java war bislang gering, da Java hauptsächlich zum Verschönern von Web-Seiten durch animierte grafische Elemente eingesetzt wurde.

Die Weiterentwicklung und die Verbreitung von Java haben dieses Bild verändert. Java wird heute mehr und mehr zur Unterstützung und Entwicklung von verteilten Anwendungen verwendet sowie dazu eingesetzt, das WWW-Angebot durch interaktive Anwendungen, sogar Sicherheitsmechanismen (siehe z.B. Bank24) zu erweitern. Mehr und mehr Firmen verwenden Java auch, um interne verteilte Anwendungen auf dem firmen-internen Intranet zur Verfügung zu stellen. Ein Abschalten des Java-Interpreters im Browser führt dazu, daß Benutzer nicht mehr das ganze Spektrum der Internet- bzw. Intranet-Angebote nutzen können.

Benutzer und Systemadministratoren sind deshalb mit der zunehmend schwierigen Frage konfrontiert, zwischen dem Sperren von Java aus Sicherheitsgründen und der Freigabe der vollen Funktionalität des Internets und des Intranets entscheiden zu müssen

Die Entwickler haben für Java gleich zu Beginn ein detailliertes Sicherheitsmodell entwickelt. Dadurch unterscheidet sich Java deutlich von ähnlichen Systemen wie z. B. ActiveX und Skript-Sprachen wie JavaScript und VBScript, die oftmals den vollen Zugriff auf das lokale Betriebssystem freigeben und für die kein Sicherheitsmodell existiert.

Gegenstand dieses Beitrags ist es, den Ansatz des Java-Sicherheitsmodells vorzustellen und aufzuzeigen, inwieweit es den Sicherheitserfordernissen von Internet-Nutzern gerecht werden kann.

Zunächst wird zusammengefaßt, welche Probleme, Bedrohungen und Risiken durch die Nutzung von aktiven WWW-Komponenten aus dem Internet entstehen. Anschließend wird das Java-Sicherheitsmodell beschrieben und diskutiert.

Im darauffolgenden Abschnitt werden die aktuellen Erweiterungen des Java-Sicherheitsmodells beschrieben. Abschließend erfolgt eine Bewertung.

1 Abgrenzung

Die Sicherheitsbetrachtungen in diesem Beitrag beziehen sich auf den Fall, daß Java-Programme in Form sogenannter Java-Applets aus unkontrollierten Quellen (z. B. über das Internet) bezogen werden.

Java kann auch zur Entwicklung von Anwendungen, die fest auf dem System installiert sind, eingesetzt werden. Solche Java-Programme stellen aus sicherheitstechnischer Sicht kein anderes Problem dar als Anwendungen, die mit Hilfe anderer Programmiersprachen entwickelt wurden. Die strengere Sprachspezifikation von Java und die Tatsache, daß Java einige Operationen (z. B. Pointer-Arithmetik), die in traditionellen Programmiersprachen immer wieder zu Problemen führen, unterbindet, sollte dazu führen, daß Java-basierte Programme insgesamt problemloser laufen.

2 Problemdefinition

Das Aufkommen von Skript- und Programmiersprachen wie Java, JavaScript, VB-Script und ActiveX haben ein neues Problem bei der Nutzung des WWW gebracht. Bis dahin bestanden WWW-Seiten ausschließlich aus passiven Elementen wie z. B. Texten, Grafiken, Video- und Sounddaten. Diese Objekte stellen keine direkte Bedrohung dar, da sie nur vom Browser des Benutzers dargestellt werden, aber nicht

aktive Kommandos auf dem Zielrechner ausführen können.[1]

Java-Applets hingegen haben ihre Stärke gerade darin, daß sie aktive Operationen auf dem Rechner des Benutzers ausführen können. In erster Linie führt dies dazu, daß Entwickler viele neue Möglichkeiten haben, die Darstellung und Funktionalität ihrer Web-Seiten zu erhöhen, sie können sogar verteilte Anwendungen entwickeln. Allerdings bringen solche Programme neue Sicherheitsproblemene mit sich.

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

- Das Programm kann Viren enthalten, die das System infizieren und schädigen.
- Das Programm könnte als „trojanisches Pferd“ unbekannte, schädigende Nebeneffekte haben, die vom Entwickler eingebaut wurden. Ein Beispiel dafür sind die Angriffe auf Nutzer des T-Online-Systems, die kürzlich bekannt wurden [LUC_98].
- Das Programm kann vom Entwickler nicht beabsichtigte Nebeneffekte haben (z. B. durch Programmierfehler).

Die Auswirkungen solcher Gefährdungen können dabei verschiedener Art sein:

- Veränderungen des Systems
- Einsicht in vertrauliche Daten des Benutzers
- Angriffe auf die Verfügbarkeit eines Systems (Denial of Service)

Zum Schutz vor solchen Bedrohungen werden üblicherweise verschiedene Methoden angewendet:

- Programme werden vor dem ersten Ausführen mit Hilfe von Virenscannern nach bekannten Virenmustern abgesucht.
- Programme werden zuerst in einer Test-Umgebung (z.B. einem „Quarantäne-Rechner“) ausgeführt, um zu untersuchen, ob unerwünschte Nebeneffekte auftreten.
- Es werden nur Programme von sogenannten vertrauenswürdigen Quellen (z. B. renommierte Hersteller) auf dem System ausgeführt.

Wenn es sich um traditionelle Programme handelt, die im allgemeinen fest auf dem Zielsystem installiert werden, sind diese Methoden relativ effektiv durchführbar ohne größere Einbußen in der Funktionalität in Kauf nehmen zu müssen.

Alle drei Ansätze sind allerdings wenig geeignet für Java-Applets. Das Hauptproblem liegt darin, daß Anwendungen, die auf der Java-Applet-Technologie beruhen, im Gegensatz zu traditionellen Anwendungen nicht langfristig auf dem Zielsystem installiert und gespeichert werden, sondern jeweils beim Aufrufen der Web-Seite dynamisch neu auf das System geladen werden. Dabei ist es durchaus möglich, daß beim erneuten Aufrufen einer Anwendung geänderter Code (z. B. neue Version, Bug-fixes etc.) geladen wird. Ohne Einbußen bei der Dynamik und Funktionalität ist es deshalb nicht möglich, die Applets erst in einer geschützten Umgebung auszuführen. Auch ist es ohne zusätzliche Maßnahmen nicht möglich, die Herkunft eines Applets eindeutig festzustellen und nur Applets aus vertrauenswürdigen Quellen auszuführen.

Das Problem von Virenscannern ist, daß sie grundsätzlich nur solche Viren identifizieren können, die bereits bekannt sind. Es besteht also immer die Gefahr, daß neue Viren in das System gelangen, die von der Virenscanner-Software nicht erkannt werden. Die Hersteller von Virenscannern haben zunehmend Schwierigkeiten, mit der ständig wachsenden Anzahl der Viren und Viren-Varianten Schritt zu halten; deshalb wird auch hier nach anderen Wegen gesucht.

Eines der Hauptmerkmale von Java ist die Plattformunabhängigkeit. Java-Applets können schon heute ohne Veränderungen auf einer großen Anzahl verschiedener Betriebssysteme ausgeführt werden (z. B. MS-Windows, MAC-OS, UNIX). Plattformunabhängigkeit an sich stellt kein Sicherheitsproblem dar, allerdings kann es die Folgen und die Verbreitung eines erfolgreichen Angriffs erhöhen. Bis heute hat sich die Vielfalt der am Internet angeschlossenen Plattformen aus sicherheitstechnischer Sicht als eine „natürliche Barriere“ ausgewirkt. Dadurch waren Sicherheitsprobleme oft auf eine Plattform beschränkt.[2] Ein Angriff, der mit Java realisiert würde, würde hingegen nahezu alle gängigen Plattformen gefährden. Welche Folgen das haben kann, ist am Beispiel der rasanten Verbreitung von Macro-Viren zu sehen.

Die Tatsache, daß Java-Applets auf dem Zielsystem ausgeführt werden, kann dazu führen, daß ein Rechner, über den ein Applet die Kontrolle gewonnen hat, als Ausgangspunkt für weitere Angriffe verwendet werden kann (z. B. durch Versenden gefälschter E-Mail oder durch Ausnutzung von Vertrauensbeziehungen z. B. bei UNIX- oder Windows NT-Systemen). Das Applet kann dabei ausnutzen, daß bei vielen Netzwerken die Sicherheitsmechanismen auf die Firewall beschränkt sind. Da Java-Applets im internen Netzwerk ausgeführt werden, sind diese Kontrollen wirkungslos.

3 Sicherheitsmodell

Sun hat mit dem Java-Sicherheitsmodell einen proaktiven Ansatz gewählt. Ziel des Modells war es, das Java-System von vorneherein so zu gestalten, daß Java-Applets auf dem Zielrechner ausgeführt werden können, ohne daß eine Gefahr für das System besteht.

3.1 Voraussetzungen

Der Ansatz des Sicherheitsmodells ist, in das Java-System Mechanismen einzubauen, mit deren Hilfe die Ausführung sicherheitskritischer Operationen auf einem Zielrechner kontrolliert werden können. Sun entwickelte deshalb das Prinzip der „Sandbox“[3]. Die Idee ist, daß Applets innerhalb einer vom Benutzer kontrollierten, eingeschränkten Systemumgebung, einer „virtuellen Java-Maschine“ ausgeführt werden. Zugriffe auf Ressourcen außerhalb dieser Systemumgebung sollten nur erlaubt sein, wenn sie der Sicherheits-Policy der jeweiligen Umgebung entsprechen.

Für die Kontrolle sind im Java-Sicherheitsmodell drei Teile verantwortlich, deren fehlerfreie Funktion entscheidend für das sichere Ausführen von Java-Applets ist:

- der Security Manager,
- der Byte Code Verifier und
- der Class Loader.

In der Literatur werden der Class Loader, der Security Manager und der Byte Code Verfier manchmal als „the three lines of defence“ des Java-Sicherheitsmodells bezeichnet. Schon in [GRFE_96] wird darauf hingewiesen, daß dies einen falschen Eindruck hinterläßt. Die Teile stellen keine drei Verteidigungslinien dar, die alle von einem Angreifer überwunden werden müssen.

[1] Gefährdungen können allerdings bei Ausführung der zugehörigen Anwendungen entstehen; das ist aber kein spezifisches Problem von Daten im WWW.

[2] Beispiele dafür sind DOS-Viren oder der Internet-Wurm [EIRO_88].

[3] Siehe auch Fox, DuD 2/1998, S. 96.

Vielmehr setzt das Java-Sicherheitsmodel ein einwandfreies Funktionieren aller drei Teile voraus.

3.2 Java Virtual Maschine

Java-Applets werden nicht direkt vom Betriebssystem ausgeführt, sondern in einer plattformunabhängigen Form, dem sogenannten Byte Code, auf das System geladen und von der Java Virtual Machine (JVM) in systemspezifischen Maschinencode umgesetzt (interpretiert). Die JVM ist der maschinenabhängige Teil des Java-Systems, der dafür sorgt, daß Java Applets auf verschiedenen Systemen ausgeführt werden können, und ist üblicherweise Teil eines Java-fähigen Browsers.

Ein Java-Applet kann nur mit Hilfe der JVM auf Ressourcen des Betriebssystems zugreifen; dort werden die Zugriffe auf Systemressourcen kontrolliert. Der Teil, mit dem die Zugriffskontrolle realisiert wird, ist der Security Manager. Abhängig von der jeweiligen Implementation wird von diesem eine Zugriffsentscheidung getroffen, wenn ein Applet auf eine kritische Ressource zugreifen will.

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

Die Operationen, die als gefährlich eingeschätzt werden und deshalb mit Hilfe des Security Managers kontrolliert werden können, sind:

- Netzwerkzugriffe,
- das Laden von Applets,
- Zugriffe auf Java-Klassen,
- alle Operationen zum Manipulieren von und der Zugriff auf Threads,
- der Zugriff auf Systemressourcen wie z. B. die AWT Event Queue,
- das Erzeugen von Toplevel Windows,
- Zugriffe auf das Dateisystem, sowie
- das Aufrufen von lokalen Programmen und Betriebssystem-Kommandos.

3.3 Byte Code Verifier

Ein großer Teil der Sicherheitsmechanismen von Java sind eng mit der Sprachspezifikation von Java verknüpft. Auf den ersten Blick ähnelt Java C++, allerdings ist Java sehr viel strikter spezifiziert, um problematische Spracheigenschaften zu vermeiden. So gibt es in Java z. B. keine Zeiger, und ein direkter Speicherzugriff ist nicht möglich. Außerdem ist die objektorientierte Sprache Java stark typgebunden, mit Zugriffsregeln für die Methoden und Parameter eines Objekts. Speziell diese Zugriffsregeln sind entscheidend für das Java-Sicherheitsmodell.

Die Aufgabe des Byte Code Verifiers ist es, zu prüfen, ob die geladenen Applets der Java-Sprachspezifikation entsprechen. Das Einhalten der Sprachspezifikation sollte bereits von Java-Compilern geprüft werden. Da bei Java-Applets von unbekannten Quellen nicht davon ausgegangen werden kann, daß der Byte Code mit einem vertrauenswürdigen Compiler (oder überhaupt einem Compiler) erzeugt wurde, überprüft der Byte Code Verifier vor Ausführen eines Applets, ob es der Java-Sprachspezifikation entspricht. Dabei versucht er auch festzustellen, ob ein Applet die Kontrollen des Security Managers umgeht.

Die Byte Code-Dateien (.class-Files), die als Teil eines Applets auf das Zielsystem geladen werden, enthalten neben den auszuführenden Operationen noch Zusatzinformationen, die den Prozeß des Byte Code Verifiers unterstützen. Der Byte Code Verifier versucht in mehreren Durchläufen zu erkennen, ob ein Applet irreguläre Zustände verursachen kann. Eine detailierte Beschreibung der Vorgehensweise des Byte Code Verifiers kann in [YELL_96] gefunden werden.

3.4 Class Loader

Die Aufgabe des Class Loaders ist es, dem auszuführenden Applet alle erforderlichen Klassenbibliotheken zur Verfügung zu stellen.[4] Von entscheidender Bedeutung ist dabei die Herkunft einer Klasse. Die Klassen, die lokal in einem speziellen Verzeichnis (spezifiziert in der Umgebungsvariable CLASSPATH) abgelegt sind, werden nicht denselben Kontrollen unterzogen wie Klassen, die über das Internet geladen werden. Wenn ein Applet eine entsprechende Klasse benötigt, durchsucht der Class Loader zuerst die lokalen Klassen und anschließend erst externe Quellen (z. B. das Herkunftsverzeichnis des Applets im Internet).
Der Class Loader sorgt ebenfalls dafür, daß verschiedene Applets, die gleichzeitig auf dem System ausgeführt werden, sich nicht gegenseitig beeinflussen können. Dies wird mit Hilfe eines getrennten Namensraums für jedes Applet realisiert.

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

3.5 Grenzen des Modells

Das Java-Sicherheitsmodell ist auf Grund verschiedener Eigenschaften kritisiert worden. Auf die wichtigsten soll hier kurz eingegangen werden:

- Komplexität sicherheitskritischer Teile

In der Literatur wird oft darauf hingewiesen, daß es das Ziel sein sollte, den sicherheitskritischen Teil eines Systems (Trusted Computing Base, TCB) so klein und einfach wie möglich zu gestalten. Denn je größer dieser Teil ist, desto schwieriger ist es, ihn fehlerfrei zu implementieren bzw. nachzuweisen, daß er als Ganzes effektiv ist. Ist er dagegen kompakt, kann möglicherweise die Fehlerfreiheit formal verifiziert werden. Im Java-System ist der sicherheitskritische Teil relativ groß (ca. 23.000 Code-Zeilen), was eine intensive Überprüfung sehr aufwendig werden läßt. Die Schwierigkeit, das Java-Sicherheitsmodell korrekt zu implementieren, manifestiert sich auch in den bei Browser-Implementierungen immer wieder aufgetretenen Problemen [CERT1, CERT2].

- Keine eindeutige Relation zwischen Byte Code und Java Code

Es gibt keine Möglichkeit nachzuweisen, daß es nicht möglich ist, Byte Code zu schreiben, der zwar nicht der Java-Sprachdefinition entspricht, aber trotzdem vom Byte Code Verifier als gültig anerkannt wird. Es ist bereits gelungen, gültigen Byte Code zu schreiben, der von einem Java-Compiler nicht generiert werden kann [LADU_97]. Die Java-Sprachspezifikation und deren Einhaltung sind jedoch von zentraler Bedeutung für das Java-Sicherheitsmodell. Daher ist dieses Problem besonders kritisch, denn es kann zu einer Untergrabung des Java-Sicherheitssystems führen.

- Verteiltes Modell

Eine der Schwächen des Java-Modells ist, daß es sich um ein verteiltes Modell handelt. Alle Sicherheits-Checks werden auf jedem einzelnen ausführenden Rechner durchgeführt. Die TCB wird dabei beliebig groß. Um sicherzustellen, daß ein komplettes Netzwerk gegen Java-Angriffe geschützt ist, muß daher sichergestellt werden, daß jeder einzelne Rechner sicher ist. In der Firewall-Literatur [CHBE_94] wird immer wieder darauf hingewiesen, daß eine Verteidigungsstrategie, bei der jeder einzelne

Netzwerk-Rechner geschützt werden soll, nahezu unmöglich zu kontrollieren und zu administrieren ist. Wie bereits erwähnt, kann bereits ein ungeschützter Rechner dazu führen, daß ein Angreifer Zugriff auf mehrere Komponenten eines Netzwerks gewinnt.

4 Entwicklungen

Das Java-Sicherheitsmodell ist seit seiner Präsentation viel diskutiert worden. Einige der Weiterentwicklungen, die aus diesen Diskussionen hervorgegangen sind, werden in diesem Abschnitt vorgestellt.

4.1 Signed Applets

In den ersten Implementierungen des Java-Sicherheitsmodells in Netscape oder Microsoft Browsern hatte der Benutzer zwei Möglichkeiten:

- Das Ausführen von Applets konnte generell unterbunden werden
- Alle Applets wurden ausgeführt, allerdings mit sehr starken Restriktionen [GRFE_96]

Dabei wurden generell alle Applets, egal aus welcher Quelle oder mit welcher Funktion, gleich behandelt. Wünschenswert ist hingegen, daß es möglich ist, abhängig von der Herkunft des Applets entweder die Ausführung zu unterbinden oder einem Applet weiterreichende Rechte einzuräumen. Als Hauptproblem dabei erweist sich, daß es die im Internet eingesetzten Protokolle (TCP/IP, HTTP) nicht erlauben, die Herkunft eines Applets zweifelsfrei zu bestimmen. Mit dem Java Development Kit (JDK) 1.1 hat Sun deshalb sogenannte „Signed Applets“ eingeführt.[5] Hinter dem Begriff „Signed Applets“ verbergen sich digital signierte Applets (bzw. einzelne Teile eines Applets) [FRMU_96]. Durch eine Prüfung der digitalen Signatur kann auf dem Zielsystem festgestellt werden, von wem dieser Code signiert und ob er seitdem verändert wurde.

Obwohl Signed Applets das Problem der Herkunftsbestimmung technisch lösen, muß diese Technik doch mit Bedacht eingesetzt werden. Eine digitale Signatur gibt lediglich an, wer das Applet unterschrieben hat, sie macht aber keine Aussage darüber, wer das Applet entwickelt/programmiert hat bzw. ob das Applet unerwünschte Nebeneffekte hat oder nicht. Der Benutzer muß daher entscheiden, ob er den Signierer als vertrauenswürdig einstuft.

Bei ActiveX, aber auch beim JDK 1.1 ist die Entscheidung des Benutzers, dem Signierer zu vertrauen, gleichbedeutend damit, daß dem ActiveX-Control bzw. dem Java-Applet unbeschränkter Zugriff auf die Systemressourcen eingeräumt wird. Im HotJava-Browser hingegen kann der Benutzer Applets abhängig von ihrer Herkunft (URL) oder einer digitalen Signatur gezielt detaillierte Zugriffsrechte vergeben.

Neben der technischen Realisierung müssen beim Einsatz von Signed Applets noch eine Reihe von Randbedingungen geklärt werden:

- Eine Public Key-Infrastruktur (PKI) muß die erforderlichen Schlüsselzertifikate bereitstellen.
- Die Frage der Haftung bei digitalen Signaturen muß geklärt werden.
- Es muß festgelegt werden, wer Applets signieren darf (d.h. Firmen, Einzelpersonen oder unabhängige Instanzen).

Es wäre z. B. denkbar, daß in Bereichen mit speziellen Sicherheitsanforderungen Applets von unabhänigen Instanzen (z. B. dem BSI oder anderen Zertifizierungsstellen) geprüft und signiert werden, oder daß Unternehmen Applets nach einer Prüfung in einer Test-Umgebung signieren und damit zur Ausführung im internen Netzwerk freigeben.

- Es muß außerdem geklärt werden, welche Rechte einem als „trusted“ erklärten Applet eingeräumt werden.

Das Ziel sollte sein, einem Applet nur die Rechte zu gewähren, die es zur Ausführung seiner Aufgabe benötigt („need-to-know“-Policy). Diese Anforderung stellt zusätzliche Anforderungen an die Signed Applets bzw. die PKI, da nicht nur Informationen über die Herkunft des Applets benötigt werden, sondern auch anwendungsspezifische Informationen (z. B. welche Zugriffsrechte werden benötigt etc.).

4.2 Protection Domains

Mit der Version 1.2 des JDK (derzeit als Beta-Version erhältlich) führte JavaSoft einige weitreichende Erweiterungen des Sandbox-Modells (Security Manager, Byte Code Verifier, Class Loader) ein [GOMU_98]. Die Funktion dieser Teile bleibt prinzipiell erhalten, nur der Security Manager und der Class Loader wurden leicht modifiziert.

Das Hauptmerkmal der Erweiterungen ist, daß die Sicherheitsregeln nicht programmiert werden müssen, sondern textuell (oder mit dem im JDK 1.2 enthalten Policy Tool) unabhängig von der Implementierung bestimmt werden können. Die Regeln können dabei basierend auf digitalen Signaturen und/oder basierend auf der Herkunft (URL) des Codes sehr detailliert (z. B. verschiedene Zugriffsarten auf einzelne Dateien oder Verzeichnisse) vergeben werden.

Der Byte Code wird ab dem JDK 1.2 in sogenannten Protection Domains ausgeführt. Die Rechte des in einer Protection Domain ausgeführten Codes setzen sich dabei aus allen Rechten zusammen, die an die Herkunfts-URL des Codes vergeben worden sind, sowie den Rechten, die an die Identitäten vergeben wurden, die eine digitale Signatur für das Applet generiert haben (jedes Applet kann dabei von mehreren Identitäten unterschrieben werden). Es gibt im JDK 1.2 auch keine Unterscheidung zwischen Byte Code, der lokal auf dem Rechner gespeichert ist, und solchem, der vom Internet geladen wird.

Jeder ausgeführte Byte Code hat nur genau die Zugriffsrechte der ausführenden Protection Domain. Einzige Ausnahme ist die sogenannte System Domain: Byte Code, der darin ausgeführt wird, darf direkt auf Systemresourcen zugreifen bzw. im Auftrag eines Applets Systemzugriffe ausführen.

Beim Zugriff auf sicherheitskritische Ressourcen wird auch weiterhin der Security Manager konsultiert. Allerdings sind die Zugriffsregeln nicht mehr im Security Manager implementiert, sondern werden vom Security Manager an den sogenannten Access Controller weitergeleitet. Der Access Controller sammelt alle Rechte, die dem Byte Code zugewiesen wurden und entscheidet dann darüber, ob der Zugriff erteilt werden soll oder nicht. Dabei werden nicht die Rechte des unmittelbar aufrufenden Byte Codes zugrunde gelegt, sondern die Rechte des Codes in der Kette der aufrufenden Funktionen, der die geringsten Rechte besitzt („least privilege“-Prinzip). Dadurch soll gewährleistet werden, daß kein Applet, das von Byte Code mit umfassenderen Zugriffsrechten aufgerufen wird, unzulässigerweise auf Systemressourcen zugreifen kann. Das explizite Untersagen von Aktionen für Applets bestimmter Herkunft oder für bestimmte Identitäten ist dagegen nicht möglich.

[5] Eine ähnliche Funktionalität ist auch von Microsoft und Netscape in die Browser eingebaut worden