„Nicht verifizierter Code”: eine Sicherheitslücke in Java

Karsten Sohr

Fachbereich Mathematik und Informatik
Philipps-Universität
D-35032 Marburg
sohr@informatik.uni-marburg.de

Kurzfassung. In diesem Artikel wird eine Sicherheitslücke im JDK 1.1.x und in Java 2 beschrieben, die Anfang März 1999 gefunden worden ist. Ursache für diese Sicherheitslücke ist ein Implementierungsfehler im Bytecode-Verifizierer, einer zentralen Komponente der Java-Sicherheitsarchitektur: Unter gewissen Umständen wird Bytecode nicht mehr vollständig verifiziert, wodurch grundlegende Regeln der Sprache Java wie z.B. die Typsicherheit verletzt werden können. Dies kann zu einem Angriff auf den Netscape Communicator 4.x ausgenutzt werden, mit allen Konsequenzen bis hin zum Löschen wichtiger Daten.

1 Einleitung

Seit ihrer Einführung im Jahre 1995 hat sich die Sprache Java schneller verbreitet als jede andere Sprache zuvor. Ein Grund hierfür besteht vor allem darin, daß sie aufgrund ihrer Plattformunabhängigkeit besonders zur Internet-Programmierung verwendet werden kann. Webseiten werden gerade dadurch für den Anwender interessant, daß sie kleine Programme (sog. *Applets*) enthalten. Java bietet nun die Möglichkeit, solche Programme schnell und bequem zu erstellen. Das Einbinden dieser Applets in Webseiten hat allerdings in bezug auf den Sicherheitsaspekt besondere Konsequenzen. Schließlich werden nun nach dem Aufruf einer Webseite Programme lokal auf dem Rechner ausgeführt, deren Herkunft zumeist unbekannt ist. Hierdurch bieten sich für einen Hacker neue Möglichkeiten, in ein System einzudringen, dort Viren einzuschleusen, geheime Daten auszuspionieren, Dateien zu löschen usw. Den Entwicklern von Java sind die eben genannten Bedenken natürlich nicht verborgen geblieben. Aus diesem Grund haben sie ein spezielles Sicherheitsmodell entworfen, das die Rechte, die einem Applet zustehen, so einschränkt, daß dieses keinen Schaden anrichten kann.

In Abschnitt 2 soll zunächst auf die Grundlagen des Java-Sicherheitsmodelles (im JDK 1.0x bzw. 1.1.x) eingegangen werden, soweit es für die Beschreibung einer Sicherheitslücke und eines daraus resultierenden Angriffs, der im März 1999 an der

Universität Marburg erfolgreich durchgeführt worden ist (s. [10]), erforderlich ist[1]. Dieser wird dann in Abschnitt 3 näher geschildert. Dabei wird gezeigt, daß das Sicherheitsmodell durchaus noch immer verwundbar ist, auch wenn es nicht einfach ist, einen solchen Angriff durchzuführen. Ferner kann man am Ablauf dieses Angriffs erkennen, daß die Folgen eines kleinen Implementierungsfehlers in der Sicherheitsarchitektur immens sein können.

2 Das Sicherheitsmodell von Java

Das Java-Sicherheitsmodell wird in seiner ursprünglichen Fassung im JDK 1.0.x (s. [11]) oft auch als *Sandboxmodell* bezeichnet. Dieser Begriff kommt daher, daß einem Applet nur eingeschränkte Rechte zugebilligt werden, d.h. sein Wirkungsbereich ist begrenzt („Sandkasten"). So ist es einem Applet beispielsweise nicht erlaubt, Dateien zu lesen (sonst könnte es beliebige Daten ausspionieren), zu löschen oder zu schreiben. Auch darf ein Applet nicht beliebige Netzverbindungen eröffnen; andernfalls könnte ein bösartiger Hacker eine Firewall (z.B. eines Intranets einer Firma) umgehen und somit Zugriff auf wichtige Firmengeheimnisse erhalten.

Das Sandboxmodell setzt sich aus vier Komponenten zusammen: Einerseits gehört dazu das Sprachdesign von Java, andererseits die Komponenten Verifizierer (*verifier*), Klassenlader (*class loader*) und Sicherheitsmanager (*security manager*). Nachfolgend sollen die einzelnen Komponenten näher erläutert werden. Wert gelegt wird dabei vor allem auf die Beschreibung der Bytecode-Verifikation, da dies für das Verständnis des Angriffs von zentraler Bedeutung ist.

2.1 Die Sprache Java

Die erste Komponente ist die Sprache Java selbst. Java gilt im Gegensatz zu C/C++ als eine sichere Sprache, da sie u.a.

- keine Zeigerarithmetik zuläßt,
- stark getypt ist und keine willkürlichen Typkonvertierungen (*type casts*) erlaubt,
- Arraygrenzenüberprüfungen (*array bounds checking*) vornimmt,
- einen Speicherbereiniger (*garbage collector*) besitzt, der selbständig den nicht mehr benötigten dynamisch allokierten Speicher wieder freigibt.

Da Java eine stark getypte Sprache ist, ist beispielsweise folgendes Programm nicht erlaubt und muß somit von einem Java-Compiler zurückgewiesen werden:

---

[1] Es wird allerdings auf eine detailliertere Beschreibung der Sicherheitsarchitektur von Java 2 (s. [5]) verzichtet, da Java 2 noch in keinem kommerziell vertriebenen Webbrowser eingebaut worden ist. Nichtsdestotrotz ist die Sicherheitslücke auch in Java 2 vorhanden und kann offenbar auch dementsprechend ausgenutzt werden (s. [10]).

```
int string2int(){
          String s = "I'm an integer!";
          return s; // Typfehler: nicht erlaubt
}
```

Offenbar liegt hier ein Typkonflikt vor, weil ein String zurückgegeben wird, obwohl das Java-System eigentlich einen Integer als Rückgabewert erwartet. Wären in Java solche Typkonflikte möglich, dann könnte es zu Inkonsistenzen kommen, die die gesamten Sicherheitsmechanismen von Java außer Kraft setzen könnten (s. Abschnitt 3.2).

2.2 Der Verifizierer

Üblicherweise werden Klassendateien (*class files*) durch Übersetzung von Java-Programmen erzeugt. Allerdings schreibt die Spezifikation der JVM (Java Virtual Machine) dies nicht unbedingt vor und läßt damit bewußt gewisse Freiheiten. Es gibt bereits Compiler, die Bytecode aus Ada- und C-Programmen erzeugen können. Darüber hinaus kann nicht einmal garantiert werden, daß die zur Erzeugung von Klassendateien verwendeten Compiler vertrauenswürdig sind. Man kann sogar Klassendateien direkt von Hand erstellen. Hierzu eignet sich z.B. der Bytecode-Assembler *Jasmin* (s. [8]), der weiter unten noch näher beschrieben wird.

Da die JVM nicht ohne weiteres erkennen kann, wer der Urheber der auszuführenden Klassendatei ist, ist in die JVM der sog. *Verifizierer*[2] als zusätzlicher Sicherheitsmechanismus eingebaut worden (s. [6]). Dieser überprüft, ob die wichtigsten Regeln der Sprache Java (wie z.B. die Typsicherheit, die richtige und vollständige Initialisierung von Variablen) auch in der Klassendatei eingehalten werden. Der Verifizierer nimmt sowohl Laufzeittests (dynamische Tests) als auch Tests beim Linken (statische Tests) vor. Im Prinzip wäre es vom Standpunkt des Implementierungsaufwandes am einfachsten, wenn alle Tests nur zur Laufzeit durchgeführt werden, da die entsprechende Information dort direkt zur Verfügung steht. Andererseits wäre hiermit ein deutlicher Effizienzverlust verbunden, da die entsprechenden Überprüfungen vor dem Abarbeiten *jeder* Bytecode-Instruktion immer wieder neu vorgenommen werden müßten. Wenn die benötigte Information direkt in der Klassendatei vorhanden ist bzw. daraus hergeleitet werden kann, können die Tests schon beim Linken vorgenommen werden, bevor das Programm ausgeführt wird. Insbesondere kann der Verifizierer bereits beim Linken u.a. sicherstellen, daß

1. die Klassendatei das korrekte Format besitzt (z.B. 0xCAFEBABE am Anfang),

---

[2] Der Begriff *Verifizierer* sollte nicht mit dem gleichlautenden Begriff aus der theoretischen Informatik verwechselt werden: Es handelt sich nicht um den *formalen Beweis* gewisser Eigenschaften, die das Bytecode-Programm zu erfüllen hat, sondern lediglich um *ad hoc*-Tests, die informell in [6] spezifiziert worden sind. Im vorliegenden Artikel ist der Begriff Verifikation meistens in diesem informellen Sinne zu verstehen.

2. jede Klasse eine Vaterklasse besitzt (mit Ausnahme von **Object**),
3. die Methoden mit den geeigneten Parametern aufgerufen werden,
4. Feldern nur Werte zugewiesen werden, die die korrekten Typen besitzen,
5. Variablen vor ihrer Verwendung initialisiert worden sind,
6. keine Überläufe und Unterläufe des Operandenstacks vorkommen.

Erkennt der Verifizierer, daß eine der obigen Bedingungen nicht eingehalten wird, dann wird der Lade- bzw. Linkvorgang der Klassendatei abgebrochen, und die JVM löst eine Ausnahme (*exception*) des Typs **VerifyError** aus. Während die ersten beiden Bedingungen einfach anhand der Klassendatei überprüft werden können, ist dies für die Bedingungen 3.-6. nicht so leicht möglich. Hierfür ist eine Daten- und Kontrollflußanalyse (s. [1]) erforderlich.

Im folgenden wird ein Beispielprogramm angegeben, das der Verifizierer beim Linkvorgang zurückweisen müßte. Es handelt sich hierbei um das Bytecodeäquivalent zum oben angegebenen inkorrekten Java-Programm (s. Abschnitt 2.1):

```
Method int string2int()
    0 ldc #14 <String "I'm an integer!">// push String
    2 ireturn
```

Ruft man im JDK 1.1 den Java-Interpreter mit der Option **–verify** auf, so erhält man in diesem Fall als Fehlermeldung: *„Expecting to find integer on stack!“*.

Darüber hinaus können nicht alle Tests beim Linken der Klassendatei durchgeführt werden. Außerdem werden andere Tests aus Gründen der Implementierung auf die Laufzeit verschoben, obwohl man sie beim Linken hätte durchführen können (s. [6]).

2.3 Der Klassenlader

Die nächste Komponente der Java-Sicherheitsarchitektur ist der *Klassenlader*. Seine Aufgabe besteht darin, Klassen über das Netz zu laden und dabei gleichzeitig zu verhindern, daß es zu Namenskonflikten und damit auch zu Typkonflikten kommt. Insbesondere darf ein Applet keine Systemklassen wie z.B. **FileInputStream** oder **SecurityManager** durch eine eigene Definition überschreiben. Auch darf es nicht zu Namenskonflikten zwischen Klassen verschiedener Applets kommen.

Jedes Applet besitzt seinen eigenen Klassenlader (**AppletClassLoader**), der die dazugehörigen Klassen in einem getrennten Namensraum installiert. Die Systemklassen besitzen einen besonderen Klassenlader und werden vom System installiert (Nullklassenlader). Auch hierfür gibt es einen separaten Namensraum. Eine Klasse wird nun nicht mehr nur durch ihren Namen, sondern zusätzlich durch ihren Klassenlader eindeutig bestimmt, d.h. es wird das Paar *(Klassenname, Klassenlader)* zur eindeutigen Identifizierung einer Klasse herangezogen.

Der Appletklassenlader ist im Gegensatz zum Verifizierer nicht direkt in die JVM integriert, sondern es handelt sich um eine Instanz einer Subklasse von **ClassLoader**.

2.4 Der Sicherheitsmanager

Wie bereits oben erwähnt, gelten für ein Applet im Sandboxmodell bestimmte Einschränkungen: Ein Applet darf gewisse gefährliche Operationen nicht durchführen. So ist es einem Applet u.a. nicht erlaubt,

- Dateien zu lesen, zu schreiben, zu löschen und zu verändern,
- beliebige Netzverbindungen herzustellen,
- beliebige Systemkommandos und –prozesse auszuführen.

Um dies zu gewährleisten, gibt es im Sicherheitsmodell von Java den sog. *Sicherheitsmanager*, der vor jeder Ausführung einer gefährlichen Operation überprüft, ob diese erlaubt ist. Ist dies nicht der Fall, so wird eine Ausnahme vom Typ **SecurityException** ausgelöst. Versucht also ein Applet eine Datei des lokalen Dateisystems zu löschen, so wird dies dadurch verhindert, daß der Sicherheitsmanager den Vorgang abbricht, wobei eine Ausnahme erzeugt wird.

Jeder Browser-Hersteller kann prinzipiell seine eigene Sicherheitsstrategie für Applets festlegen. Hierfür stellt die Java-Klassenbibliothek die abstrakte Klasse **SecurityManager** zur Verfügung, die der Browser-Hersteller dann noch gemäß seiner Sicherheitsstrategie implementieren muß. Der Sicherheitsmanager ist somit immer eine Instanz von **SecurityManager**.

2.5 Signierte Applets

In der oben beschriebenen Form erwies sich das Sandboxmodell als zu wenig flexibel; denn es waren hierdurch nicht einmal Standardanwendungen wie z.B. ein einfacher Texteditor als Applet möglich, da man hierzu die entsprechenden Schreib- bzw. Leserechte auf die Dateien benötigt. Aus diesem Grund hat SUN das Sicherheitsmodell um kryptographische Methoden vom JDK 1.1 an erweitert. Applets können jetzt mit einer *digitalen Signatur* versehen werden. Digitale Signaturen weisen ähnliche Eigenschaften wie Unterschriften auf, mit denen man Textdokumente unterzeichnet (z.B. die Möglichkeit einer eindeutigen Identifizierung des Unterzeichners). Mit Hilfe der digitalen Signatur kann der Anwender entscheiden, ob er dem Applet alle Zugriffsrechte gewährt oder nicht, je nachdem, ob er dem Unterzeichner vertraut oder nicht. Es handelt sich somit um ein Schwarz-Weiß-Modell, bei dem ein Applet entweder alle oder aber nur die Rechte der Sandbox besitzt.

In Java 2 und auch im Netscape Communicator 4.x ist dieses Modell noch einmal dahingehend erweitert worden, daß auch Zugriffsrechte feinerer Granularität eingerichtet werden können. So kann man z.B. einem Applet nur Leserechte (aber keine Schreibrechte) auf Dateien geben. Die Entscheidung, welche Rechte eingeräumt werden sollen, kann der Anwender davon abhängig machen, welche Funktionalität das Applet haben soll und wer dieses signiert hat. Zur Implementierung dieser Erweiterung stellt beispielsweise Netscape die sog. *Capability Classes* – ein API im Package **netscape.security** - zur Verfügung (s. [9]). Dabei konsultiert der Sicher-

heitsmanager einen Privilegmanager, der mit Hilfe einer Zugriffsmatrix bestimmt, *welches* Applet *welche* Zugriffsrechte besitzt.

3 Der Angriff auf das Java-Sicherheitssystem

Nachfolgend soll das Prinzip eines erfolgreich durchgeführten Angriffs auf das Java-Sicherheitssystem (s. [12]) beschrieben werden. Hierbei handelt sich jedoch nicht um den ersten Angriff dieser Art. In den letzten drei Jahren sind knapp 20 ähnliche Angriffe vorgenommen worden, wobei die meisten davon von Forschern der Universität Princeton (s. [2], [7]) stammen. Die Konsequenz der meisten dieser Angriffe war die völlige Kontrolle über den attackierten Rechner bis hin zum Löschen wichtiger Daten, Ausführen beliebiger Betriebssystemkommandos etc. Während in der ersten Zeit nach der Einführung Javas fast monatlich eine neue schwerwiegende Sicherheitslücke gefunden wurde, sind in letzter Zeit erfolgreiche Angriffe immer seltener geworden. So stammt die letzte von SUN berichtete schwerwiegende Sicherheitslücke aus dem Juli 1998. Dies verleitete viele zu der Annahme, Java sei nun wirklich sicher, alle Fehler seien behoben worden. Das neuerdings gefundene Sicherheitsproblem beweist offenbar das Gegenteil: 100prozentige Sicherheit gibt es noch immer nicht und wird es wohl auch in der Zukunft nicht geben.

Der nun zu beschreibende Angriff nutzt einen Implementierungsfehler im Bytecode-Verifizierer aus. Dabei wird offenbar, daß eine kleine Lücke in einer der Komponenten des Sicherheitsmodelles zur völligen Lahmlegung des gesamten Sicherheitssystems von Java führen kann. Daß das gesamte Sicherheitsmodell funktioniert, kann nur dann gewährleistet werden, wenn alle Komponenten[3] korrekt arbeiten.

Der Angriff setzt sich aus zwei Teilen zusammen:

1. Erzeugung einer Klassendatei, die grundlegende Sicherheitsregeln der Sprache Java verletzt, aber trotzdem vom Verifizierer akzeptiert wird,
2. Ausnutzen dieser Sicherheitslücke zum Angriff auf den Netscape Communicator 4.x.

3.1 Der Fehler im Bytecode-Verifizierer

Ein wichtiges Hilfsmittel zur Aufdeckung des Verifizierer-Fehlers, der später in diesem Abschnitt näher beschrieben wird, war der Bytecode-Assembler *Jasmin* (s. [8]). Aus diesem Grund soll zunächst auf diesen eingegangen werden. Wie oben bereits erwähnt, kann man mit Hilfe von Jasmin bequem Klassendateien erzeugen, die keinem äquivalenten Java-Programm im Sinne der Java-Sprachspezifikation

[3] Aus diesem Grund sollte man auch nicht von vier Ebenen (Stufen) des Sicherheitsmodelles sprechen, weil dies fälschlicherweise suggeriert, daß beim Ausfall einer Stufe noch die anderen Stufen alles wieder in Ordnung bringen könnten.

entsprechen und mithin grundlegende Regeln der Sprache Java verletzen. Jasmin wurde im Jahre 1996 von Jonathan Meyer entwickelt und ist ein Bytecode-Assembler mit einer Syntax, die der des Programmaufrufes von **javap** mit der Option **-c** ähnlich ist. Allerdings wurde diese Syntax noch um diverse Assembler-Direktiven (Anweisungen, die mit dem Zeichen „.“ beginnen) zur Darstellung von Meta-Level-Informationen erweitert. So gibt es z.B. die Direktiven **.method** (Deklaration einer Methode) und **.class** (Beginn einer Klassendeklaration).

Zur Illustration wird hier nun die Jasmin-Variante der (unkorrekten) Methode **string2int()** aus Abschnitt 2.1 angegeben:

```
.class Test
.super java.lang.object
.method string2int()I
.stack 2
ldc "I'm an integer!"
ireturn
.end method
```

Jasmin erzeugt aus dem eben angegebenen Code eine Klassendatei **Test.class**, wobei – wie schon angedeutet – nicht überprüft wird, ob sie den wichtigsten Regeln der Sprache Java genügt. Dies ist bekanntlich die Aufgabe des Verifizierers der JVM!

Man betrachte nun folgendes Bytecode-Programm (in **javap**-Syntax), das mit Jasmin erzeugt worden ist und keinem korrekten Java-Programm entspricht:

```
Method int string2int()
    0  aconst_null
    1  goto 10
    4  pop
    5  ldc #14 <String "I'm an integer!">
    7  goto 11
    10 athrow
    11 ireturn
Exception Table:
from      to     target     type
10        12     4          <Class java.lang.Exception>
```

Laut Deklaration müßte die Methode **string2int()** einen Integer-Wert zurückliefern. Betrachtet man aber den Programmablauf, so erkennt man, daß in Wirklichkeit ein String zurückgegeben wird; denn es werden die Bytecode-Instruktionen in der Reihenfolge 0-1-10-4-5-7-11 abgearbeitet. Es handelt sich demnach um einen klassischen Typkonflikt. Ein Bytecode-Verifizierer müßte obiges Programm aufgrund des Typkonfliktes als unkorrekt zurückweisen, und zwar während der Link-Phase bei der Datenflußanalyse (s. Abschnitt 2.2).

Nun akzeptieren aber die Verifizierer des JDK 1.1.6, von Java 2 und des Netscape Communicators 4.x (x < 6) obiges Bytecode-Programm ohne Beanstandung. Weitere Untersuchungen ergaben, daß zwischen den beiden **goto**-Anweisungen jede

beliebige Bytecode-Sequenz eingefügt werden kann, die eigentlich vom Verifizierer während der Phase der Datenflußanalyse als unzulässig erkannt werden müßte. Hierzu gehören u.a.

- das Verwenden von nicht initialisierten Variablen,
- die Erzeugung beliebiger Typkonflikte,
- die Erzeugung von Operandenstacküberläufen und –unterläufen und
- der Zugriff auf nicht erlaubte Register (*locals*).

Es sieht demnach so aus, als ob der Verifizierer irrtümlicherweise den Code zwischen den beiden **goto**-Anweisungen für toten Code hält und somit auf eine weitere Verifikation verzichtet. Bei der Implementierung des Verifizierers ist offenbar ein Sonderfall in der Spezifikation der JVM (s. [6]) übersehen worden, daß nämlich die Obergrenze des durch einen Ausnahmebehandler (*exception handler*) geschützten Bereiches gleich der Codelänge sein darf (im obigen Beispiel haben die Codelänge und die Obergrenze **to** jeweils den Wert 12). In [6] findet man dazu die folgende Festlegung, wobei **end_pc** die Obergrenze des geschützten Bereiches ist: *„The value of end_pc either must be a valid index into the code array of the opcode of an instruction or must be equal to code_length, the length of the code array."* Eine Obergrenze, die der Codelänge entspricht, ist laut Spezifikation demnach zugelassen.

Insgesamt zeigt sich an dieser, aber auch an einigen anderen in den letzten Jahren gefundenen schwerwiegenden Sicherheitslücken, daß der Prozeß der Bytecode-Verifikation fehleranfällig ist. So wurde der Klassenladerangriff der Universität Princeton vom März 1996 erst durch einen Fehler in der Bytecode-Verifikation ermöglicht (s. [2, 3]). Auch die Ergebnisse des Projektes *Kimera* der Universität Washington deuten in diese Richtung (s. [7]). Offenbar steckt bei der Bytecode-Verifikation der Teufel im Detail: Die grundlegende Strategie ist zwar korrekt implementiert, es gibt jedoch immer wieder kleine Lücken. Aber genau hierin liegt die Problematik im Java-Sicherheitsmodell (in allen Versionen): Aufgrund der Abhängigkeiten der einzelnen Komponenten kann ein kleiner Fehler zur völligen Außerkraftsetzung aller Sicherheitsmechanismen führen. Dies soll unten noch weiter verfolgt werden.

3.2 Durchführung eines Angriffs

Nachfolgend soll anhand eines *fiktiven* Beispieles näher erläutert werden, was passieren kann, wenn keine Typsicherheit durch die JVM garantiert wird. Angenommen, man hat zwei Klassen **SecurityManager** und **SpoofSecurityManager** (s. Abbildung 1). In beiden Klassen ist dabei das erste Datenfeld (**FileAccessAllowed**) jeweils vom Typ **boolean**. Das erste Datenfeld von **SecurityManager** ist privat (*private*), während das Gegenstück von **SpoofSecurityManager** öffentlich (*public*) ist. Man beachte ferner, daß die JVM für Objekte ein ähnliches Speicherlayout wie die Sprache C verwendet, bei dem alle Datenfelder der Reihe nach hintereinander angeordnet sind.

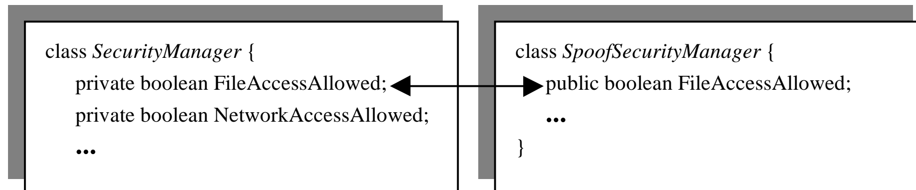

Abbildung 1. Die Klassen *SecurityManager* und *SpoofSecurityManager* besitzen beide als erstes Datenfeld ein Boolesches Datenfeld. Bei einem Typkonflikt kann man dies ausnutzen, um auf die private Membervariable *FileAccessAllowed* auch außerhalb der Klasse *SecurityManager* zugreifen zu können.

Wenn die JVM beliebige Typkonflikte zuläßt (bei dem in Abschnitt 3.1 beschriebenen Fehler im Verifizierer ist dies der Fall), kann man z.B. ein **SecurityManager-**Objekt einer Variablen vom Typ **SpoofSecurityManager** zuweisen. Somit kann die Membervariable **FileAccessAllowed** von **SecurityManager** auf den Wert **true** gesetzt werden, obwohl diese eigentlich privat ist: Die JVM nimmt nämlich an, daß es sich um das erste Datenfeld von **SpoofSecurityManager** handelt und erlaubt mithin die Zuweisung.

Nachdem das Datenfeld **FileAccessAllowed** des Sicherheitsmanagers auf den Wert **true** gesetzt worden ist, erlaubt der Sicherheitsmanager (in unserem fiktiven Beispiel) beliebige Dateizugriffe, ohne daß einem Applet explizit entsprechende Rechte zugebilligt worden sind. Das folgende Codefragment soll noch einmal das Schema zeigen, wie ein Angriffsapplet aussehen könnte:

```
class AttackApplet extends Applet{
    public void init(){
          SecurityManager SM;
          SpoofSecurityManager SSM;
          // hole den Sicherheitsmanager des Webbrowsers
          SM  = System.getSecurityManager();
          SSM = SM; // Typkonflikt
          SSM.FileAccessAllowed = true;
          // => SM.FileAccessAllowed ist true
          . . .
    }
    public void run(){
    // führe Angriff durch: z.B. Dateien löschen
    // Sicherheitsmanager erlaubt alle Dateizugriffe
    }
}
```

Zu beachten ist in diesem Zusammenhang, daß der Aufruf von **System.getSecurityManager()** den Sicherheitsmanager des Webbrowsers (s. Ab-

schnitt 2.4) liefert. Anschließend wird also der Sicherheitsmanager des Webbrowsers manipuliert. Außerdem muß die Zuweisung **SSM = SM** in einer manuell erzeugten Klassendatei (nach dem Schema aus Abschnitt 3.1) vorgenommen werden, da ein korrekter Java-Compiler obiges Programm wegen des Typkonfliktes zurückweisen muß. Beispielsweise könnte man sich mit Hilfe des Bytecode-Assemblers Jasmin eine Klassendatei erzeugen, die eine Methode **SM2SSM()** enthält und in der dann die entsprechende Zuweisung vorgenommen wird.

Neben dem Sicherheitsmanager ist auch der Klassenlader eine Instanz einer Java-Klasse. Somit wird klar, warum die Typsicherheit Voraussetzung dafür ist, daß das Java-Sicherheitssystem funktioniert. Da auch das Netscape-Modell intensiven Gebrauch von Java-Klassen macht (*Capability Classes*), gelten für den Netscape Communicator ähnliche Bemerkungen. Darüber hinaus wird an dieser Stelle offenbar, daß der Ausfall einer Komponente (in unserem Fall ist dies der Verifizierer) das gesamte Sicherheitssystem kompromittiert.

Weiter oben wurde ein fiktives Beispiel behandelt; der eigentlich erzeugte Typkonflikt, der für das Angriffsapplet ausgenutzt wurde, kann in diesem Rahmen aus Gründen der Sicherheit nicht beschrieben werden. Letztendlich zeigt aber obiges Beispiel das Prinzip, nach dem man Typkonflikte zum Erzeugen eines vollständigen Angriffs ausnutzen kann. Der eigentliche Angriff verlief analog. Es gibt allerdings noch weitere Möglichkeiten, Typkonflikte entsprechend auszunutzen. Eine davon besteht darin, Objekte für Integer-Werte zu halten und hierdurch Speicheradressen herauszufinden. In diese Speicherstellen kann man dann seinen eigenen Maschinencode plazieren, den man dann mit Hilfe eines Tricks direkt ausführen kann (s. [3]).

3.3 Die Folgen und die Reaktion von SUN und Netscape

SUN und Netscape haben den Fehler innerhalb kurzer Zeit bestätigt und die Ursache für diesen herausgefunden. Angeblich handelt es sich um einen Implementierungsfehler, der durch Hinzufügen einer einzelnen Programmzeile beseitigt werden konnte. Wir haben überprüft, daß der Fehler im JDK 1.1.8 und in Java 2 (SDK v1.2.1) von SUN nicht mehr auftritt. Auch in der JVM des Netscape Communicators 4.6 ist der Fehler behoben worden. Klar ist weiterhin, daß der Fehler nicht das gesamte Sicherheitskonzept von Java gefährdet.

4 Zusammenfassung und Ausblick

In diesem Artikel ist eine schwerwiegende Sicherheitslücke des Java-Sicherheitsmodelles vom JDK 1.1 und Java 2 beschrieben worden, die zum Erzeugen eines Angriffsapplets im Netscape Communicator 4.x ausgenutzt werden kann. Die Ursache für die Sicherheitslücke lag in der Implementierung des Verifizierers, wodurch keine Typsicherheit mehr garantiert werden konnte. Die Typsicherheit ist aber die Grundlage des Java-Sicherheitsmodelles, zumal integrale Teile der Java-Sicherheitsarchitektur (in *allen* Versionen) auf Java-Klassen beruhen.

Auch wenn es sich bei dem hier behandelten Fehler offensichtlich um einen relativ leicht zu behebenden Implementierungsfehler handelt, so sind auch in der Zukunft ähnliche Fehler nicht auszuschließen, da der Vorgang der Bytecode-Verifikation komplex und mithin fehleranfällig ist. Dies liegt vor allem daran, daß das Bytecode-Format eine lineare Programmrepräsentation ist. Eigenschaften wie die Typsicherheit müssen daraus mittels einer Daten- und Kontrollflußanalyse wiedergewonnen werden. Wenn man dagegen abstrakte Syntaxbäume als Zwischenformat gewählt hätte (die üblicherweise im Frontend eines Compilers erzeugt werden), dann wäre die Verifikation gewisser Eigenschaften wie z.B. der Typsicherheit einfacher und mithin weniger fehleranfällig. Denn dort sind die Programmeigenschaften des ursprünglichen Quellprogrammes direkt enthalten. Es gibt bereits das System *Juice*, das wie Java das Einbinden von Applets in Webseiten unterstützt, im Gegensatz zu Java aber komprimierte abstrakte Syntaxbäume als Verteilungsformat verwendet (s. [4]). Eine Übertragung dieses Ansatzes auf das Java-System wäre eine interessante Alternative zum Bytecode-Format.

5 Literatur

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading, 1988
2. Dean, D., Felten, E.W., Balfanz, D., Wallach, D.S.: Java Security: Web Browsers and Beyond. In Denning, D.E., Denning, P.J. (Herausgeber): Internet Beseiged: Countering Cyberspace Scofflaws. ACM Press, New York, 1997
3. Dean, D.: Formal Aspects of Mobile Code Security. Dissertation. Princeton, 1999
4. Franz, M., Kistler, Th.: Juice-Homepage. http://www.ics.uci.edu/~juice, 1997
5. Gong, L., Mueller, M., Prafullchandra, H., Schemers, R.: Going Beyond the Sandbox: An Overview of the New Security Architecture in the JDK 1.2. In: Proceedings of the USENIX Symposium on Internet Technologies and Systems, Monterey, 1997
6. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Addison-Wesley, Reading, 1997
7. McGraw, G., Felten, E.W.: Securing Java: Getting Down to Business with Mobile Code. Wiley, New York, 1999
8. Meyer, J.: Jasmin-Homepage. http://mrl.nyu.edu/meyer/jvm/jasmin.html, 1996
9. Netscape Communications Corporation: The Java Capabilities API. http://developer.netscape.com/docs/manuals/signedobj/capsapi.html, 1999
10. Secure Internet Programming: History. http://www.cs.princeton.edu/sip/history/, 1999
11. Sun Microsystems: HotJavaTM: The Security Story. http://java.sun.com/sfaq/may95/security.html, 1995
12. Sun Microsystems: SUN Sets to Deliver Software Fix for Java Development Kit Security Bug. http://java.sun.com/pr/1999/03/pr990329-01.html, 1999