Informatik B - Objektorientierte Programmierung in Java

Vorlesung 19:
Sicherheit

Inhalt

Sicherheitsmechanismen in Java

Literatur: Guido Krüger: „Handbuch der Java-Programmierung“, www.javabuch.de: Kapitel 47

Sicherheitsmechanismen in Java

- Sprachsicherheit
  - Java wurde mit hohen Sicherheitsansprüchen entworfen, da Applets aus dem unsicheren Internet geladen werden können.
  - Bösartige Applets sollen keine Angriffe auf den lokalen Computer oder die Ressourcen des Anwenders ausführen können.

Sicherheitsmechanismen in Java

- Sprachsicherheit
  - Javas Speichermanagement arbeitet automatisch, d.h. es hat keine Sicherheitslücken durch provozierte Speicherüberläufe.
  - Typkonvertierungen, Arrayzugriffe und Strings werden zur Laufzeit geprüft. Zugriffe, die außerhalb des erlaubten Bereichs liegen, führen nicht zu undefiniertem Verhalten.
  - Bytecode, der über das Netz geladen wird, wird vor der Ausführung untersucht.

Sicherheitsmechanismen in Java

- Sicherheitskonzepte für mobile Programme
  - Damit der Benutzer von mobilen Programmen (Java-Applets) vor unerwünschten Aktionen (wie z.B das Ausspionieren von persönlichen Daten, Veränderungen/Manipulation von Dateien auf der Festplatte) geschützt ist, verfügt Java über entsprechende Sicherheitskonzepte:
  - Das Sandbox-System
  - Die Code-Signatur

Das Sandbox-System von Java

Java-Applets verfügen über eine sehr strenge Sicherheitsarchitektur. Sie werden durch die Java Virtual Machine vom Rest des Systems strikt abgeschottet und befinden sich in einer eigenen Umgebung, der sogenannten Sandbox. Den Mini-Applikationen ist dadurch **jedweder** Zugriff auf das lokale System verboten, weder können sie Dateien lesen oder schreiben, noch auf Hardwareressourcen zugreifen. Der einzige Kommunikationsweg „nach draußen“ führt zurück zu dem Server, von dem sie geladen wurden. Dadurch soll verhindert werden, dass Dateien manipuliert, Viren eingeschleust oder Informationen ausgespäht werden. Idee: Untrusted Code läuft in vertrauenswürdiger Umgebung.

Sicherheit von Applets

- Der Browser stellt in seiner Laufzeitumgebung Methoden zur Verfügung, um bösartige Wirkungen des Applet-Codes zu verhindern.
- Ein (über das Netz geladenes) Applet kann deshalb
  - nur zu demjenigen Rechner eine Netzwerk-Verbindung aufbauen, von dem es geladen wurde,
  - kann keine lokalen Dateien lesen oder schreiben,
  - kann keine externen Programme starten,
  - kann keine nativen Routinen aufrufen,
  - darf nur sehr beschränkt Systeminformationen auslesen.

Die Code-Signatur bei Java

Durch die Signatur von mobilen Programmen soll sich deren Absender genau identifizieren lassen. Trust-Center können digitale Signaturen und Schlüssel mit Zertifikaten versehen und dem Empfänger eines mobilen Programms eine vorab erfolgte Identitätsprüfung des Absenders testieren. Die Code-Signatur steht bei Java-Applets ab der Version 1.1.x als Alternative oder Ergänzung zum Sandbox-System zur Verfügung. Applets, denen der Zugriff auf lokale Systemressourcen gestattet werden soll, können durch Code-Signatur freigegeben werden. Dies geschieht allerdings nicht automatisch, sondern der Freigabe muss vom Benutzer **explizit** zugestimmt werden.

Sicherheit: trusted Applets

- JVM im Browser unterstützt ein differenziertes Sicherheitskonzept.
- Applets können mehr Rechte eingeräumt werden.
- Ebenso können signierte Applets ("trusted Applets") mehr Rechte bekommen.
  Zum Signieren wird z.B. **`keytool`** sowohl zur digitalen Unterschrift als auch zur Überprüfung derselben verwendet.
- Signiert werden die **`jar`**-Dateien, in denen das Applet komprimiert gespeichert ist: **`jarsigner`**

Signierte Applets

Im folgenden:

- Entwicklung eines Applets, das auf beschränkte Ressourcen zugreift
- Erzeugung einer Signatur
- Weitergabe von Zertifikaten an Applets
- Anpassung der Sicherheitseinstellungen, damit das Applet auf die beschränkten Ressourcen zugreifen kann

- Exkurs: Kryptografische Grundlagen

Ein „unerlaubtes“ Applet

- Beispiel: TrustedApplet kann gestartet werden mit **`TrustedApplet.html`**.
- TrustedApplet versucht, eine Datei zu erzeugen: **`C:\tmp\applets\TrustedApplet.log`**.
- Es versucht, das aktuelle Datum mit der aktuellen Zeit in die Datei zu schreiben.
- Es versucht, System-Properties zu lesen: **`user.name`**, **`user.home`**, **`user.dir`**
- Es gibt abschließend eine Meldung aus, ob alle Sicherheitshürden überwunden wurden.

Ein „unerlaubtes“ Applet

- Hinweis: Das Verzeichnis **`C:\tmp\applets`** muss vorhanden sein, sonst gibt es eine **`IOException`**, wenn Zugriff erlaubt.

- Starten des Applets im Browser oder mit **`appletviewer TrustedApplet.html`** scheitert schon am Erzeugen der Datei:

```
java.security.AccessControlException: access denied (java.io.FilePermission
```

Signieren des Applets - Vorbereitung

- Zum Signieren ist es zunächst erforderlich, alle für den Betrieb des Applets nötigen Dateien in ein jar-File zu packen. Signiert wird also nicht eine einzelne .class-Datei, sondern alle Dateien innerhalb des jar-Archivs.
- **`jar -cvf trapp.jar TrustedApplet.class`**

  erzeugt die Datei **`trapp.jar`**, die die Klassendatei **`TrustedApplet.class`** enthält.

  `jar` wird in der Eingabeaufforderung ausgeführt:

```
C:\tmp>jar cvf trapp.jar TrustedApplet.class
Manifest wurde hinzugef³gt.
Hinzuf³gen von: TrustedApplet.class(ein = 1826) (aus= 1057)(komprimiert 42 %)
```


Applets in jar-Archiven in HTML

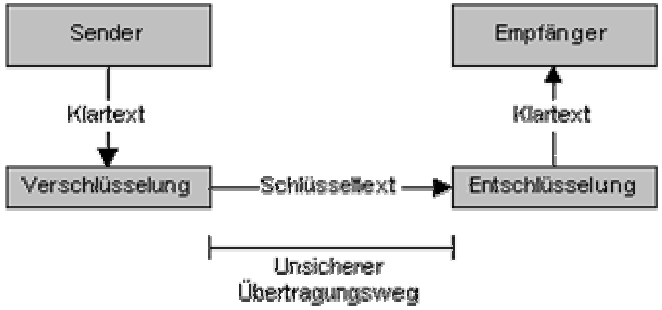
- Im **`Applet`**-Tag von HTML dient **`ARCHIVE=`**... dazu, die benötigte(n) **`jar`**-Datei(en) zu laden.
- Beispiel: TrustedAppletJAR.html:

```
 ...
<h1>TrustedAppletJAR Demo</h1>

<applet
  archive="trapp.jar"
  code="TrustedApplet.class"
  width=600
  height=200
>
...
```


Kryptografie - Grundlagen

- Signaturen basieren auf dem Prinzip der Verschlüsselung mit dem Public-Key-Verfahren
- Verschlüsseln einer Nachricht:



Public-Key-Verfahren

- Idee: Verschlüsselung und Entschlüsselung sind nicht symmetrisch.
- Der Sender verschlüsselt die Nachricht mit dem „öffentlichen“ Schlüssel des Empfängers, der sie mit seinem geheimen „privaten“ Schlüssel entschlüsselt.
- Die Verschlüsselung basiert also darauf, dass der Empfänger allen Sendern öffentlich den Schlüssel zum Verschlüsseln mitteilt (z.B. auf seiner Web-Seite, in einer E-Mail oder in der Zeitung), aber nur er kann mit seinem geheimen Schlüssel die Nachricht entschlüsseln.

Signaturen = Digitale Unterschriften

- Ein großer Vorteil der Public-Key-Kryptosysteme ist es, dass sie Möglichkeiten zum Erstellen und Verifizieren von *digitalen Unterschriften* bieten. Eine digitale Unterschrift besitzt folgende wichtige Eigenschaften:
- Authentizität
  - Sie stellt sicher, dass eine Nachricht von einem ganz bestimmten und eindeutig identifizierbaren Absender stammt.
- Integrität
  - Sie stellt sicher, dass die Nachricht intakt ist und nicht während der Übertragung verändert wurde.

Digitale Unterschrift - Funktionsweise

- Will A eine Nachricht signieren, so verschlüsselt er sie mit seinem privaten Schlüssel. Jeder, der im Besitz des öffentlichen Schlüssel von A ist, kann sie entschlüsseln. Da nur A seinen eigenen privaten Schlüssel kennt, *muss* die Nachricht von ihm stammen. Da es keinem Dritten möglich ist, die entschlüsselte Nachricht zu modifizieren und sie erneut mit dem privaten Schlüssel von A zu verschlüsseln, ist auch die Integrität der Nachricht sichergestellt. Den Vorgang des Überprüfens der Integrität und Authentizität bezeichnet man als *Verifizieren* einer digitalen Unterschrift.

Erzeugen und Verwalten von Schlüsseln

- Seit JDK 1.2 wird eine Schlüsseldatenbank verwendet, auf die mit Hilfe des Hilfsprogramms **`keytool`** zugegriffen werden kann. **`keytool`** kann Schlüsselpaare erzeugen, in der Datenbank speichern und zur Bearbeitung wieder herausgeben.
- Die Datenbank hat standardmäßig den Namen ".keystore" und liegt im Home-Verzeichnis des angemeldeten Benutzers (bzw. im Verzeichnis C:\Dokumente und Einstellungen\<Name> eines Windows-XP-Einzelplatzsystems).

Erzeugen eines Schlüsselpaars

- Wir wollen zunächst ein Schlüsselpaar mit dem Aliasnamen „pk1" erzeugen und mit dem Paßwort „pk1key" vor unberechtigtem Zugriff schützen. Die Schlüsseldatenbank wird beim Anlegen des ersten Schlüssels automatisch erzeugt und bekommt das Paßwort „keystore" zugewiesen. Wir verwenden dazu folgendes Kommando in der Eingabeaufforderung, dem mit **`–dname`** ein strukturierter Name des Schlüsselbesitzers mitgegeben wird:
- **`C:\> keytool -genkey -alias pk1 -dname "CN=FB Informatik,O=FH Dortmund,C=de"`**

```
C:\Dokumente und Einstellungen\Administrator>keytool -genkey -alias pk1 -dname "
CN=FB Informatik,O=FH Dortmund,C=de"
Geben Sie das Keystore-Passwort ein:  keystore
Geben Sie das Passwort f³r <pk1> ein.
        (EINGABETASTE, wenn Passwort dasselbe wie f³r Keystore):  pk1key
```

Überprüfen des Schlüsselpaars

- **keytool –alias pk1 –list –v**

```
C:\Dokumente und Einstellungen\Administrator>keytool -alias pk1 -list -v
Geben Sie das Keystore-Passwort ein:  keystore
Aliasname: pk1
Erstellungsdatum: 13.06.2004
Eintragstyp: keyEntry
Zertifikatskettenlönge: 1
Zertifikat[1]:
Eigent³mer: CN=FB Informatik, O=FH Dortmund, C=de
Aussteller: CN=FB Informatik, O=FH Dortmund, C=de
Seriennummer 40cc35e2
G³ltig ab: Sun Jun 13 13:09:22 CEST 2004 bis: Sat Sep 11 13:09:22 CEST 2004
Zertifikatfingerabdr³cke:
         MD5: 23:F6:DC:44:2F:73:0C:F9:65:B0:A2:DA:CD:C4:D0:1B
        SHA1: 45:40:7E:B1:BA:7D:92:D6:7E:6E:05:60:98:A4:B5:CB:0E:FB:D8:F5
```

- Es wird ein Eigenzertifikat für den gerade generierten öffentlichen Schlüssel erstellt, das dazu verwendet werden kann, um die digitale Unterschrift zu verifizieren.
- Dazu dient der Fingerprint, der an prominenter Stelle veröffentlicht werden kann, z.B. in einer E-Mail.

Signieren des Applets

- Es wurde bereits das JAR-Archiv **trapp.jar** erstellt.
- Dieses Archiv muss nun signiert werden. Dazu steht im JDK das Hilfsprogramm **jarsigner** zur Verfügung. Es arbeitet kommandozeilenbasiert und wird folgendermaßen aufgerufen:

jarsigner –signedjar strapp.jar trapp.jar pk1

```
C:\tmp>jarsigner -signedjar strapp.jar trapp.jar pk1
Enter Passphrase for keystore: keystore
Enter key password for pk1: pk1key
```

- Es wird das signierte Archiv mit dem Namen **strapp.jar** erzeugt.

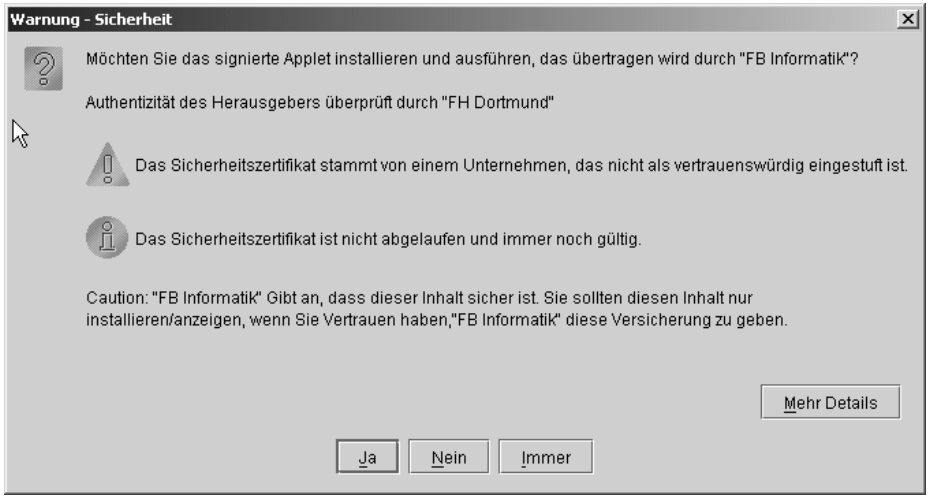
Laden des signierten Applets in HTML

- Um das Applet aus dem signierten jar-Archiv zu laden, muss die HTML-Datei folgendermaßen verändert werden:
- Beispiel SignedApplet.html:

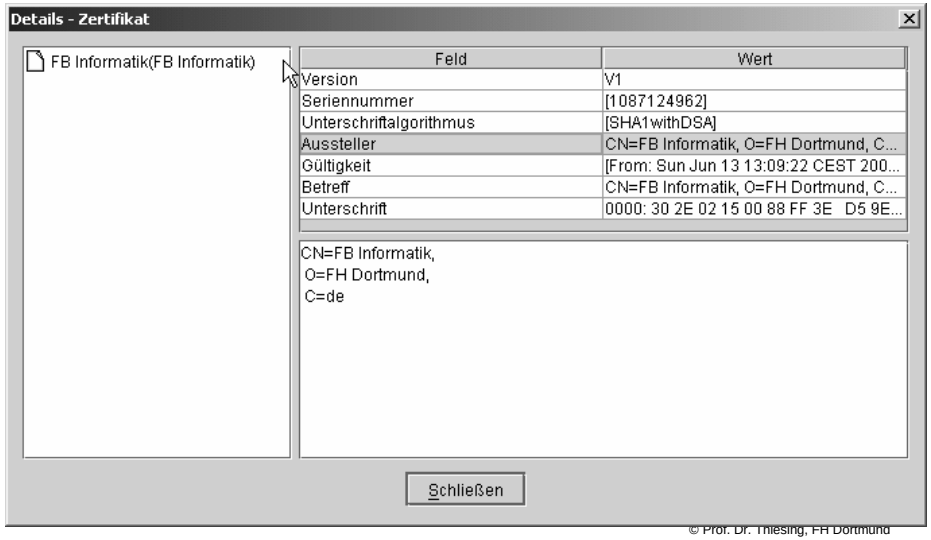
```
...
<h1>SignedApplet Demo</h1>

<applet
  archive="strapp.jar"
  code="TrustedApplet.class"
  width=600
  height=200
>
SignedApplet Demo
</applet>
...
```


Warnung bei Start des signierten Applets



Mehr Details



Ex- und Import von Zertifikaten

- Soll das signierte Applet auf anderen Arbeitsplätzen laufen, ist es erforderlich, das Zertifikat des Schlüssels, mit dem es signiert wurde, dort zu installieren. Soll es dagegen nur auf dem Arbeitsplatz laufen, auf dem auch der Schlüssel generiert wurde, ist das nicht erforderlich. Bei der Generierung des Schlüssels wurde ja auch ein (selbstsigniertes) Zertifikat erzeugt.

Export des Zertifikats

- Um das Zertifikat weitergeben zu können, muss es zunächst unter Verwendung der Option **`-export`** von **`keytool`** aus der lokalen Schlüsseldatenbank exportiert werden:

```
keytool -export -alias pk1 -file pk1.cert
```

Es liegt nun in der Datei **`pk1.cert`** und kann auf das Zielsystem kopiert werden.

Import des Zertifikats

- Mit der **`-import`**-Option von **`keytool`** kann es auf dem Zielsystem in die Schlüsseldatenbank aufgenommen werden:

```
keytool -import -alias pk1 -file pk1.cert
```

Nach dem Aufruf muss zunächst das Paßwort der Schlüsseldatenbank angegeben werden. Dann zeigt das Programm die Eigenschaften des Zertifikats an und erwartet, dass die Informationen bestätigt werden. Anschließend wird das Zertifikat in die Schlüsseldatenbank aufgenommen.

Policy-Datei

- Wir könnten das Applet jetzt wie zuvor starten, würden aber bei einem über das Netz geladenen Applet immer noch dieselbe **`SecurityException`** erhalten. Es ist zwar signiert und das Zertifikat ist auf diesem Rechner bekannt (denn hier wurde es erstellt). Die Policy-Datei ist aber noch nicht angepasst, und daher lehnt der **`SecurityManager`** des JDK die Ausführung der gefährlichen Operationen nach wie vor ab.

Anpassen der Policy-Datei

- Applets, die mit dem Zertifikat verifiziert werden können, das unter dem Alias „pk1" in der Schlüsseldatenbank abgelegt wurde, sollen Dateien im Verzeichnis **`c:\tmp\applets`** lesen und schreiben und auf die System-Properties "**`user.name`**", "**`user.home`**" und "**`user.dir`**" zugreifen können.

Policy-Datei

- Die Sicherheitseinstellungen des JDK werden mit Hilfe von Policy-Dateien definiert. Es gibt zwei Stellen im Dateisystem, von denen das JDK sie standardmäßig einliest:
  - Die System-Policies befinden sich in der Datei **`java.policy`** im Unterverzeichnis **`jre\lib\security`** des JDK-Installationsverzeichnisses. Diese Datei braucht normalerweise nicht verändert zu werden, sie enthält die globalen Sicherheitseinstellungen.
  - Die benutzerbezogenen Sicherheitseinstellungen können in der Datei **`.java.policy`** abgelegt werden. Sie liegt im Home-Verzeichnis des aktuellen Benutzers. Auf Windows-XP-Einzelplatzsystemen liegt sie (wie die Schlüsseldatenbank) im Verzeichnis **`c:\Dokumente und Einstellkungen\`**<user>. Diese Datei ist standardmäßig nicht vorhanden, kann aber leicht selbst angelegt werden.

Policy-Datei

- Policy-Dateien können auch an beliebigen anderen Stellen im Dateisystem liegen. In diesem Fall muss beim Aufruf des Java-Interpreters das System-Property "**`java.security.policy`**" mit dem Namen der zu verwendenen Policy-Datei gesetzt werden. Wäre beispielsweise **`pk1policy`** die zu verwendende Policy-Datei, so müsste der Appletviewer mit der Option "**`-J-Djava.security.policy=pk1policy`**" aufgerufen werden.

Das Format der Policy-Datei

- Policy-Dateien sind zeilenorientierte Textdateien, die mit einem gewöhnlichen Texteditor bearbeitet werden können. Alternativ stellt das JDK ein einfaches GUI-basiertes Hilfsprogramm mit dem Namen **`policytool`** zur Verfügung, mit dem Policy-Dateien erstellt und bearbeitet werden können.
- Eine Policy-Datei enthält zwei Arten von Einträgen. Beide sind optional:
  - Einen "keystore"-Eintrag, der die Lage der Schlüsseldatenbank angibt.
  - Beliebig viele "grant"-Einträge, mit denen Berechtigungen definiert werden.

Erstellen der Policy-Datei

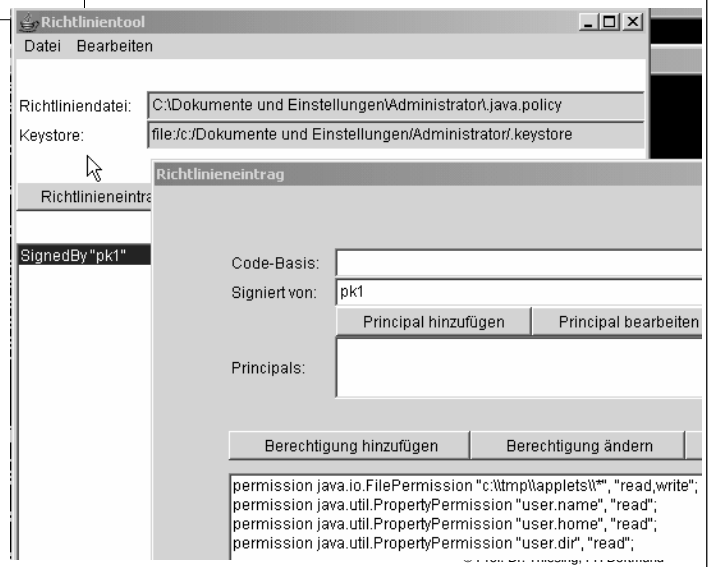
- C:\Dokumente und Einstellungen\<user>\.java.policy

```
keystore "file:///c:/Dokumente und
  Einstellungen/Administrator/.keystore";

grant SignedBy "pk1" {
  permission java.io.FilePermission
  "c:\\tmp\\applets\\*", "read,write";

  permission java.util.PropertyPermission
  "user.name", "read";
  permission java.util.PropertyPermission
  "user.home", "read";
  permission java.util.PropertyPermission
  "user.dir", "read";
};
```


Aufruf von policytool



Start des signierten Applets

- Der Start im Browser oder

  `appletviewer SignedApplet.html`

führt nun zu der erlösenden Meldung:

Und in **`c:\tmp\applets\TrustedApplet.log`** steht:

```
Erzeugt von Applet: 13.6.2004 14:4:38
user.name=Administrator
user.home=C:\Dokumente und Einstellungen\Administrator
user.dir=C:\Dokumente und
  Einstellungen\Administrator\Desktop
```

Die Klasse SecurityManager

- Die Prüfung der Zugriffsberechtigungen wird mit Hilfe der Klasse **`SecurityManager`** aus dem Paket java.lang vorgenommen.
- Zugriffe auf den **`SecurityManager`** sind an den Stellen der Laufzeitbibliothek eingebaut, an denen auf sensible Ressourcen zugegriffen wird.
- Applets besitzen grundsätzlich einen **`SecurityManager`**. Der AppletViewer bzw. Web-Browser sorgen während der Initialisierung für dessen Instanziierung.
- Applikationen dagegen haben normalerweise keinen **`SecurityManager`** (das ist der Grund, weshalb in Applikationen alle gefährlichen Operationen erlaubt sind). Soll eine Applikation einen **`SecurityManager`** erhalten, so kann sie entweder mit der Option "**`-Djava.security.manager`**" gestartet werden, oder der **`SecurityManager`** kann im Programm selbst installiert werden.