

Security



Security

Einführung

Klassenlader und Bytecode-Prüfung

Sicherheitsmanager und Berechtigungen



Sicherheitsmechanismen in Java

- Sicherheitsmechanismen¹ sind integraler Bestandteil von Java
- Folgende Mechanismen tragen zur Sicherheit bei:
 - Sprachliche Entwurfsmerkmale, z.B.
 - Bereichsprüfung bei Arrays
 - sichere Typecasts
 - keine Zeigerarithmetik
 - Zugriffskontrolle: steuern von zulässigen und unzulässigen Operationen, z.B.
 - Dateizugriff
 - Netzwerkzugriff
 - usw.
 - Codesignierung

¹ Sicherheit von Code bedeutet hier, dass der Code kein Unheil auf dem Computer anrichten kann



Komponenten des Sicherheitsmanagements

- Virtuelle Maschine
 - fehlerhafte Arrayzugriffe
 - fehlerhafte Casts
 - Integrität des Byte-Codes
- Klassenlader
 - Laden von Code
- Sicherheitsmanager-Klassen
 - erlaubte und unerlaubte Operationen und Zugriffe
- Verschlüsselungsmechanismen (java.security)
 - Codesignierung
 - Authentifikation



Einführung

Klassenlader und Bytecode-Prüfung

Sicherheitsmanager und Berechtigungen



Laden von Klassen

- In Java wird Code **inkrementell nach Bedarf** geladen

Prozess des Ladens von Klassencode

(veranschaulicht anhand Ausführung des Programms MyProgram):

- **Laden des Codes in File MyProgram.class**
- Laden von **allen Klassen von Variablen und Superklassen** der Klasse MyProgram
- **Ausführung der Methode main** der Klasse MyProgram
- Laden **aller bei der Ausführung von main benötigten Klassen**



Klassenlader

- Das Laden von Klassen wird durch CI assLoader-Objekte durchgeführt
- Von jeder Klasse aus kann auf den CI assLoader zugegriffen werden

```
Class clazz = Class.forName("MyProgram");  
ClassLoader loader = clazz.getClassLoader();
```

- CI assLoader erlaubt Klassen

- explizit zu laden

```
String className = "MyClass";  
Class myClass = loader.loadClass(className);
```

- und zu definieren

```
byte[] classCode = ...;  
loader.defineClass("DefinedClass", classCode, 0, classCode.length);
```



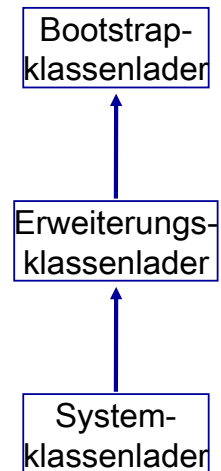
Die 3 Arten von Klassenlader

- Java verwendet 3 Arten von Klassenlader
 - Bootstrap-Klassenlader:
 - lädt alle Systemklassen aus JAR-Datei `rt.jar`
 - Erweiterungsklassenlader:
 - lädt alle Erweiterungen aus Verzeichnis `re/lib/ext`
 - Systemklassenlader (besser Anwendungsklassenlader):
 - lädt die Anwendungsklassen aus Klassenpfad (**CLASSPATH**)



Die 3 Arten von Klassenlader

- Klassenlader stehen in einer Über-/Untergeordnet-Beziehung zueinander
- Hierarchie
 - zu oberst Bootstrap-Klassenlader
 - dann Erweiterungsklassenlader
 - dann Systemklassenlader
- Jeder Klassenlader delegiert Laden einer Klasse an übergeordneten Klassenlader
und lädt die Klasse nur selber, wenn der übergeordnete scheitert



Explizites Laden von Klassen

- Klassen können auch explizit geladen werden
 - Explizit aber ohne benannten Klassenlader:
Es wird der Klassenlader der Klasse verwendet, in dem dieser Code steht

```
Cl ass. forName(cl assName);
```

- Mit einem Klassenlader-Objekt

```
Cl assLoader loader = new MyCl assLoader();  
loader. loadCl ass(cl assName);
```

- Zugriff auf Klassenlader

- es gibt einen Standard-Systemklassenlader

```
Cl assLoader. getSystemCl assLoader();
```

- Jeder Thread hat einen ContextCl assLoader

```
Thread t = Thread. currentThread();  
Cl assLoader loader = t. getContextCl assLoader();
```

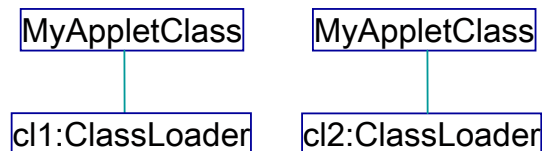


Eindeutigkeit von Klassen durch Klassenlader

- Eindeutigkeit einer Klasse ist gegeben durch:
 - vollständigen Name, d.h. inklusive Paketnamen (z.B. `java.sql.Date`)
 - und **Klassenlader**, durch den sie geladen wurde
- Dadurch ist es möglich, dass in einer VM
 - Klassen (mit gleichem Namen) von mehreren Quellen geladen und gleichzeitig ausgeführt wird
 - Klassen in mehreren Versionen gleichzeitig geladen werden

Beispiel:

- Applets geladen von mehreren Sites werden durch eigene Klassenlader unterschieden



Bytecode-Prüfung

- Nach dem Laden des Bytecodes durch den Klassenlader wird der Code durch einen Prüfer verifiziert
- Diese Prüfungen sind für fremden Bytecode wichtig
 - selbstcompilierter Bytecode ist sicher (!)
 - aber fremder Code kann händisch verändert und somit unsicher sein
- Folgende Prüfungen werden durchgeführt
 - alle Variablen initialisiert
 - Argumente bei Methodenaufrufen korrekt
 - keine Zugriffsverletzungen gemacht
 - Zugriffe auf lokale Variablen innerhalb des Gültigkeitsbereichs
 - kein Überlauf des Laufzeit-Stacks (!)

Achtung: Codeverifikation kann auch ausgeschaltet werden:

```
java -noverify MyProgram
```



Einführung

Klassenlader und Bytecode-Prüfung

Sicherheitsmanager und Berechtigungen



Sicherheitsmanager

- Bei Ausführung des Codes kann ein installierter Sicherheitsmanager prüfen, ob bestimmte (potentiell gefährliche) Operationen erlaubt sind; unter anderem:
 - aktueller Thread neuen **Klassenlader erzeugen** darf
 - aktueller Thread eine **Unterprozess erzeugen** darf
 - aktueller Thread eine **DLL laden** darf
 - aktueller Thread auf **Systemeigenschaften zugreifen und diese modifizieren** darf
 - aktueller Thread eine bestimmte **Datei oder Verzeichnis lesen, schreiben oder löschen** darf
 - aktueller Thread **Socket-Verbindung zu einem Host und Port** öffnen darf
 - aktueller Thread auf **eine Verbindung zu einem angegebenen Host und Port** warten darf
 - aktueller Thread bestimmte **Methoden eines anderen Threads oder ThreadGroup aufrufen** darf
 - eine Klasse einen **Druckauftrag starten** darf
 - eine Klasse auf die **Zwischenablage des Systems zugreifen** darf
 - eine Klasse auf die **AWT-Ereignisqueue zugreifen** darf
 - aktueller Thread ein **Fenster auf oberster Ebene öffnen** darf
 - aktueller Thread einen **eigenen Sicherheitsmanager installieren** darf
 - ein Thread eine **Applikation beenden** darf
 - ...



Ablauf einer Prüfung

- Operationen mit Sicherheitsprüfung beruhen auf folgenden Mechanismus
 - Zugriff auf installierten SecurityManager über `System.getSecurityManager()`
 - Wenn einer installiert ist (`!= null`), Aufruf der entsprechenden check-Methode beim SecurityManager
 - Die check-Methode wirft bei Nicht-Bestehen der Prüfung eine `SecurityException`
 - Ausführung der eigentlichen Operation nach bestandener Prüfung

```
public void <checkedOperation>(...) {  
    SecurityManager security = System.getSecurityManager();  
    if (security != null) {  
        security.<checkOperation>(...); // may throw SecurityException  
    }  
    <uncheckedOperation>(...);  
}
```

Beispiel: `exit`-Operation

```
public void exit(int status) {  
    SecurityManager security = System.getSecurityManager();  
    if (security != null) {  
        security.checkExit(status);  
    }  
    Shutdown.exit(status);  
}
```



Installation eines Sicherheitsmanagers

- Bei Applikationen ist standardmäßig kein SecurityManager installiert; es werden daher auch keine Prüfungen durchgeführt
- Installation eines SecurityManagers für eine Applikation mit
`System.setSecurityManager(SecurityManager sm)`
- In Java 2 stehen 2 SecurityManager zur Verfügung
 - `SecurityManager`: für Standardapplikationen
 - `RMI SecurityManager`: für RMI-Applikationen, mit Laden von Code von Remote-Sites
- Eigene SecurityManager können als Ableitungen realisiert werden
- Bei Applets:
 - bei Browsern wird automatisch der SecurityManager `AppletSecurity` installiert
 - damit laufen Applets in der Sandbox



Sicherheitsmodell bei Java 2

- Java 2 Sicherheitsmodell in der Form von Sicherheitsrichtlinien (*Security Policy*)

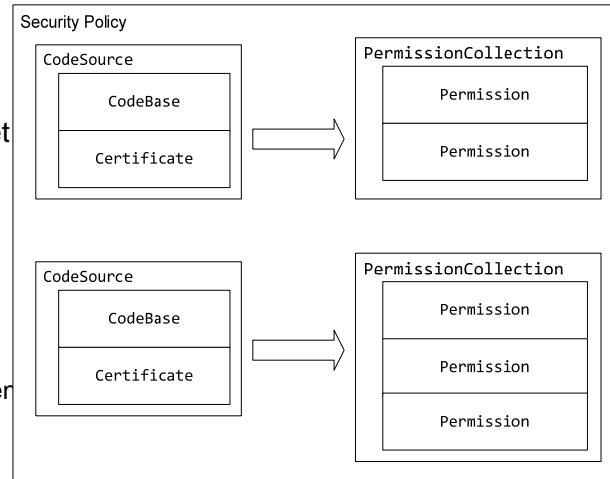
- Sicherheitsrichtlinien bilden
 - Codequellen (*code source*)
 - auf Berechtigungen (*permissions*)

ab, d.h. einer bestimmten Codequelle sind eine Menge von Berechtigungen zugeordnet

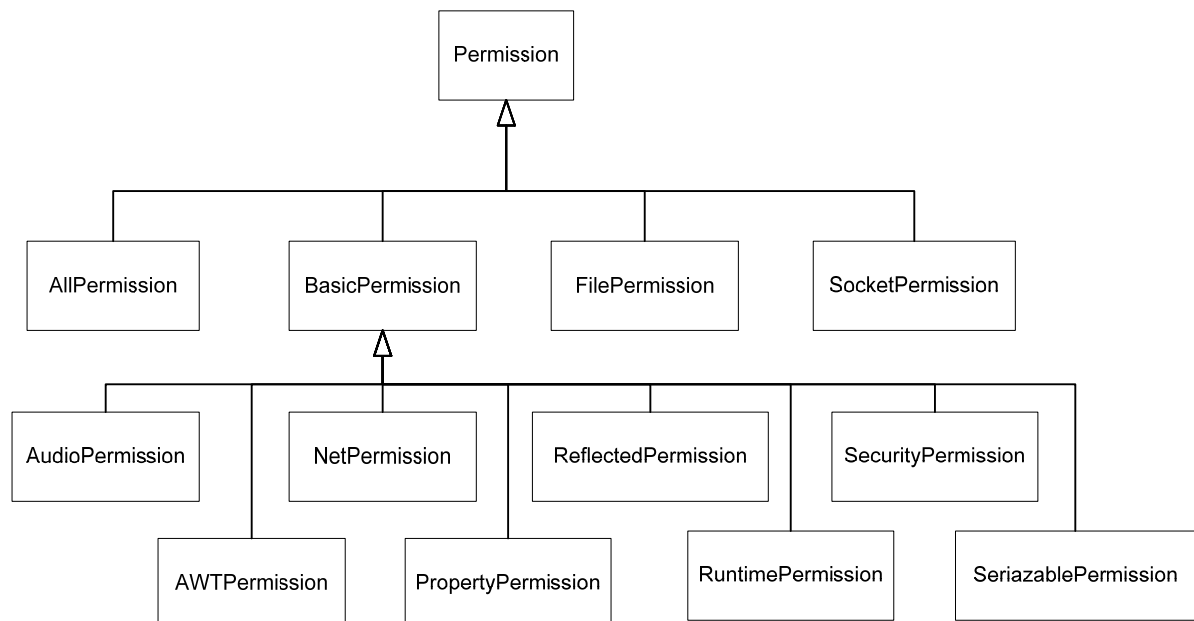
- Codequellen und Berechtigungen sind objektorientiert dargestellt

- Klassen *CodeSource*, *CodeBase* und *Certificate*
- Hierarchie von Berechtigungsklassen (*Permission*) in Mengen von Berechtigungen (*PermissionCollection*)

- Sicherheitsrichtlinien können in Dateien (*Policy-Files*) spezifiziert werden



Hierarchie von Berechtigungsklassen



Werden mit unterschiedliche Parametern konfiguriert!



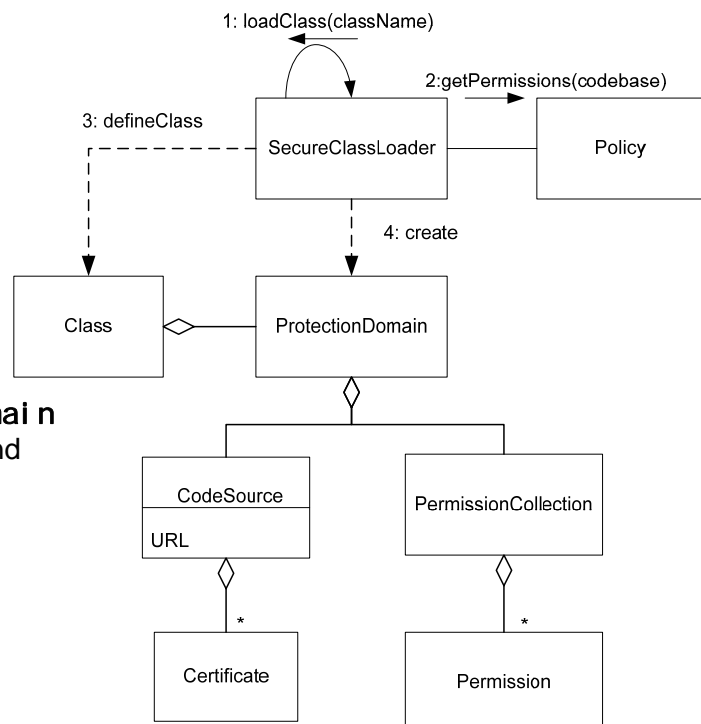
Standardmechanismus für Berechtigungsprüfungen

- baut auf Permissions auf
- Es spielen die folgenden Konzepte zusammen
 - Permission-Objekte und PermissionCollections und Prüfen von Permission-Objekten gegen PermissionCollections
 - SecureClassLoader (Basisklasse aller ClassLoader) mit der Installation von ProtectionDomains beim Laden von Klassen
 - Policy-Objekte, die Richtlinien in der Form von CodeSource und PermissionCollections liefern
- Berechtigungen werden bei Klassen in ProtectionDomains verwaltet



Ablauf beim Erzeugen eines ProtectionDomains

1. **Laden der Klasse** durch SecureClassLoader
2. **Holen der Permissions** aus Policy aufgrund der Codebase
3. **Definieren der Klasse** (defineClass)
4. **Erzeugen einer ProtectionDomain** für die Klasse mit CodeSource und Permissions



Ablauf einer Prüfung

```
checkForPermissions(Permissions p) { Pseudocode  
    for all classes of methods on call stack {  
        protectionDomain = class.getProtectionDomain();  
        permissions = protectionDomain.getPermissions();  
        if (required permission p for operation  
            checked against permissions != ok)  
            throw SecurityException(...);  
    }  
    return;  
}
```

Prüfung einer Permission p einer auszuführenden Operation läuft folgend ab

- Für alle Klassen für alle Methoden am Call-Stack
 - Zugriff auf die Permissions der ProtectionDomain der Klasse
 - Testen, ob Permissions der Klasse die erforderliche Permission p erlauben
- Wenn alle ProtectionDomains die Operation erlauben,
 - Prüfung bestanden
 - sonst Auslösung einer SecurityException



Beispiel: Ausgabe auf File

```
FileOutputStream fos = null;  
try {  
    fos = new FileOutputStream(fileName);  
    write(fos, model);  
} catch (Exception exc) {  
    Logger.getLogger(Logger.GLOBAL_LOGGER_NAME).log(Level.SEVERE,  
        exc.getMessage(), exc);  
} finally {  
    if (fos != null) {
```

Die Klassen aller Methoden am Stack müssen die entsprechende Permission haben!



Prüfen von Permissions gegen PermissionCollections

- Permission-Objekte werden verwendet für
 - Definition von Berechtigungen der Klassen (PermissionsCollections der ProtectionDomains)
 - Definition der notwendigen Berechtigung, um eine Operation auszuführen

z.B. wird bei checkExit folgendes ausgeführt

```
public void checkExit(int status) {  
    checkPermissions(new RuntimePermissions("exitVM"));  
}
```

Permission, die notwendig ist, um System.exit(..) auszuführen!

- Eine Permission p1 besteht Test gegenüber einer PermissionsCollection, wenn eine Permission p2 aus c die Permission p1 impliziert!

Beispiel exit:

```
public void exit(int status) {  
    SecurityManager security = System.getSecurityManager();  
    if (security != null) {  
        security.checkExit(status);  
    }  
    Shutdown.exit(status);  
}
```



Methode impliziert bei Permission

- Jede Permission implementiert eine Methode `boolean impliziert(Permission permission)` welche überprüft, ob sie die übergebene Permission „impliziert“
- Dadurch ist es möglich zu testen, ob eine für eine Operation notwendige Berechtigung durch eine in einer Klasse gegebene Berechtigung erlaubt wird

Beispiele:

`RuntimeException("**")`

impliziert

`RuntimeException("ExitVM")`

`FilePermission("C:\temp*", "read")`

impliziert

`FilePermission("C:\temp\MyFile", "read")`

`SocketPermission("*:1024-65535", "connect")`

impliziert

`SocketPermission("yourserver.com:1099", "connect")`



Policy-Klasse und Policy-Files

- SecureClassLoader erhält die eingestellten Sicherheitsrichtlinien vom installierten Policy-Objekt
- Standardmäßig ist das ein Objekt der Klasse PolicyFile, welches die Richtlinien aus Richtliniendateien (Policy-Files) liest
- Normalerweise sind zwei Policy-Files eingestellt

```
java.security.policy in der Java-Installation  
java.policy im Basisverzeichnis des Benutzers
```

- Policy-Files können auch bei Programmstart angegeben

```
java -Djava.security.policy=MyAll.policy MyApp
```

oder im Programm über Systemproperties eingestellt werden

```
System.setProperty("java.security.policy", "MyAll.policy");
```



Einstellungen zum Security-Mechanismus

globale Einstellungen zum Security-Mechanismus können in `java.security` vorgenommen werden, wie

- Policy-Klasse: z.B.
`policy.provider=sun.security.provider.PolicyFile`
- URLs von Policy-Files: z.B.

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



Format von Policy-Files

- Policy-Files bestehen aus einer Folge von grant-Einträgen

```
grant Codesource
{
    Permi ssi on_1;
    Permi ssi on_2;
};
```

- die Codesource besteht aus
 - einer URL für die Codebase und
 - einer Menge von Namen von vertrauenswürdigen Zertifikatsstellen

```
grant
    codebase codebase-URL
    certificate-name ...
{
    ...
};
```

- und jede Permission ist in der Form
 - Schlüsselwort `permi ssi on`
 - Klassenname der Permission-Klasse
 - einem berechtigungsspezifischem Zielwert (z.B. ein Verzeichnis oder eine Operation)
 - optional einer Liste von berechtigungsspezifischen Aktionen

```
...
{
    permi ssi on className
    target
    action1, ...;
    ...
};
```



Beispiel eines Policy-File

```
grant codeBase "file:${java.home}/lib/ext/*" {
    permi ssi on java. security. AllPermi ssi on;
};

grant {
    permi ssi on java. lang. Runti mePermi ssi on "stopThread";
    permi ssi on java. net. SocketPermi ssi on "local host: 1024-", "l i s t e n";
    permi ssi on java. uti l. PropertyPermi ssi on "java. versi on", "read";
    permi ssi on java. uti l. PropertyPermi ssi on "java. vendor", "read";
    ...
};

grant {
    permi ssi on javax. crypto. CryptoPermi ssi on "DES", 64;
    permi ssi on javax. crypto. CryptoPermi ssi on "DESede", *;
    ...
};

grant codeBase "www. ssw. uni -l i n z. ac. at/c l a s s e s/" {
    permi ssi on java. net. SocketPermi ssi on "*: 1024-65535", "connect";
    permi ssi on java. i o. Fi l ePermi ssi on "${user. home}${/}-",
        "read ", " write ",
    "execute";
    ...
};
...
```



Tabelle mit Permission-Einträgen (Auszug)

Permission	Target	Action
java.io.FilePermission	Dateiziel	read, write, execute, delete
java.net.SocketPermission	Socket-Ziel	accept, connect, listen, resolve
java.util.PropertyPermission	Eigenschaftsziel	read, write
java.lang.RuntimePermission	createClassLoader createSecurityManager exitVM stopThread queuePrintJob ...	
java.net.NetPermission	setDefaultAuthenticator specifyStreamHandler requestPassword-Authentication	
java.awt.AWTPermission	showWindowWithoutWarningBanner accessClipboard accessEventQueue listenToAllAWTEvents readDisplayPixels	
java.security.SecurityPermission	getPolicy, setPolicy ...	
...		



Literatur

- Horstmann, Cornell, Core Java 2, Band 2 - Expertenwissen, Markt und Technik, 2002: Kapitel 9
- Krüger, Handbuch der Java-Programmierung, 3. Auflage, Addison-Wesley, 2003, <http://www.javabuch.de>: Kapitel 47

