Sicherheit — was ist das?

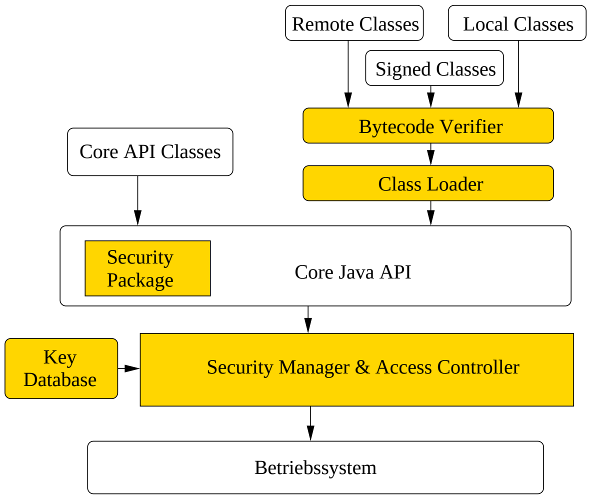
- *Safety*: Sicherheit gegen Bedrohung von innen
- *Security*: Sicherheit gegen Bedrohung von außen
- *Security* kann folgendes umfassen:

  Sicherheit vor *malware*, Datensicherheit (Geheimhaltung privater Daten), Authentisierung, Verschlüsselung, Offenheit (sensitive Operationen sichtbar), Korrektheit und Wohldefiniertheit (bestimmte Protokolle werden eingehalten), Zertifikation nach Orange Book, . . .

Das Java Sicherheitsmodell

- *The Java security model is designed to protect the information on a computer from being accessed or modified (including a modification that would introduce a virus) while still allowing the Java program to run on that computer.*
  (Scott Oakes: *Java Security*, O'Reilly 1998)
- Sicherheit tief in Java integriert
- Programme laufen in einer *sandbox*

Das Java-Sicherheitsmodell im Überblick



Der Class Loader

- Herkunft einer Klasse entscheidet über Privilegien:
  Applikation (voller Zugriff) vs. *Applet* (beschränkter Zugriff)
- Das Namensproblem: verschiedene Klassen gleichen Names unterschiedlicher Herkunft.
- Abstrakte Klasse `ClassLoader`, verschiedene Implementationen für:

  Interne Klassen (aus dem Java API und `CLASSPATH`)

  `URLClassLoader`: Zugriff via URLs; Sonderfall/Vorgänger: Applets

  `RMIClassLoader`: Zugriff auf entfernte Klassen via RMI)

  `SecureClassLoader` Sichere Klassen ()

  eigene Implementation (classes on the fly)

Der Bytecode Verifier stellt sicher:

- Korrektes Format des `class files`.
- Finalität von Klassen, Methoden und Attributen wird gewahrt.
- Jede Klasse (außer `java.lang.Object`) hat genau eine Superklasse.
- Keine illegale Konversion zwischen primitiven Typen.
- Keine illegale Typkonversion von Objekten (Konversion nur in Super- oder Subklasse).
- Keine Überlauf oder Unterlauf des Operandenstacks.

- Implementation: verzögert (bei Methodenaufruf etc.)
- Prüfungen zur Laufzeit: Felderindizes und Konversionen in Unterklassen

Security Manager & Access Controller

- Access Controller kontrolliert Zugriff auf Systemresourcen, wie

  Dateien lesen und schreiben

  Netzwerkzugriff (von Sockets senden und empfangen)

  Klassen laden und starten
- Teil der Package `java.security`
- Implementation von Sicherheitsrichtlinien (*security policies*) durch

  Permissions

  Domains

Der Security Manager

- Jedes Programm hat einen *security manager* (Klasse `java.lang.SecurityMana`

  `public static SecurityManager getSecurityManager()`

  `public static void setSecurityManager(SecurityManager s)`

  security manager kann nur einmal gesetzt werden.
- Default-Verhalten: sehr einfache *security policy* mit

  trusted classes: aus Kern-API oder dem `CLASSPATH` — dürfen alles

  untrusted classes: von Class Loader (irgendwoher) — eingeschränkte Rechte
- Zugriffskontrollen sind Methoden des *security managers*

Zugriffskontrolle

- Geschützt werden: das Dateisystem, Netzverbindungen, die Integrität der JVM, Threads, und System Resourcen.
- Beispiel Dateisystem:

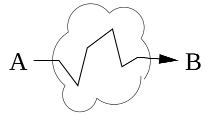
```
public void checkRead(FileDescriptor fd)
public void checkRead(String file)
public void checkRead(String file, Object context)
public void checkWrite(String file)
public void checkWrite(FileDescriptor fd)
```

- Methoden werden aufgerufen von anderen Klassen wie `FileInputStream()`, `File.renameTo()` oder `File.delete()`
- *untrusted classes* haben keinerlei Dateizugriff

Andere Beschränkungen für *untrusted classes*

- Netzwerk: nur Verbindungen zu dem Webserver, von dem sie geladen wurden.
- Integrität der JVM: keine `ClassLoader` kreieren, *stdin/stdout/stderr* umlenken, Fremdcode linken (`Runtime.load()`, `Runtime.loadLibrary()`), JVM beenden, Prozesse kreieren (`Runtime.exec()`)
- Threads: nur Manipulation von selbst kreierten Threads (und Thread-Gruppen)
- System Resourcen: kein Zugriff auf Drucker, System Clipboard, Event Queue, Systemeigenschaften (`System.getProperties`), nur gekennzeichnete Fenster
- Java selbst: nur `public` Methoden anderer Klassen, keine sicherheitsrelevanten Eigenschaften

Kryptographie: Einige grundlegende Begriffe

Situation: 

B will sicherstellen:

- *Authentisierung*: A Absender der Nachricht
- *Verschlüsselung*: Nachricht wurde nicht gelesen
- *Digitale Signatur:* Nachricht wurde nicht verändert

Verschlüsselungsverfahren

- Symmetrisch (*secret key*) — ein Schlüssel

  Verschlüsselung und Entschlüsselung mit gleichem Schlüssel

  Vorteil: schnell, sicher; Nachteil: Schlüssel muß geheim bleiben

  Beispiele: DES
- Asymetrisch (*public key*) — zwei Schlüssel: privat und öffentlich

  Verschlüsselung mit privatem Schlüssel

  Entschlüsselung mit öffentlichem Schlüssel

  Vorteil: sicher, Nachteil: langsam

  Beispiele: RSA, SHA

Die Java Cryptography Architecture (JCA)

- Generische Architektur: unabhängig von spezifischen Algorithmus, Implementation und Anbieter
- Abstrakte Klassen definieren Services (*engines*):

  `MessageDigest` — Prüfsummen

  `Signature` — digitale Signatur

  `KeyPairGenerator` — Generierung von Schlüsselpaaren

  `KeyStore` — Verwaltung von Schlüsseln
- Konkrete Klassen durch Instantiierung (*Factory*-Entwurfsmuster)

Beispiel: Nachricht digital signieren

1. Privaten Schlüssel besorgen
2. Signatur initialisieren `Signature.initSign()`
3. Daten lesen, Signatur aktualisieren `Signature.update()`
4. Datei signieren `Signature.sign()`

Zur Überprüfung der Signatur: ähnlich, aber

a. Zertifikat statt privatem Schlüssel

b. `Signature.initVerify()`

c. Am Ende `boolean Signature.verify()`

Zusammenfassung

- Sicherheit (*Security*) in Java durch Sandboxarchitektur
- Dadurch grösseres Maß an Sicherheit als in anderen Sprachen
- Implementierung von anspruchsvollen Sicherheitsrichtlinien möglich; *default policy* unterscheidet nur zwischen *trusted* und *untrusted*
- Reiche Kryptographie-Bibliothek in Java mit Authentisierung, Zertifizierung, Verschlüsselung und Schlüsselverwaltung
- **Kryptographie ist schwierig — Bibliotheken nutzen!**