Universität Potsdam

Security Considerations for Java Graders

Sven Strickroth

ABP2019

Gliederung

- Motivation
- Java Security Architektur
- Arbeitsspeicher
- Zeit
- Dateisystem
- Reflection
- Serialization und Deserialization
- Grader-Result Schnittstelle
- Zusammenfassung

Motivation

Anforderungen an Grader (Forišek 2006)

- Robustheit
- Vertrauen
  - System läuft verlässlich/nachvollziehbar und lässt sich nicht betrügen
  - sensible Daten werden vom System private gehalten

Allgemeine Angriffe

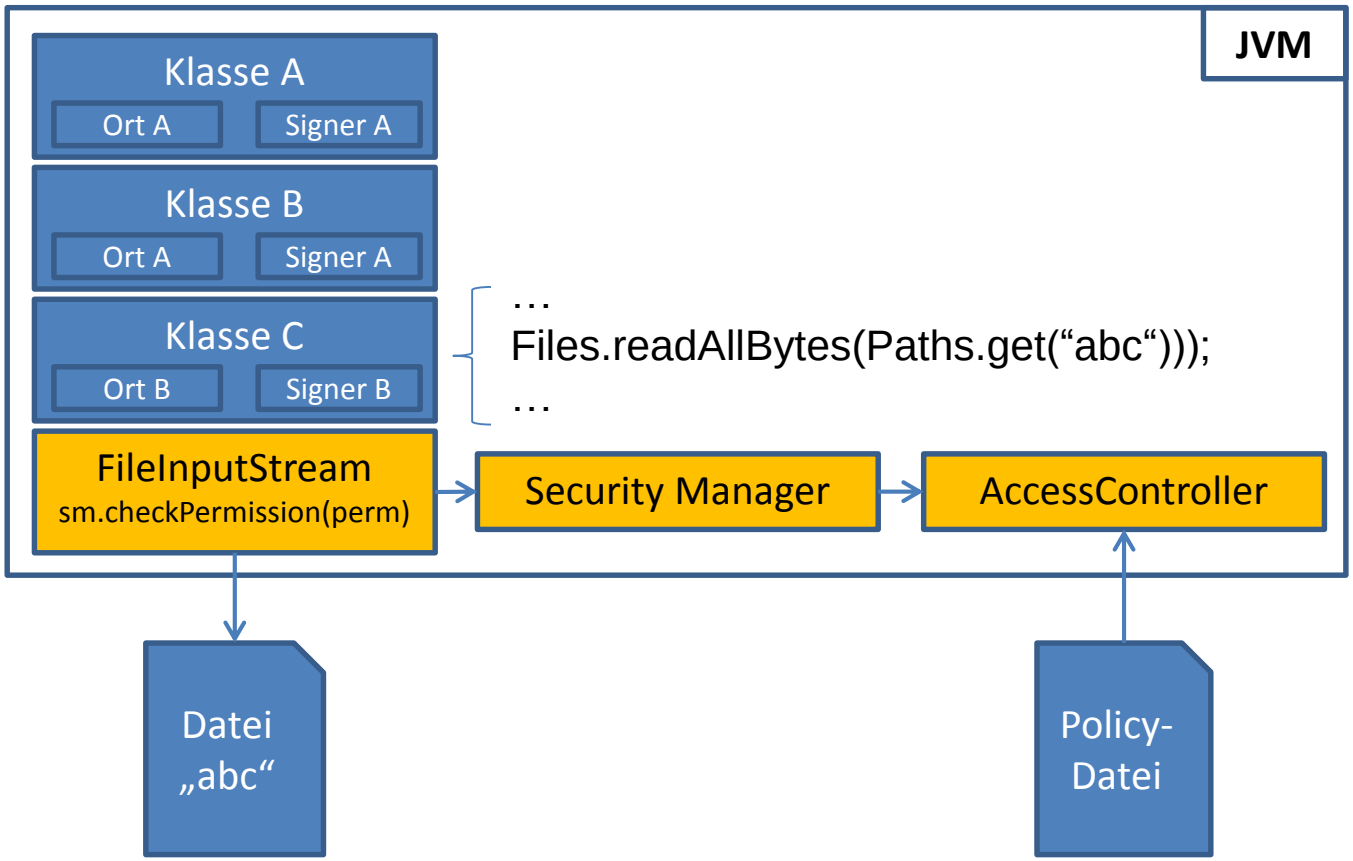
- Denial-of-Service
- Privilege Escalation
- Destructive Attacks

Eigenheiten von Java

Java Security Architektur

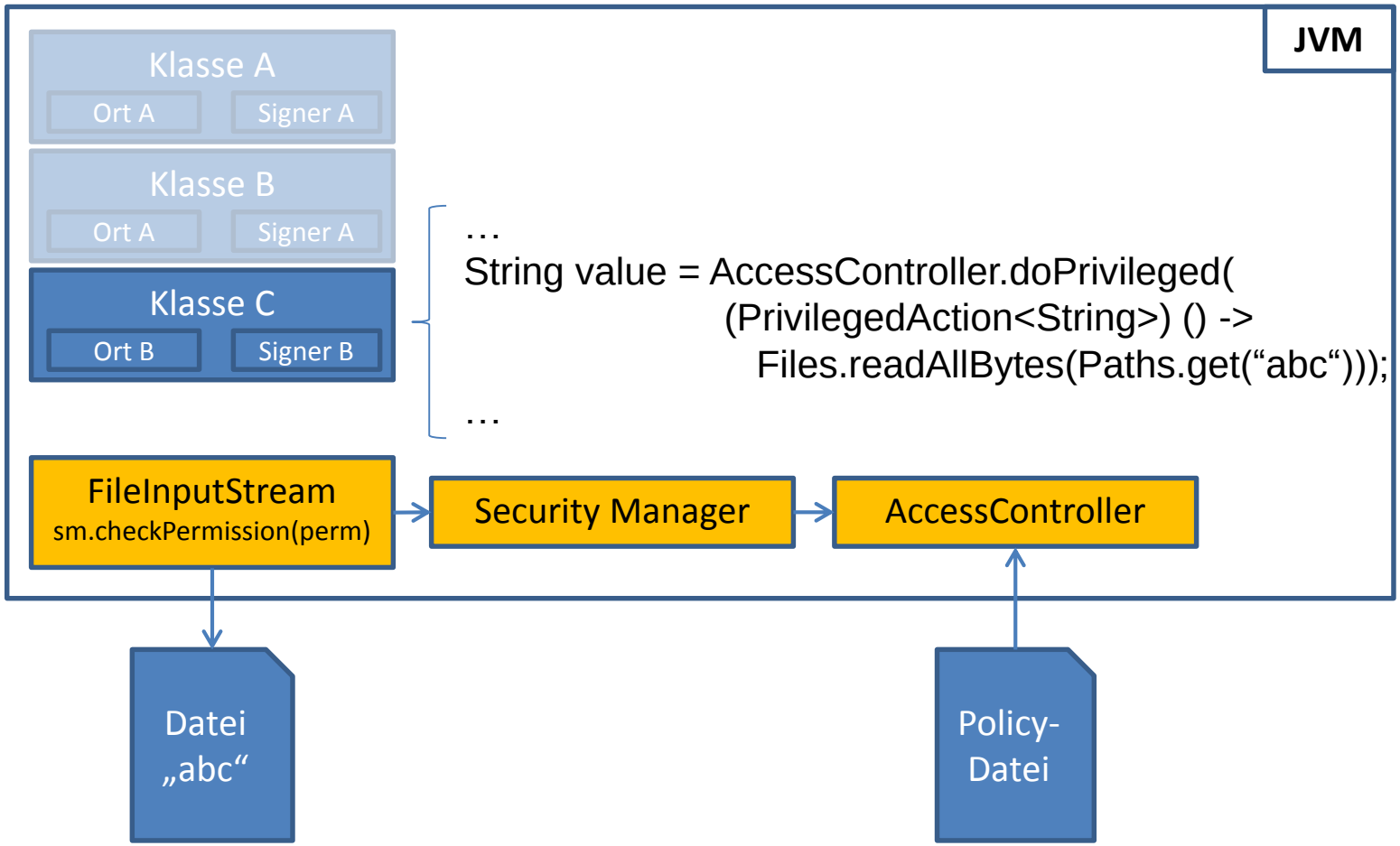
Code läuft by-default in einer Sandbox

Java besitzt einen stack-basierten Prüfsansatz



Spezialfall: PrivilegedAction

Stack-Prüfung kann „begrenzt“ werden:



Arbeitsspeicher

Problem: erschöpfende Speicherallozierung

Limitierbar mittels `setrlimit` (auf *nix)

JVM hat ein Limit bei 25% des verfügbaren Arbeitsspeichers

- schlägt Speicheranforderung fehl, kommt es zum `OutOfMemoryError`
- “a reasonable application should not try to catch an Error exception“ (JavaDoc)
- => keine shared JVM

⇒ Limit angemessen wählen (abh. von Parallelität, swapping, …)

⇒ Limits auch für Compiler (Beispiel folgt)!

Zeit

Problem: Endlosschleife, Deadlock oder unendliches Warten

Oft gesehen: `setrlimit`, limitiert aber nur CPU Zeit

Beispiel:

```
Semaphore semaphore = new Semaphore(1);
semaphore.acquire();
semaphore.acquire();
```

Läuft > 3 Stunden bei 5 Sekunden CPU-Zeitlimit (`ulimit -t5`)

=> „echte Zeit“ limitieren!

Compiler-Bomb

```
// Source: https://habr.com/en/post/245333/
class A {{
  int a;
  try {a=0;} finally {
  try {a=0;} finally {
  try {a=0;} finally {
  try {a=0;} finally {
  try {a=0;} finally {
  try {a=0;} finally {
  try {a=0;} finally {
  try {a=0;} finally {
  try {a=0;} finally {
  try {a=0;} finally {
  try {a=0;} finally {
  try {a=0;} finally {
  a=0;
  }}}}}}}}}}}
}}
```

Compile-Zeit: 3 Sekunden
Ausgabe: 6 MiB .class-Datei

OpenJDK 1.8.0_222

⇒ Restriktionen auch für Compiler

Dateisystem

Probleme:

- Zugriff auf “geheime” Materialien (z.B. korrekte Ausgaben für Tests, Musterlösungen)
- Verändern oder Beschädigen der Testumgebung
  - Alle Dateien löschen
  - Tests verändern/austauschen
  - Status-Informationen über Testläufe hinterlegen
  - Testergebnisse überschreiben
- zumüllen der Festplatte
  - direkt: Speicherplatz oder Inodes
  - indirekt: „große“ `stderr`/`stdout` Ausgaben
- hohe I/O-Last (viele kleine Schreibanfragen)

=> Verarbeitung von `stderr` und `stdout` limitieren

In Java Zugriff auf Dateien geregelt über explizite `FilePermissions`, **ABER:**

- automatische Leserechte für Dateien im CWD
- keine Platz- oder I/O-Beschränkungen
- Mögliche Probleme mit Classpath bei Schreibzugriff

Dateisystem: Schreibzugriff

Allgemein

⇒ Testbereich abschotten und nach jedem Test zurücksetzen

⇒ Speicherplatz beschränken (z.B. über Quotas, separate (logische) Partitionen oder fixed-sized Image-Dateien)

⇒ UID/GID-Konzepte umsetzen (z.B. Sims 2012)

Classpath

- Einspeisung eigener `.class`-Dateien
  - Umgehung statischer Code-Analysen
  - package-private Zugriff auf Testcode
  - existierende Klassen zu „überschreiben“
    - Standard `SecurityManager` hat Mitigations
- => am Besten: Classpath darf nicht beschreibbar sein

- Mitigations:
- => separate Class-Loader Instanzen
- => Test- und Studierenden-Code am Ende des Classpath

Reflection

Reflection ist **immer** nutzbar und benötigt im allgemeinen keine Permissions

alle `.class`-Dateien im Classpath sind instanziierbar (z.B. Musterlösung)

ABER: Zugriff auf protected/private Methoden/Felder benötigt explizite Permission

- `RuntimePermission "accessDeclaredMembers"`
- `ReflectPermission "suppressAccessChecks"`

Standard-Security-Manager hat weitere Einschränkungen für JDK, siehe Paper

Ausnutzbarkeit von Reflection muss im Einzelfall beurteilt werden, keine allgemeine Lösung bekannt.

Serialization und Deserialization

Mögliche Probleme mit der Deserialisierung (Svoboda 2016)

- Denial-of-Service (z.B. Speicherausnutzung, Deserialization Bomben)
- (remote) code execution, evtl. auch privilege escalation im Grader, Testcode, Library oder auch im JDK!

=> Niemals „fremde“ Daten deserialisieren! (Bytestream vergleichen)

Serialisierung kann genutzt werden, um

- Zugriff auf sonst protected/private Data zu erhalten (Bytestream)
- Instanziieren von Klassen mit privatem Konstruktor
- …

Betrifft alle Dateien im Classpath, die die `Serializable`-Schnittstelle implementieren

Benötigt keine Schreibrechte im Datensystem (`ByteArrayOutputStream`).

=> Kritische Klassen sollten nicht `Serializable` implementieren

Grader-Result Schnittstelle

Wie werden die Ergebnisse des Test-Codes bzw. des Graders an ein Aufrufendes System übermittelt?

- `stdout`
- Ergebnis-Datei
- API
- Prüfung des Exit-Codes der JVM

=> Sicherstellen, dass Ausgabe nicht gefälscht werden kann

- Studentencode könnte JUnit-Ausgabe fälschen
- Grader-API aufrufen
- Ergebnis-Datei überschreiben (z.B. über Shutdown-Hooks; brauchen explizite Permission)
- Beenden der JVM mit Exit-Code „0“ (`System.exit(0)` braucht keine Permission mit Standard `SecurityManager`)

Zusammenfassung

Vielen möglichen Problemen kann durch (die richtigen) Limits begegnet werden

Regelmäßig testen!

Kein *security by obscurity*

Auch Test-Code betrachten, wie gut ist der & von wem?

KISS: Grader, Permissions und Tests kurz, einfach und überschaubar halten

Beispiel-Code zum Testen:
https://gitlab.com/javagradersec/examples

Java Secure Coding Guildelines:
https://www.oracle.com/technetwork/java/seccodeguide-139067.html

Universität Potsdam

Dr. Sven Strickroth

Universität Potsdam
Institut für Informatik
Lehrstuhl für Komplexe Multimediale Anwendungsarchitekturen

Campus Griebnitzsee
Haus 4, Raum 1.15
0331 – 977 – 3068
sven.strickroth@uni-potsdam.de