# Themen zur Computersicherheit

# Hardware Security Module und PKCS #11

PD Dr. Reinhard Bündgen
buendgen@de.ibm.com

# Schutz wertvoller Schlüssel

- Problem: Wie kann man wertvolle Schlüssel schützen?
  - Selbst vor Subjekten mit höchster Autorisierung
    - z.B. jemand mit Unix Root-Autorisierung
      - kann den Speicher des Kerns und aller Prozesse lesen
      - kann all Dateien lesen
- Beispiel wertvoller Schlüssel/Daten
  - PINs von EC/Kreditkarten
  - Wegfahrsperren
  - militärische Geheimnisse
  - private Schlüssel von CAs
  - DNSEC, signieren von DNS zone files

# Hardware Security Modules (HSMs)

- Spezialhardware zum Abspeichern eines oder mehrerer Geheimnisse (Schlüssel), so dass
  - die Geheimnisse die HW nicht verlassen können
  - keine Seitenkanäle
  - die HW manipulationssicher (tamper proof) ist
- Beispiele
  - spezielle Kryptoadapterkarten
  - Smart Cards
  - Network HSMs
  - speziell abgeschirmte Räume

# Schlüsselreferenz bei HSMs

- nicht extrahierbare Schlüssel
  - Referenz via Handle (Index oder Hash)
- sichere Schlüssel (secure keys)
  - sensibel, aber extrahierbar

    HSM hat nicht-extrahierbaren Hauptschlüssel (master key)
  - alle extrahierbaren Schlüssel werden mit Hauptschlüssel verschlüsselt, wenn sie das HSM verlassen

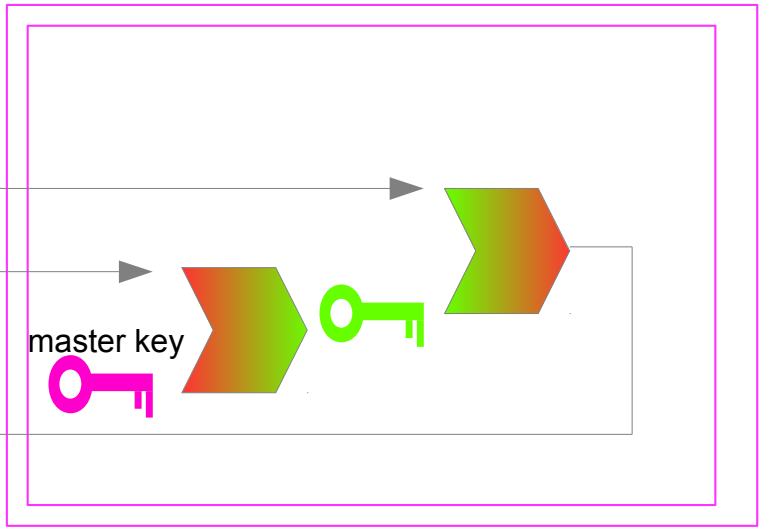
# Klartextschlüssel vs. sichere Schlüssel

## Klartextschlüssel (clear key)

Clear key in memory

## sicherer Schlüssel (secure key)


## Hardware Security Module (HSM) tamper proof

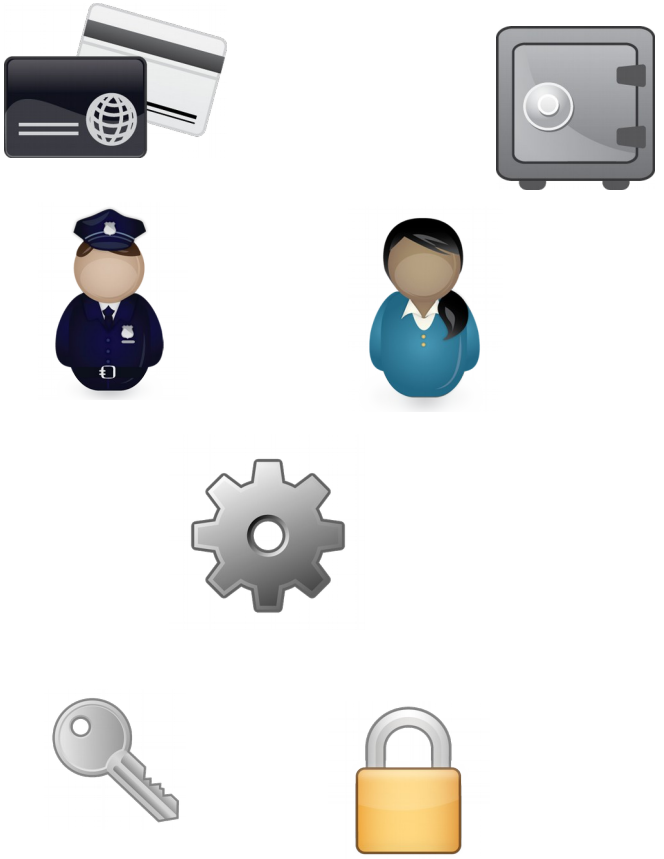


# Was ist PKCS #11

- ein verbreiteter kryptografischer Standard der ein API für kryptografische Methoden
  - ursprünglich von der RSA Firma veröffentlicht
  - z.Zt. bei OASIS beheimatet
  - aktuelle Version: 2.4 http://docs.oasis-open.org/pkcs11/
    - alte Version: 2.2 (+ 3 amendments), version 2.3 nur draft
  - cryptoki – Name des C/C++ API
- unterstützt die Entwicklung von SW die kryptographische Algorithmen nutzt
  - die in HW „Tokens“ implementiert sind
    - HSMs (Kryptoadapterkarten oder Smartcards)
    - Krypto-Beschleuniger (accelerators)
  - weitestgehend HW-unabhängig
  - unterstützt mehrere Token
  - unterstützt unterschiedliche Token
- mit PKCS #11 können bestehende Anwendungen zur Nutzung von kryptographischer HW konfiguriert werden
  - über konfigurierbare plug-in Mechanismen
    - Apache/mod_nss, IBM Webshpere Application Server
  - über kryptografische Bibliotheken mit PKCS#11 plug-in Möglichkeiten
    - gnutls
    - Java JCA/JCE
    - GSKIT (kryptografische Bibliothek der IBM)

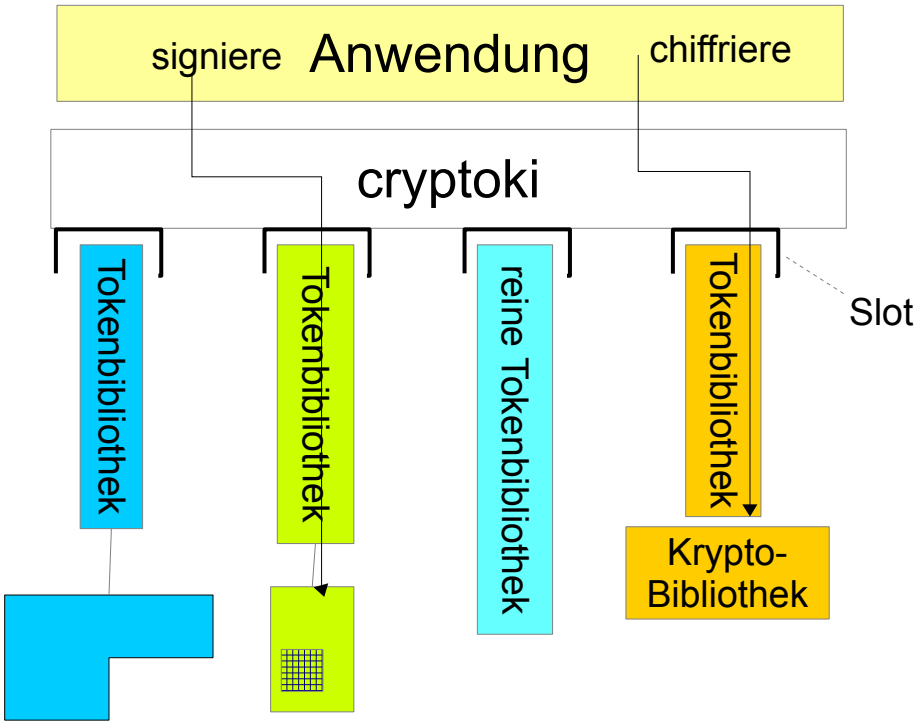
# PKCS #11 Konzepte

- Slots und Token
- Rollen und Sitzungen
- Funktionen und Mechanismen
- Objekte und Schlüssel (Zertifikate)
- Verschiedenes



# PKCS #11 Konzept: Slots und Token

- Modell: Smartcards und Lesergeräte
  - Lesegerät (Einschub): Slot
  - Krypto-HW: Token, das in Slot eingesteckt wird
- Slots und Token können HW spezifisch sein
- Slot- und Token-Funktionen
  - C_GetSlotList(), C_GetSlotListInfo()
  - C_WaitForSlotEvent()
  - C_InitToken(), C_GetTokenInfo()
  - C_initPin(), C_setPIN()
- Slot info
  - Token vorhanden
  - Gerät entfernbar
  - ...
- Token info
  - Login benötigt,
  - zu viele Fehlversuche bei PIN-Eingabe
  - hat RNG
  - ...



# PKCS #11 Konzepte: Rollen und Sitzungen

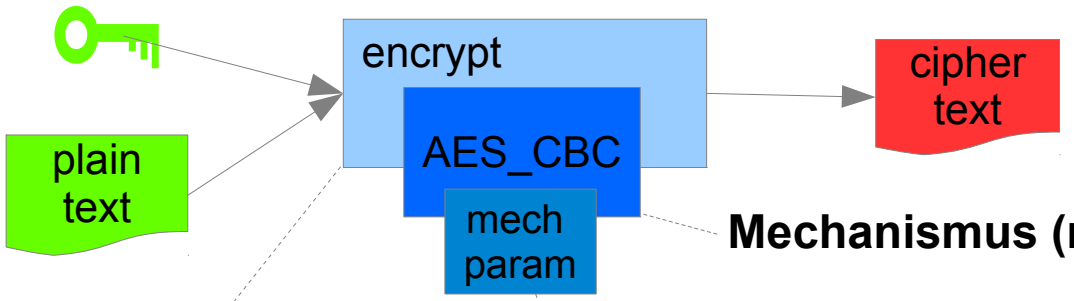
**Rollen (roles)**

- Security Officer (SO)
  - einer pro Token
  - hat SO PIN
  - initialisiert Token
  - gewährt Token normalen User
    - kann User PIN setzen
- (normaler) User
  - einer per Token
  - hat User PIN
  - kann sich in Sitzungen einloggen
  - kann private Objekt erzeugen und auf sie zugreifen
  - kann kryptographische Operationen ausführen

**Sitzungen (sessions)**

- Kontext für kryptographische Operationen
- hält Zustand  komplexer (multi part) Funktionen
- “nur eine” Operation pro Sitzung auf einmal
- Sitzungstypen
  - read-only / read write
  - public / user session
- user session – nach user login
- Sitzungsfunktionen
  - C_OpenSession() / C_CloseSession()
  - C_GetSessionInfo()
  - C_GetOperationState() / C_SetOperationState()
  - C_Login() / C_Logout()

# PKCS #11 Konzepte: kryptographische Funktionen und Mechanismen




**kryptographische Funktion (function)**

- generische kryptographische Funktion
  - z.B. C_Encrypt(), C_Sign()
- läuft im Kontext einer Sitzung
- wird durch einen Mechanismus instanziiert
- C_Init*Fkt*(session, mechanism, key,...)
- einfache (single part) Funktionen
  - C_*Fkt*Init(); C_*Fkt*();
- komplexe (multi part) Funktionen (zur Bearbeitung langer Nachrichten)
  - C_*Fkt*Init(), C_*Fkt*Update(), ..., C_*Fkt*Update(); C_*Fkt*Final();
- Es ist abhängig vom Token welche Funktionen unterstützt werden
  - C_GetFunctionList()

**Mechanismus (mechanism)**

- Menge spezifischer kryptographischer Verfahren (e.g. CKM_AES_CBC)
- implementiert kryptographische Funktion
- Mechanismenattribute definiert in CK_MECHANISM_INFO Struktur
  - min/max Schlüssellängen
  - unterstützte Funktionen
  - HW Unterstützungs-Flag
- einige Mechanismen haben Parameter, z.B. um den IV für CKM_xyz_CBC zu spezifizieren
- die Menge der unterstützten Mechanismen und die von ihnen unterstützten Funktionen hängt vom Token ab
  - C_GetMechanismList()
  - C_GetMechanismInfo()

# PKCS #11 kryptographische Funktionen

| *Fkt* | C_*Fkt*Init | C_*Fkt* | C_*Fkt*Update | C_*Fkt*Final | Comment |
|---|---|---|---|---|---|
| Encrypt | x | x | x | x | |
| Decrpt | x | x | x | x | |
| Digest | x | x | x | x | kein Schlüsselargument für DigestInit |
| DigestKey | | x | | | wird wie DigestUpdate genutzt |
| Sign | x | x | x | x | |
| SignRecover | x | x | | | einfache Funktion |
| Verify | x | x | x | x | |
| VerifyRecover | x | x | | | einfache Funktion |
| DigestEncrypt | | | x | | jede Funktion muss individuell initialisiert und finalisiert werden |
| DecryptDigest | | | x | | |
| SignEncrypt | | | x | | |
| DecryptVerify | | | x | | |
| GenerateKey | | x | | | symmetrische Schlüssel |
| GenerateKeyPair | | x | | | asymmetrisches Schlüsselpaar |
| WrapKey | | x | | | implizite Initialisierung |
| UnwrapKey | | x | | | |
| DeriveKey | | x | | | |
| SeedRandom | | x | | | nutzt keine Mechanismen |
| GenerateRandom | | x | | | |

C_VerifyUpdate()

# PKCS #11 Mechanismen (Beispiele)

| Mechanismus | unterstützte Funktionen | | | | | | |
|---|---|---|---|---|---|---|---|
| | Encrypt Decrypt | Sign Verify | SE VR | Digest | Gen Key / Key Pair | Wrap Unwrap | Derive |
| | | | | | | | |
| CKM_RSA_PKCS_KEY_PAIR_GEN | | | | | x | | |
| CKM_RSA_PKCS | x | x | x | | | x | |
| CKM_SHA256_RSA_PKCS | | x | | | | | |
| … | | | | | | | |
| CKM_EC_KEY_PAIR_GEN | | | | | x | | |
| CKM_ECDSA | | x | | | | | |
| CKM_ECDSA_SHA1 | | x | | | | | |
| CKM_ECDH1_DERIVE | | | | | | | x |
| | | | | | | | |
| CKM_AES_KEY_GEN | | | | | x | | |
| CKM_AES_ECB | x | | | | | x | |
| CKM_AES_CBC | x | | | | | x | |
| CKM_AES_CBC_PAD | x | | | | | x | |
| CKM_AES_CTR | x | | | | | x | |
| CKM_AES_MAC | | x | | | | | |
| | | | | | | | |
| CKM_SHA256 | | | | x | | | |
| CKM_SHA256_HMAC_GENERAL | | x | | | | | |
| CKM_SHA256_HMAC | | x | | | | | |
| CKM_SHA256_KEY_DERIVATION | | | | | | | x |

Diese Tabelle hat in der Version 2.2 des PKCS#11 Standards ca. 300 Zeilen.

# PKCS#11 Konzepte: Objekte und Schlüssel (Zertifikate)

**Objekte**

- Objektklassen
  - Sitzungsobjekte (volatil) vs Tokenobjekte (persistent)
  - privat vs public
  - read-only vs read-write
- auf private Objekte
  - kann nur in User Sitzung (Login!) zugegriffen werden
- Objektattribute
  - Type, Wert, Größe (length) der Werts
- Objektverwaltungsfunktionen
  - C_CreateObject(),
  - C_CopyObject()
  - C_DestroyObject()
  - C_GetObjectSize()
  - C_{Get|Set}AttributeValue()
  - C_FindObjects[Init|Final]()

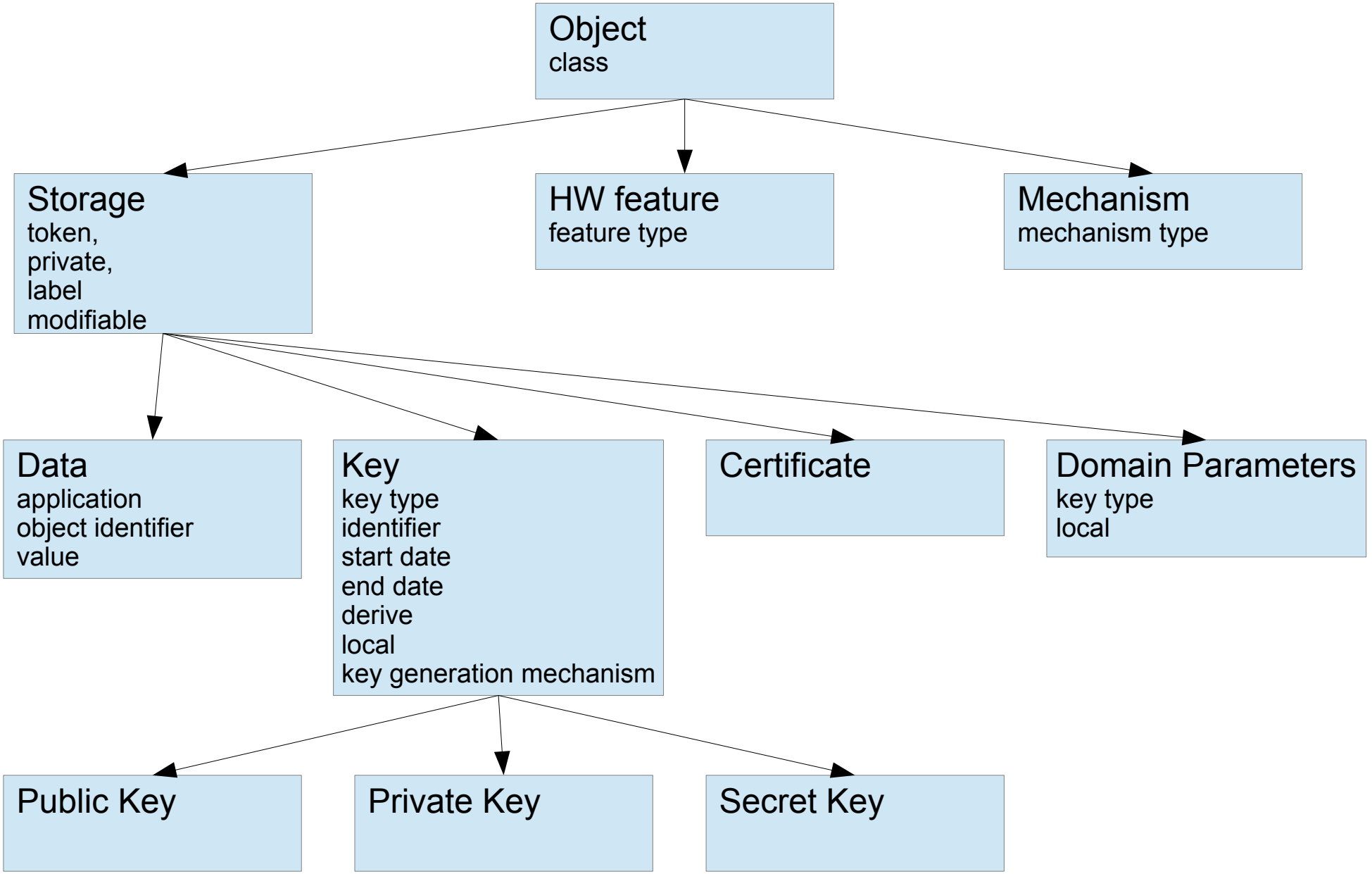
**Schlüsselobjekte**

  - private Schlüssel (CKO_PRIVATE_KEY)
  - öffentliche Schlüssel (CKO_PUBLIC_KEY)
  - geheime Schlüssel ( CKO_SECRET_KEY)
- einige Attribute von Schlüsselobjekten.
  - CKA_WRAP
  - CKA_SENSITIVE (nicht für öffentl. Schlüssel)
  - CKA_MODULUS (nur RSA)
  - Attribute für erlaubte Operationen
- Schlüsselverwaltungsfunktionen
  - C_GenerateKey()
  - C_GenerateKeyPair()
  - C_WrapKey()
  - C_UnwrapKey()
  - C_DeriveKey()

**Zertificatesobjekte**

  - X.509
  - WTLS
  - keine Funktionen auf Zertifikaten

# PKCS#11 Objektklassenhierarchie



# PKCS #11 Konzepte für sichere Schlüssel in HSMs

- Schlüsselattribute
  - CKA_SENSITIVE: Schlüsselwert kann nicht im Klartext gezeigt werden
  - CKA_ALWAYS_SENSITIVE: Schlüssel hatte immer das Attribut CKA_SENSITIVE = CK_TRUE gehabt
  - CKA_EXTRACTABLE: Schlüssel können das Token (HSM) verlassen, falls CKA_EXTRACTABLE = CK_FALSE, kann der Schlüssel das Token nicht verlassen.
  - CKA_NEVER_EXTRACTABLE: Schlüssel hatte nie das Attribute CKA_EXTRACTABLE = CK_TRUE gehabt
  - CKA_LOCAL: Schlüssel wurde lokal erzeugt

- Schlüsselexport
  - um Kommunikationspartner Schlüssel zuzusenden
    - „Transportschlüssel“
    - Exportformat: Byte-String
    - C_WrapKey()

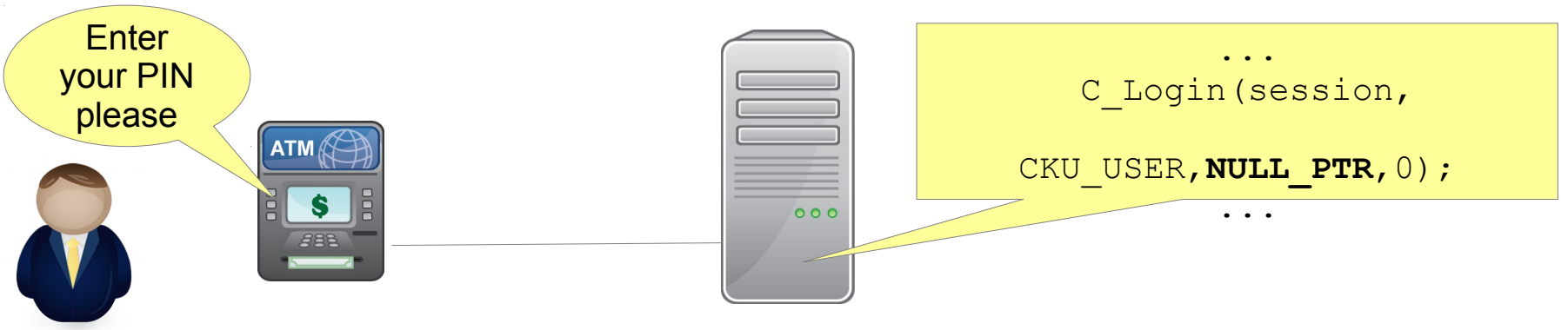
- Schlüsselimport
  - um Schlüssel von Kommunikationspartner zu empfangen
    - C_UnwrapKey()
    - Importformat: Byte-String
  - zum Testen
    - known answer tests
    - C_CreateObject()

# PKCS #11 Konzepte: Verschiedenes

- Unterstützung für parallelen Tokenzugriff (z.B.  multi threading)
  - C_Initialize() hat Argument, das beschreibt
    - threading Fähigkeiten
    - Synchronizationsfunktionen (Mutex)

- Slot -Verwaltung
  - C_WaitForSlotEvent()
    - z:B. hinzufügen bzw wegnehmen von Token

- Token PIN Eingabe am physischen Token
  - CKF_PROTECTED_AUTHENTICATION_PATH Flag muss in der Token Info gesetzt sein
  - C_Login() wird mit NULL_PTR als PIN Argument aufgerufen



# Ein typischer PKCS #11 Programmfluss (vereinfachter C Code)

```
#include <pkcs11types.h>

…

rc = C_Initialize(...);

rc = C_GetSlotList(...);

rc = C_GetSlotInfo(slot,...);

rc = C_GetTokeInfo(slot,...);

rc = C_OpenSession(slot,..., &session);

rc = C_Login(session, ...)


rc = C_Logout(session)

rc = C_CloseSession(session)

rc = C_Finalize(...)
```

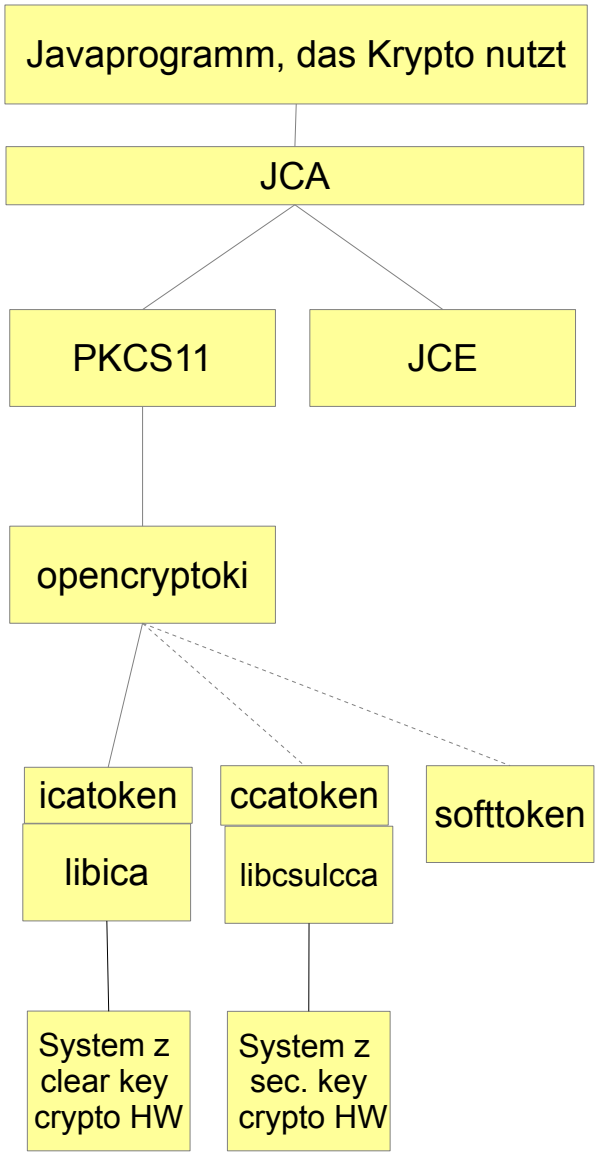
initialization /
session handling

```
C_EncryptInit(session, mechanism, key);

while ( /* there are still pieces of the message left */ ) {

        C_EncryptUpdate(session, message_part, …);
        ...

}

C_EncryptFinal(session, last_part, ...);

...
```

cryptographic operation(s)
inside a session

# PKCS #11 & Java, Die Java Cryptographic Architecture (JCA)

- Java Cryptography Extension (JCE)
- API für grundlegende kryptographische Algorithmen
- Java Cryptography Architecture (JCA)
  - Provider-Architektur für Sicherheits-APIs
  - unterstützt mehrere „Provider“ mit unterschiedlichen Prioritäten und Fähigkeiten
  - Provider die JCE API implementieren:
    - SunJCE / IBMJCE: Softwareimplementierung
    - SunPKCS11/ IBMPKCS11Impl ruft PKCS#11 API auf (z.B. openCryptoki)



# Java Konfiguration um Crypto-HW zu nutzen

Die `java.security` Datei definiert die verfügbaren JCA Provider.

Standardlokcation:

`/usr/lib/jvm/java-<version>/jre/lib/security/java.security`

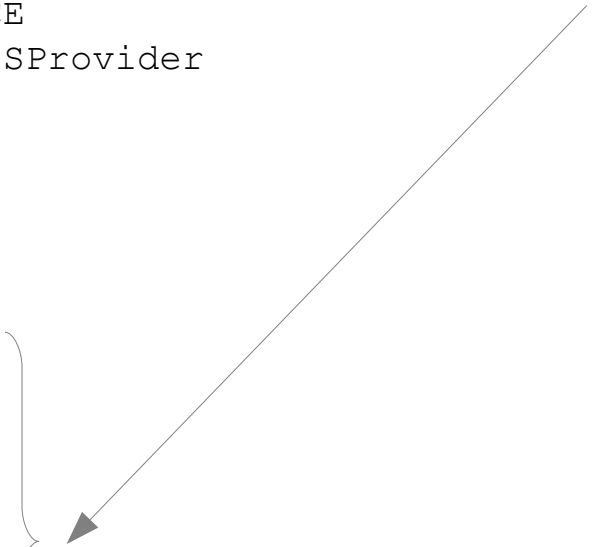
Beispielextrakt aus java.security

```
...
# List of providers and their preference orders (see above):
#
security.provider.1=com.ibm.crypto.pkcs11impl.provider.IBMPKCS11Impl /root/zpkcs.cfg
security.provider.2=com.ibm.crypto.provider.IBMJCE
#security.provider.3=com.ibm.security.jgss.IBMJGSSProvider
...
```

Der IBMPKCS11Impl Provider hat eine Konfigurationsdatei als Argument

Beispielkonfiguration für IBMPKCS11Impl:

```
name = Sample
description = Sample config for Linux on z
library = /usr/lib64/pkcs11/PKCS11_API.so
# the following references the icatoken
slot = 0
# the following references the ccatoken
#slot = 1
# the following references the softtoken
#slot = 2
disabledmechanisms = { CKM_SHA_1 }
```



# Auswahl der JCA Provider

Die Java Cryptography Architecture (JCA)

- stellt plug-in Mechanismus für „provider“ von kryptographischen Funktionen zur Verfügung
- *XXX*getInstance() Funktion wählt provider für Klasse *XXX* *aus*
- implizite Providerauswahl:
  - kein provider im optionalen Argument von *XXX*getInstance() Aufruf angegeben
  - JCA wählt für jede kryptographische Funktion einen Provider basierend auf den Fähigkeiten und der Prioritäten der Provider aus.
  - die Providerpriorität ist über die Reihenfolge der Providereinträge in der `java.security` Datei definiert.

AES → provider 2
RSA → provider 1
DH von keinem provider unterstützt