Implementierung eines eigenen JCE-Providers

Hardware-Sicherheits-Module und Java

Christoph Pfeifer – Radebeul, 28.09.2018

robotron®

Themen

- Einführung ins Thema
- JCE-Provider
- Beispiel Signaturerzeugung

Einführung ins Thema

Datensicherheit – warum überhaupt?

- Gesetzliche Anforderungen (DSGVO, IT-Sicherheitsgesetz, …)
- Behördliche Anforderungen (BSI, …)
- Kundenanforderungen (Kritische Infrastrukturen, Reputation des Kunden, …)

Schutzziele

Vertraulichkeit	• Informationen werden nur Berechtigten bekannt
Integrität	• Informationen sind richtig, vollständig und aktuell oder aber dies ist erkennbar nicht der Fall
Verfügbarkeit	• Informationen sind dort und dann zugänglich, wo und wann sie von Berechtigten gebraucht werden

Schutzmaßnahmen

- Ende-zu-Ende Verschlüsselung
- Anonymisierungsdienste
- Signatursysteme
- Hochverfügbarkeitssysteme
- …

Hardware-Sicherheits-Modul

- Wikipedia:
  - „Der Begriff **Hardware-Sicherheitsmodul** […] bezeichnet ein internes oder externes Peripheriegerät für die effiziente und sichere Ausführung kryptographischer Operationen oder Applikationen.“ [Wikipedia]
- Generierung und Speicherung privater Schlüssel → Operationen darauf werden direkt im HSM durchgeführt
- Generierung sicherer Zufallszahlen

Umgesetzte Schnittstellen

- Utimaco CXI
- Utimaco EID
- Worldline JSS
  - Schlüsselgenerierung außerhalb vom HSM
- Utimaco PKCS#11
  - Keine KeyAgreement Unterstützung auf HSM Seite
  - Für uns nicht nutzbar, da auch Verschlüsselung gebraucht wird

Java Cryptography Extension Provider

robotron®

Allgemeines

```
System.out.println(Arrays.toString(Security.getProviders()));
```

Standardprovider:
[SUN version 1.8, SunRsaSign version 1.8, SunEC version 1.8, SunJSSE version 1.8, SunJCE version 1.8, SunJGSS version 1.8, SunSASL version 1.8, XMLDSig version 1.8, SunPCSC version 1.8, SunMSCAPI version 1.8]

Mit zusätzlichen Providern:
*[SUN version 1.8, SunRsaSign version 1.8, SunEC version 1.8, SunJSSE version 1.8, SunJCE version 1.8, SunJGSS version 1.8, SunSASL version 1.8, XMLDSig version 1.8, SunPCSC version 1.8, SunMSCAPI version 1.8, **BC version 1.59, RDS-HSM version 3.0**]*

Signatur der JAR Datei

- Oracle Java: Provider muss signiert sein mit Zertifikat aus Oracle CA:
  - Einige Funktionen prüfen Signatur (z.B. Key Agreement)
  - Andere Funktionen prüfen sie nicht (z.B. Hashwerte, Signaturen)
- JAR mit der Provider Implementierung muss signiert sein
- Abhängige JARs müssen signiert sein, wenn gleiche Packages genutzt werden
- OpenJDK: keine Signatur notwendig

Java Cryptography Architecture

Provider

- Zuordnung von „Servicenamen“ zu Services
- Eigener Service erweitert jeweiliges Service Provider Interface des gewünschten Dienstes
- Konstruktoren argumentlos
- Pfad wird über Punktnotation abgebildet
  - z.B. KeyPairGenerator.EC bedeutet, KeyPairGenerator mit Algorithmus EC
- Bei Zugriff wird der Pfad gebildet und in der Zuordnung gesucht
- Registierung des Providers über:
  `Security.addProvider(provider)`
  `Security.insertProviderAt(provider, position)`

```
addService("KeyAgreement.ECDH", HsmKeyAgreement.class);
addService("KeyManagerFactory." + NAME, HsmKeyManagerFactory.class);
addService("TrustManagerFactory." + NAME, HsmTrustManagerFactory.class);
addService("KeyPairGenerator.EC", HsmKeyPairGenerator.class);
addService("KeyPairGenerator.RSA", HsmKeyPairGenerator.class);

addService("Signature.NONEwithECDSA", HsmSignatureEcdsaNoHash.class);
addService("Signature.SHA1withECDSA", HsmSignatureEcdsaSha1.class);
addService("Signature.SHA256withECDSA", HsmSignatureEcdsaSha256.class);
addService("Signature.SHA384withECDSA", HsmSignatureEcdsaSha384.class);
addService("Signature.SHA512withECDSA", HsmSignatureEcdsaSha512.class);

addService("Signature.NONEwithRSA", HsmSignatureRsaNoHash.class);
addService("Signature.SHA1withRSA", HsmSignatureRsaSha1.class);
addService("Signature.SHA256withRSA", HsmSignatureRsaSha256.class);
addService("Signature.SHA384withRSA", HsmSignatureRsaSha384.class);
addService("Signature.SHA512withRSA", HsmSignatureRsaSha512.class);

addService("SecureRandom." + NAME, HsmRandom.class);

private void addService(final String service, final Class<?> implementation) {
    AccessController.doPrivileged(new RegisterServiceAction(this, service,
implementation));
}

provider.put(service, implementation.getName());
```

Problem - Abbildung von Entitäten

- Entitäten dienen zur Trennung von Objekten im HSM
- Sind für Mandantentrennung sehr wichtig
- Keystore bietet direkt keine Möglichkeit, Schlüssel abhängig von einer Entität abzufragen
  - Problem wird über Kodierung des Alias gelöst
  - Schlüsselobjekte werden direkt über den Konstruktor im Code erzeugt

Beispiel Signaturerzeugung

SignatureSpi Erweiterung

- Implementierung der HSM Signaturklasse liegt im gleichen Package wie der HSM Provider
- Über package-sichtbare Methoden kann über den HSM Provider auf die HSMs zugegriffen werden
  - `getSecurityModule`, `getKeyIdentifier`
- Eigenes Interface `BasicCryptographyService` versteckt die eigentliche Implementierung der konkreten HSM Schnittstelle

```
@Override
protected final void engineInitSign(PrivateKey privateKey) throws
InvalidKeyException {
    this.privateKey = getKeyIdentifier(privateKey);
    this.sign = true;
    this.data = new ByteArrayOutputStream();
}

@Override
protected final void engineUpdate(byte b) throws SignatureException {
    if (this.sign) {
        this.data.write(b);
    } else {
        this.verifier.update(b);
    }
}

@Override
protected final void engineUpdate(byte[] b, int off, int len) throws
SignatureException {
    if (this.sign) {
        this.data.write(b, off, len);
    } else {
        this.verifier.update(b, off, len);
    }
}

@Override
protected final byte[] engineSign() throws SignatureException {
    try {
        final byte[] digest = createMessageDigest(this.data.toByteArray());
        final BasicCryptographyService hsm =
            getSecurityModule().getBasicCryptography();
        return hsm.createSignature(digest, this.privateKey);
    } catch (Exception e) {
        throw new SignatureException(e);
    }
}
```

Implementierung Utimaco EID Schnittstelle

- Implementierung des Interfaces führt dann die Operation mit Hilfe der konkreten HSM Bibliothek aus

```
@Override
public byte[] createSignature(byte[] digest, KeyIdentifier identifier) {
    final KeyAttributes atts = createKeyAttributes(identifier);
    final MechParams params = new MechParams(MechParams.FORMAT_ASN1);
    return new CreateSignatureCryptographyService(digest, identifier, atts,
        params, getName()).result(this);
}
```

```
@Override
protected byte[] serviceResult(SynchronizingSingleCommandAuthenticator
                               authenticator)
        throws IOException, ReconnectException {
        try {
            final byte[] signature =
                authenticator.prepareUsage().sign(this.atts, this.digest,
                this.params);
            return signature;
        } catch (CryptoServerException e) {
            throw new IllegalStateException(FAILED_TO_CREATE_SIGNATURE +
                this.identifier.toString(), e);
        }
    }
```

Zu beachten

- Brainpool Kurven nicht nativ im Java (erst mit Java 11)
  - Nutzung von anderen Bibliotheken (z.B. BouncyCastle) notwendig
- Schlüssel auf HSMs sind teilweise nicht schnittstellenübergreifend verfügbar
  - Utimaco CXI und PKCS#11
- Java starke/sichere Schlüsselverfahren nach US Exportgesetz verboten
  - Bis Java 8 u151 musste Unlimited Strength Cryptography aktiviert werden
  - Ab Java 8 u151 kann das über die Property `crypto.policy` aktiviert werden
  - Ab Java 8 u162 standardmäßig aktiviert

Danke für das Interesse

Christoph Pfeifer
Entwickler robotron*e collect
Gruppenleiter „Intelligente Messsysteme“

MIT DATEN MEHR BEWEGEN.

robotron®