NutriSafe

Sicherheit in der Lebensmittelproduktion und -logistik
durch die Distributed-Ledger-Technologie

Giesecke+Devrient

NutriSafe Toolkit

– DLT-Sicherheitsbausteine –

Anbindung von PKCS#11 basierten HSM Lösungen an die Hyperledger Fabric Blockchain zur Verbesserung der Sicherheit

Ingo Rudorff

GEFÖRDERT VOM

Bundesministerium für Bildung und Forschung

Bundesministerium Landwirtschaft, Regionen und Tourismus

SIFO.de

KIRAS Sicherheitsforschung

FFG Forschung wirkt.

Dieses Dokument ist Bestandteil im **NutriSafe Toolkit**:

nutrisafe.de/toolkit

In einer Kooperation zwischen Deutschland und Österreich forschen Universitäten, Unternehmen und Behörden daran, die Lebensmittelproduktion sowie deren Logistik unter Nutzung von Distributed-Ledger-Technologie sicherer zu machen.

Das Projekt NutriSafe wird auf Österreichischer Seite innerhalb des Sicherheitsforschungs-Förderprogramms KIRAS durch das Bundesministerium für Landwirtschaft, Regionen und Tourismus (BMLRT) gefördert (Projektnummer: 867015). Auf Deutscher Seite wird das Projekt innerhalb des Programms Forschung für die zivile Sicherheit vom Bundesministerium für Bildung und Forschung (BMBF) gefördert (FKZ 13N15070 bis 13N15076).

nutrisafe.de | nutrisafe.at

Anbindung von PKCS#11 basierten HSM Lösungen an die Hyperledger Fabric Blockchain zur Verbesserung der Sicherheit

Ingo Rudorff[1]

[1]Giesecke+Devrient Mobile Security GmbH

München 2021

Giesecke+Devrient Mobile Security GmbH

Prinzregentenstr. 159

81677 München

Inhalt

1 Ausgangssituation

1.1 Das Testnetzwerk

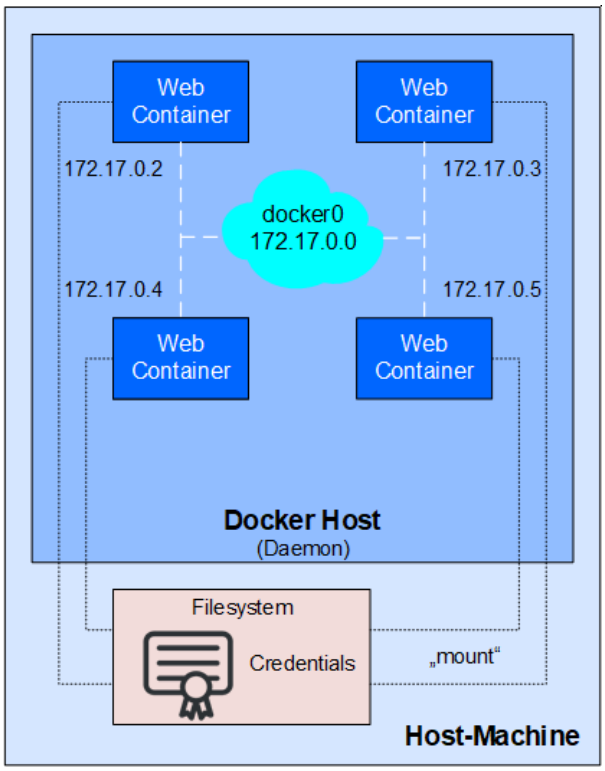
Im Rahmen des Förderprojektes NutriSafe hat die Bundeswehr Universität Neubiberg, die das deutsche Konsortium des Projektes leitet, ein Testnetzwerk - das sogenannte ***Milchnetzwerk*** - aufgesetzt.

Dieses Netzwerk basiert auf ***Hyperledger Fabric*** als Distributed Ledger Technologie (DLT). Es orientiert sich im Wesentlichen an der Hyperledger Fabric Beispiel-Implementierung. Jedoch weißt diese Beispiel-Implementierung von Hyperledger Fabric bei der Speicherung von Schlüsselmaterial große Sicherheitsschwachstellen auf.

Der Hyperledger Fabric Beispielcode basiert auf Docker Containern, die auf einem Host-Rechner mit Linux laufen und über ein Docker-Netzwerk miteinander kommunizieren können. Der Docker Daemon auf dem Host übernimmt dabei den Start der Container und stellt das interne Docker-Netzwerk bereit.

Dieses Netzwerk wurde später auf mehrere-Rechner mit Hilfe eines Multi-Host Netzwerkes aufgeteilt.

Das folgende Bild zeigt den prinzipiellen Aufbau eines Netzwerkes auf einem Host:



1.2 Sicherheitsschwachstellen

Hyperledger Fabric ist die Implementierung einer ***Permissioned DLT***. Das bedeutet, dass der Zugang und die Nutzung nur für Mitglieder möglich ist, die sich zuvor registriert und autorisiert haben. Dies geschieht über eine PKI Infrastruktur mit Zertifikaten und private/public Schlüsseln (Credentials). Die Echtheit und Unverfälschtheit wird durch Signaturen sichergestellt.

Beim Beispielnetzwerk werden die Credentials, wenn das Netzwerk aufgesetzt wird, einmalig mit dem Hyperledger-Tool ***cryptogen*** auf der Host-Machine erzeugt. Dabei werden sie direkt im Dateisystem der Host-Machine abgelegt. Die laufenden Docker-Container haben "per Default" keinen Zugriff auf das Dateisystem der Host-Machine, außer die Dateipfade werden beim Start der Container explizit "freigegeben". Dann werden nur diese speziellen Pfade mit den darin befindlichen Dateien in das Dateisystem der Container gemounted und sind damit zugreifbar.

Jeder Web-Container bekommt natürlich nur seinen Credential-Bereich (Zertifikate, public & private Keys) gemounted und sieht die Credentials der anderen Container nicht und kann auf diese auch nicht zugreifen.

Dennoch sind die Credentials aller Container ungeschützt im Dateisystem der Host-Machine abgelegt und können so angegriffen bzw. manipuliert werden.

Diese Sicherheitsschwachstelle besteht auch bei der Aufteilung auf ein Multi-Host Netzwerk.

1.3 Maßnahmen zur Erhöhung der Sicherheit

Um die Credentials effektiv vor Missbrauch und Manipulationen zu schützen, bietet sich der Einsatz eines Hardware Security Moduls (HSM) an.

Mit der PKCS#11 Schnittstelle ergeben sich die vielfältigsten Implementierungsmöglichkeiten. Die PKCS#11 Schnittstelle stellt ein Interface bereit, das einen Cryptografic Service Provider (CSP) implementiert. Dabei hat PKCS#11 einen im Vergleich zu CSP erweiterten Funktionsumfang.

Folgende Entwicklungsstufen (siehe Bild unten) wurden von Giesecke+Devrient durchgeführt:

1.3.1 Entwicklungsstufe 1

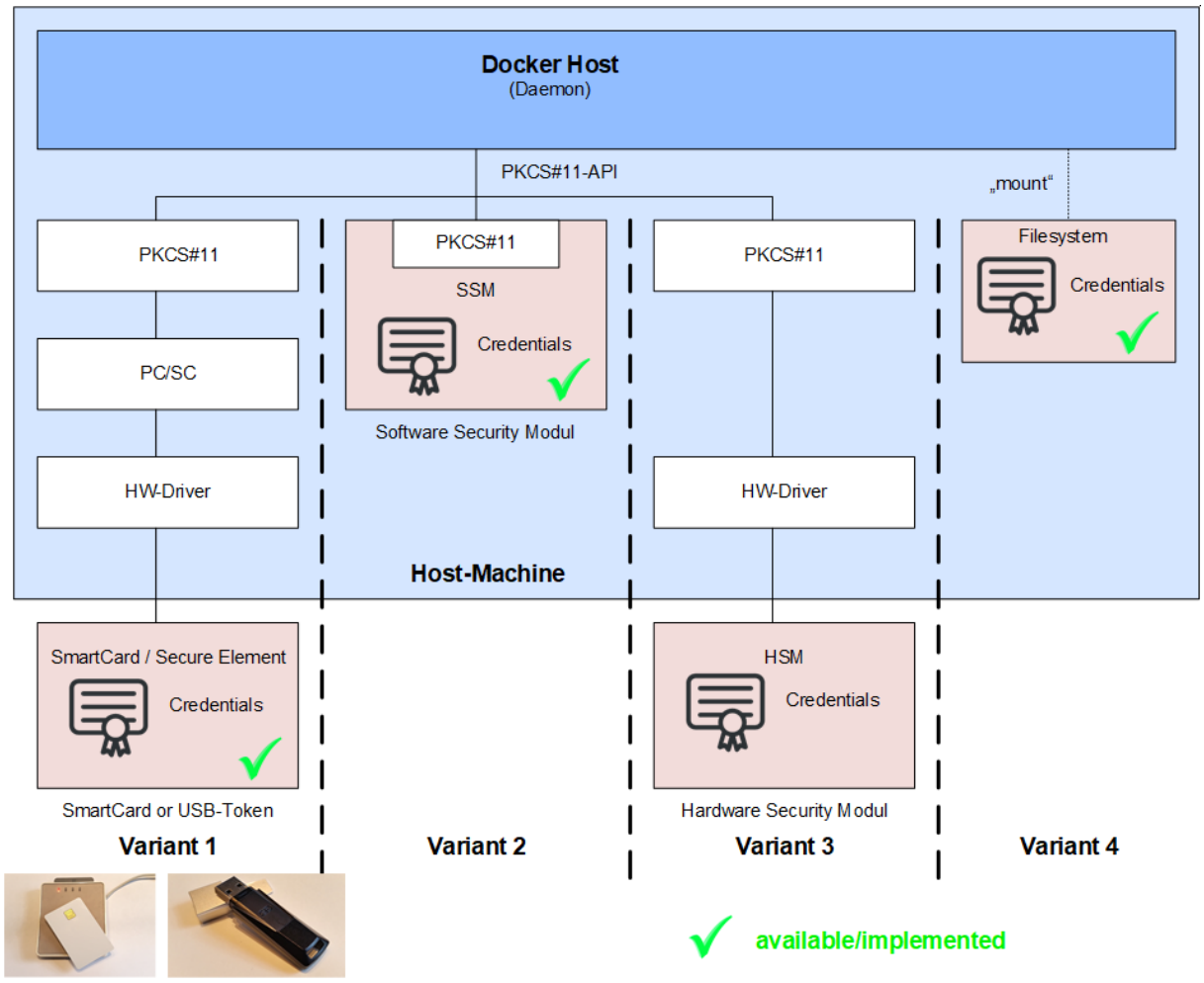
Bei der Variante 2 werden die Credentials in einem sogenannten Software HSM (SSM) verwaltet. Der Name ist dabei irreführend, da HSM ja Hardware Security Modul bedeutet.

Das SSM ist eine Software-Lösung, die eine Library mit PKCS#11 API bereitstellt. Die Credentials werden nach wie vor im Filesystem des Hosts abgelegt – aber verschlüsselt. Der Vorteil dieser Lösung ist, dass SSM relativ einfach zu erstellen und in Software (die PKCS#11 API nutzen) integrierbar sind. Damit kann dann die Entwicklung sowie alle Tests wie mit einem echten HSM schnell und kostengünstig durchgeführt werden.

1.3.2 Entwicklungsstufe 2

In der Entwicklungsstufe 2 hat Giesecke+Devrient die SSM Lösung durch eine SmartCard bzw. USB-Token ersetzt (Variante 1). Dabei kommt eine PKCS#11 fähige Java-SmartCard mit einer Middleware (OpenSC) und einem Java Applet (ISO-Applet) zum Einsatz.

Beim Nutrisafe Milchnetzwerk ist der Einsatz eines Tokens/SmartCard besonders für Peer Nodes wie z.B. einen Milchbauern oder für die Transportkette interessant, wo mit geringem Aufwand, portabel und trotzdem sicher ein Zugang zum Milchnetzwerk zur Verfügung gestellt werden soll.



Ein HSM (Variante 3) ist für die Absicherung eines großen Peer Nodes/Organisation denkbar.

2 HSM/PKCS#11

2.1 HSM

Ein HSM ist ein physikalisch eigenständiges Hardware-Device, dass dazu benutzt werden kann digitale Schlüssel zu generieren und zu speichern. Das HSM kann außerdem kryptographische Operationen ausführen.

Der Vorteil gegenüber einer Speicherung in einem Keystore oder direkt im Filesystem auf dem Host-System/Rechner ist, daß die "Secrets", d.h. die privaten Schlüssel" ausschließlich im HSM verbleiben und niemals direkt auf dem Host-System zugreifbar/sichtbar sind.

Bei Hyperledger Fabric kann ein HSM kann eingesetzt werden, um die privaten Schlüssel der Hyperledger Fabric Nodes zu generieren und zu speichern. Peer- oder Ordering-Nodes können mit Hilfe eines HSMs Transactions signieren und bestätigen (Endorsement).

Als HSM können verschiedene HSM-Varianten zum Einsatz kommen („Soft“HSM, HSM, Smartcards, USB-Token)

2.2 Public Key Cryptography Standard #11(PKCS #11)

Um auf ein HSM zugreifen zu können, wird meistens das PKCS#11 API benutzt. PKCS#11 ist ein Standard-API, das durch die "nonprofit" Organisation OASIS Open spezifiziert wurde.

Der aktuelle PKCS#11 Standard für ein Token Interface und für das Verhalten eines Tokens ist verfügbar unter PKCS#11 Standard V2.40

Hyperledger Fabric unterstützt den PKCS#11 Standard, um mit einem HSM zu kommunizieren.

Allerdings sind die Standard(Prebuild)-Hyperledger (Docker) Images so gebaut, dass der PKCS#11 Support disabled ist. Daher müssen die Images neu gebaut werden. Dieser Vorgang ist relativ ungenau beschrieben und wird daher im Folgenden detailliert erläutert.

Erschwerend kommt hinzu, daß die Dokumentationen teilweise immer noch fehlerhaft sind oder wichtige Details fehlen.

3 Realisierung

3.1 Übersicht

Die Credentials eines Teilnehmer am Hyperledger Fabric Netzwerk sollen auf einem HSM/Smartcard/Token (externer Schlüsselspeicher) sicher gespeichert sein. Zu signierende Daten werden im Betrieb über die Credentials auf dem HSM signiert. Damit können andere Teilnehmer im Netzwerk die Authentizität des Senders feststellen können und sicher sein, daß die Daten nicht verändert wurden.

Da sich die verschiedenen Arten von Schlüsselspeichern in ihrer Einrichtung unterscheiden werden sie gesondert beschrieben. Zum Beispiel ist es entscheidend, ob das Schlüsselmaterial (Public- und Private Key) nur auf dem Schlüsselspeicher generiert werden kann oder auch extern. Im Gegensatz zu Smartcards gibt es bei HSMs häufig die Option das Schlüsselmaterial extern zu generieren und es dann auf den HSM zu importieren.

3.2 Einrichtung SoftHSM

3.2.1 Installation softhsm2 auf dem Host (VM)

Am einfachsten ist es die Open Source Software **softhsm2** über die Debian Paketquellen zu installieren. Dabei muß nur beachtet werden, das die Library **libsofthsm2** auch installiert wird.

Wenn eine andere Version benötigt wird als in den Paketquellen vorhanden ist, oder bei der verwendeten Linux Version kein softhsm2 in der Distribution enthalten ist, kann das Paket auch aus den Sourcen gebaut werden.

Weiterführende Links:

https://github.com/opendnssec/SoftHSMv2
https://www.howtoforge.com/tutorial/how-to-install-and-use-softhsm-on-ubuntu-1604-lts/
https://blog.wlami.com/blog/2016/01/24/pkcs-number-11-entwicklung-ohne-hardware-softhsm-als-smartcard-oder-hsm-ersatz/

3.2.2 SoftHSM Konfiguration

Der "default" HSM-Konfigurationsfile ist nach der Installation hier gespeichert:

```
/etc/softhsm/softhsm2.conf
```

Er hat folgenden Inhalt:

```
# SoftHSM v2 configuration file

directories.tokendir = /var/lib/softhsm/tokens/
objectstore.backend = file

# ERROR, WARNING, INFO, DEBUG
log.level = ERROR

# If CKF_REMOVABLE_DEVICE flag should be set
slots.removable = false
```

Die Variable directories.tokendir bestimmt, wo die "geheimen" Daten (Tokens) des HSM gespeichert werden. Entweder werden die Daten in einer Datei oder in einer Datenbank gespeichert.

Da das Token Verzeichnis per "default" im System-Verzeichnis liegt, können unprivilegierte Anwender und Apps keine Token speichern. Daher ist es sinnvoll für den einfachen Testbetrieb ein eigenes Verzeichnis im Nutzer Bereich anzulegen und dort den Konfigurationsfile und die Token zu speichern

Die default Datei softhsm2.conf wird in folgendes Verzeichnis kopiert:

```
/home/fabric/.config/softhsm2
```

Der Inhalt wird folgendermaßen abgeändert:

```
# SoftHSM v2 "Host" configuration file

# The location where SoftHSM can store the tokens.
#directories.tokendir = /var/lib/softhsm/tokens/
directories.tokendir = /home/fabric/.config/softhsm2/tokens

# The backend to use by SoftHSM to store token objects.
objectstore.backend = file

# ERROR, WARNING, INFO, DEBUG
log.level = DEBUG

# If CKF_REMOVABLE_DEVICE flag should be set
slots.removable = false
```

Damit der HSM den neuen Konfigurationsfile benutzt, muß die Umgebungsvariable gesetzt sein:

```
SOFTHSM2_CONF=/home/fabric/.config/softhsm2/softhsm2.conf
```

z.B.: export SOFTHSM2_CONF=/home/fabric/.config/softhsm2/softhsm2.conf <softhsm command>

Mit diese Konfiguration läßt sich der SoftHSM auf dem Host-System benutzen.

3.2.2.1 SoftHSM Konfiguration für Docker Container

Damit bei Hyperledger Fabric der HSM mit der PKCS#11 API auch vom Docker Container benutzt werden kann, muss die PKCS#11-Library, das Konfigurationsfile und das Token-Verzeichnis in den Docker-Container als Volume gemounted werden.

Dies wird im Kapitel ***Nutzung SoftHSM vom Docker Container*** beschrieben.

Dabei ist zu beachten, daß die Pfadangaben im softhsm2.conf sich auf den Host beziehen und diese Pfade im Docker Container nicht vorhanden sind. Als Lösung wird ein modifizierter Konfigurationsfile vom Host in den Container gemounted.

Es wird eine Kopie des Konfigurationsfiles in dem neuen Verzeichnis /home/fabric/.config/softhsm2_docker auf dem Host mit folgendem Inhalt angelegt:

```
# SoftHSM v2 “Docker” configuration file

# The location where SoftHSM can read the token.
directories.tokendir = /etc/hyperledger/fabric/HSM_tokens

# The backend to use by SoftHSM to store token objects.
objectstore.backend = file

# ERROR, WARNING, INFO, DEBUG
log.level = DEBUG

# If CKF_REMOVABLE_DEVICE flag should be set
slots.removable = false
```

3.2.3 SoftHSM Token initialisieren

Nach der Konfiguration des SoftHSM muß das Token initialisiert werden. Das im Konfigurationsfile angegebene Verzeichnis zum Speichern der Token MUSS vorhanden sein! Sonst muss es angelegt werden.

ACHTUNG: Alle Daten im Token werden gelöscht !!!

Als Beispiel wird hier der SoftHSM für den Peer Pinzgauer mit Hilfe des Tools softhsm2-util (wird mit softhsm2 installiert) initialisiert:

```
SOFTHSM2_CONF=/home/fabric/.config/softhsm2/softhsm2.conf  softhsm2-util  --
init-token --slot 0 --label "ForPinzgauer" --so-pin 7777 --pin 12345678

The token has been initialized and is reassigned to slot 1431388730
```

Die dezimale Slot ID 1431388730 entspricht der hexadezimalen Slot ID 0x5551423A (siehe Funktionstest)

Es werden dabei 2 selbstgewählte Pins angegeben:

- 7777 Der Pin für den Security Officer (SO)
- 12345678 Pin für den "normalen" Nutzer

3.2.4 Schlüssel in den SoftHSM schreiben

Bei dem eingesetzten SoftHSM ist es möglich die bereits mit dem Hyperledger-Tool ***cryptogen*** generierten Schlüssel zu verwenden.

Beispielhaft ist das Schlüsselmaterial für den Peer Pinzgauer dargestellt:

```
├── crypto-config
│   ├── ordererOrganizations
│   │   └── unibw.de
  :   :
│   └── peerOrganizations
  :       :
│         ├── pinzgauer.de
│         │   ├── ca
│         │   │   ├── ca.pinzgauer.de-cert.pem
│         │   │   └── ea992c14f8c9ca5d….534311262885cdf605a8411_sk
│         │   ├── msp
│         │   │   ├── admincerts
│         │   │   │   └── Admin@pinzgauer.de-cert.pem
│         │   │   ├── cacerts
│         │   │   │   └── ca.pinzgauer.de-cert.pem
│         │   │   └── tlscacerts
│         │   │       └── tlsca.pinzgauer.de-cert.pem
│         │   ├── peers
│         │   │   └── peer0.pinzgauer.de
│         │   │       ├── msp
│         │   │       │   ├── admincerts
│         │   │       │   │   └── Admin@pinzgauer.de-cert.pem
│         │   │       │   ├── cacerts
│         │   │       │   │   └── ca.pinzgauer.de-cert.pem
│         │   │       │   ├── keystore
│         │   │       │   │   └── 5f5d94...6d3456_sk
│         │   │       │   ├── signcerts
│         │   │       │   │   └── peer0.pinzgauer.de-cert.pem
│         │   │       │   └── tlscacerts
│         │   │       │       └── tlsca.pinzgauer.de-cert.pem
│         │   │       └── tls
│         │   │           ├── ca.crt
│         │   │           ├── server.crt
│         │   │           └── server.key
│         │   ├── tlsca
│         │   │   ├── dfde0db4a53….e2b95df_sk
│         │   │   └── tlsca.pinzgauer.de-cert.pem
│         │   └── users
```

Das betrachtete Schlüsselmaterial für den Peer ist:

5f5d94...6d3456_sk	Private Key (eigentlich Keypair) Pinzgauer
peer0.pinzgauer.de-cert.pem	Zertifikat

Für die Nutzung dieser Credentials im HSM müssen die unten beschriebenen Schritte durchgeführt werden:

Für Tests wurde das Schlüsselmaterial aus dem generierten Baum in folgendes Verzeichnis kopiert:
/home/fabric/Workspace/Projects/Nutrisafe/G+D_Pinzgauer/test-pinzgauer/HSM-Credentials

```
Step 1:
Private Key des Node: peer0.pinzgauer.de in HSM schreiben
cd /home/fabric/Workspace/Projects/Nutrisafe/G+D_Pinzgauer/test-
pinzgauer/HSM-Credentials
pkcs11-tool --module /usr/lib/x86_64-linux-gnu/softhsm/libsofthsm2.so -l --
pin 12345678 --write-object ./5f5d94...6d3456_sk --type privkey --id
5f5d94...6d3456

Step2:
Extract pubKey aus Peer Zertifikat (.pem)
openssl x509 -pubkey -nocert -in peer0.pinzgauer.de-cert.pem >
peer0.pinzgauer.de-pubkey.pem

Step3:
Extrahierten Public Key in SoftHSM schreiben:
Der pubKey MUSS mit der gleichen ID wie der prvKey geschrieben werden !!!
pkcs11-tool --module /usr/lib/x86_64-linux-gnu/softhsm/libsofthsm2.so -l --
pin 12345678 --write-object ./peer0.pinzgauer.de-pubkey.pem --type pubkey -
-id 5f5d94...6d3456

Step4:
Wenn alles funktioniert hat, wird der Private Key 5f5d94...6d3456_sk aus
dem generierten Baum auf dem Host gelöscht.
```

3.2.5 Funktionstest

Anzeige Slots und Objects auf dem Token/HSM (mit User Authentisierung):

```
root@SwarmManager:/home/fabric# export
SOFTHSM2_CONF=/home/fabric/.config/softhsm2/softhsm2.conf
root@SwarmManager:/home/fabric# pkcs11-tool -v -L -O --pin 12345678 --
module /usr/lib/x86_64-linux-gnu/softhsm/libsofthsm2.so
vailable slots:
Slot 0 (0x5551423a): SoftHSM slot ID 0x5551423a
manufacturer: SoftHSM project
hardware ver: 2.4
firmware ver: 2.4
flags: token present
token label : ForPinzgauer
token manufacturer : SoftHSM project
token model : SoftHSM v2
token flags : login required, rng, token initialized, PIN initialized,
other flags=0x20
hardware version : 2.4
firmware version : 2.4
serial num : 962ddf77e9e5ebb7
pin min/max : 4/255
Slot 1 (0x1): SoftHSM slot ID 0x1
manufacturer: SoftHSM project
```

```
hardware ver: 2.4
firmware ver: 2.4
flags: token present
token label :
token manufacturer : SoftHSM project
token model : SoftHSM v2
token flags : login required, rng, SO PIN locked, SO PIN to be changed,
other flags=0x20
hardware version : 2.4
firmware version : 2.4
serial num :
pin min/max : 4/255
Using slot 0 with a present token (0x69e5ebb7)
Public Key Object; EC EC_POINT 256 bits
EC_POINT:
044104ba2a9684c22026df6dc9248cd87919ac627b1340ddebdfb9633c9864ef1507acec2a8
2f372d24248b0837c9ed47cc6c24432d2a2761cf7097713056df2a37bcb
EC_PARAMS: 06082a8648ce3d030107
label:
ID: 5f5d94f5a8914560ef29b4c8208e679a470c08ada0fa58ad06faf5f5b36d3456
Usage: encrypt, verify, wrap
Private Key Object; EC
label:
ID: 5f5d94f5a8914560ef29b4c8208e679a470c08ada0fa58ad06faf5f5b36d3456
Usage: decrypt, sign, unwrap
```

3.3 Einrichtung Smardcard(-HSM)

Das Ziel ist die Credentials eines Teilnehmer am Hyperledger Fabric Netzwerk auf einer Smartcard sicher zu speichern. Konkret wird das Schlüsselmaterial des Orderers, der der Organisation UniBw angehört, auf der Smartcard gespeichert.

3.3.1 Komponenten

Da die Anbindung einer Smartcard an Hyperledger Fabric ein präzises Zusammenspiel zwischen Software, Hardware-Treibern und Hardware erfordert, folgt eine genaue Auflistung der benutzten Komponenten und Versionen. Die Nutzung ist nicht „out of the box“, sie erfordert auch Codeänderungen an der beteiligten Software.

Auf diese Konstellation beziehen sich alle weiteren Beschreibungen:

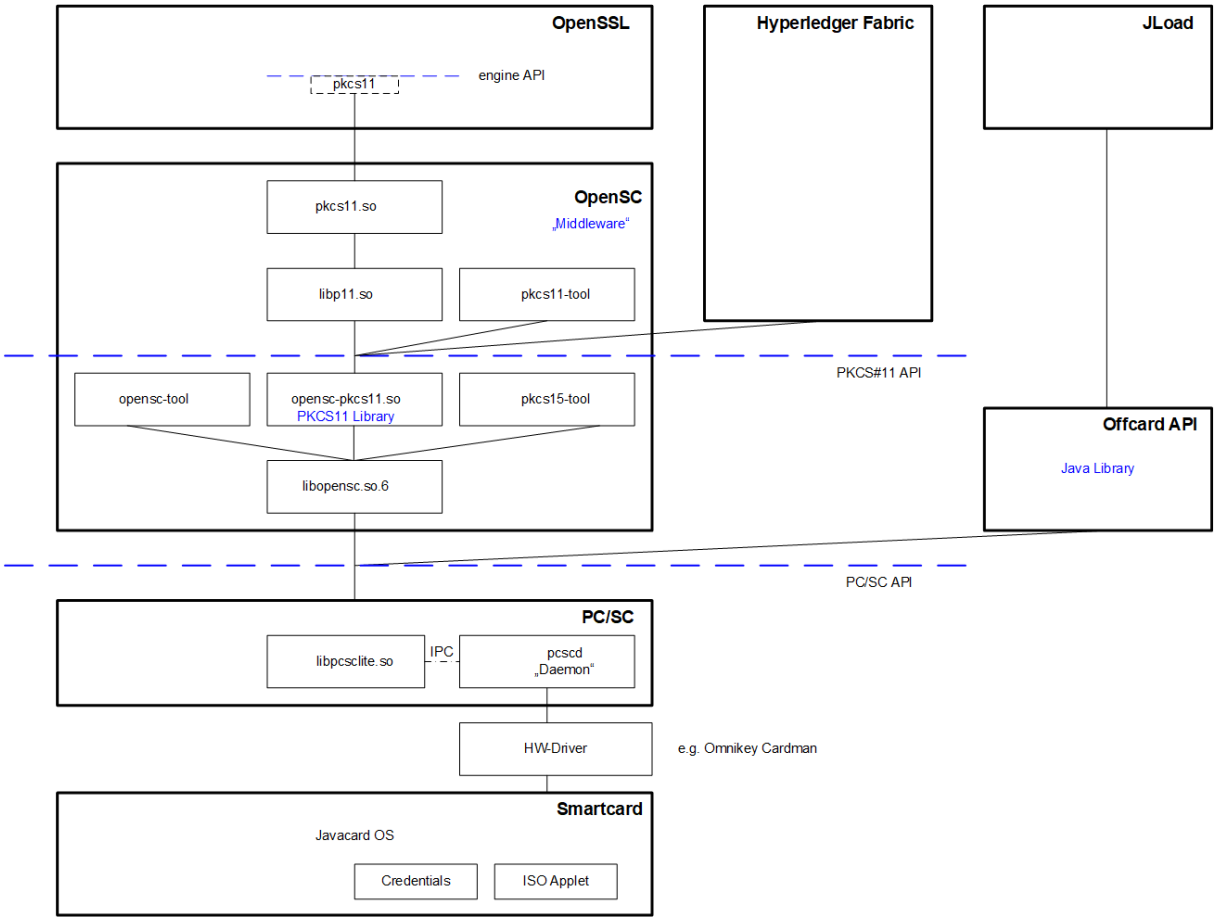
1. Giesecke+Devrient Smartcard mit Javacard OS
2. ISO Applet 0.6.1
3. OpenSC 0.21.0-rc1
4. libp11 0.4.10
5. OpenSSL 1.1.1h

3.3.2 Zusammenfassung

1. Das Schlüsselmaterial muss auf der Smartcard erzeugt werden, der Private Key verläßt niemals die Smartcard.
2. Für das Schlüsselmaterial muß ein neues Zertifikat außerhalb der Smartcard mit dem Public Key, der auf der Smartcard liegt, von der Root-CA erzeugt und signiert werden.
3. Es soll die existierende Root-CA für eine Organisation (hier für die Orderer Organisation UniBw) bestehen bleiben, da von dieser CA weitere vom Hyperledger Fabric Tool cryptogen "abgeleitete" Zertifikate existieren, d.h. signiert sind. Wenn das Schlüsselmaterial (Public- und Private Key) auf der Smartcard erzeugt wird, muß ein neues Zertifikat außerhalb der Smartcard erzeugt werden, das von der Root-CA signiert wurde. Dies wird mittels eines Certifikate Signing Request (CSR) bei der CA durchgeführt (siehe Schlüssel- und Zertifikatserzeugung).
4. Um den CSR zu erzeugen, wird OpenSSL benutzt. Die Signatur im CSR wird dabei von der Smartcard mittels des Private Keys erzeugt. Damit die Smartcard mit dem Hash SHA256 signiert müssen weitere Änderungen an der OpenSC Middleware und am ISO Applet auf der Karte gemacht werden (siehe Code Änderungen).
5. Das von der CA aus dem CSR generierte Zertifikat wird auf die Smartcard importiert.
6. Im CSR wie auch im Zertifikat kann von OpenSSL ein Subject Key Identifier (SKI) erzeugt werden (X509v3 Extention). Dieser ist per default der Hash (SHA1) über den Public Key im Zertifikat. Wenn Hyperledger Fabric das Schlüsselmaterial in der Smartcard benutzen möchte identifiziert/adressiert es die Credentials über den SKI. Leider benutzt Hyperledger nicht den SKI aus dem Zertifikat, sondern berechnet den SKI intern selbst. Dabei wird ebenfalls ein Hash über den PubKey gerechnet - allerdings ein SHA256!
   Damit Hyperledger auf den Schlüssel in der Smartcard mit der "SHA256-SKI" zugreifen

kann, ist, nachträglich die Objekt ID der Schlüssel auf der Smartcard auf diesen Hashwert zu ändern!

Das folgende Bild zeigt einen Überblick über die beteiligte Software und den Software-Stack. Dieses Übersichtsbild ist wichtig, da für die Einrichtung und den Betrieb einer Smartcard als PKCS#11 kompatibler HSM viel verschiedene Tools benötigt werden.



3.3.3 Eigenschaften und Vorbereitung des Smartcard

Eigenschaften der Smartcard

Eine Smartcard muß für den beschriebenen Einsatz mit Hyperledger Fabric bestimmte Eigenschaften erfüllen:

1. Implementierung der PKCS #11 Cryptographic Token Interface Base Specification Version 2.40
   http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/pkcs11-base-v2.40.html
2. Unterstützung der Elliptic Curve Cryptography (ECC) für folgende ECDSA Signatur: secp256r1 (aka: prime256v1, P-256)
3. Es muß möglich sein ohne vorheriges Hashen eine Signatur zu erzeugen. Diese Eigenschaft ist notwendig, da das Hashen bereits extern in OpenSSL durchgeführt wird.

Die eingesetzte Smartcard ist eine Giesecke+Devrient Javacard mit einem Java Card Betriebssystem (Java Card Expert 7). Auf diesem Betriebssystem (OS) laufen Java Applets, die entweder von Giesecke+Devrient entwickelt wurden oder als Open Source Applets zur Verfügung stehen. Das Javacard OS implementiert einen bestimmten Javacard Standard - bei der angegebenen Giesecke+Devrient Javacard: Java Card Classic Platform Specification 3.0.4 (https://www.oracle.com/java/technologies/javacard-specs-downloads.html). Dieses OS unterstützt den geforderten Signatur-Algorithmus (Signature.ALG_ECDSA_SHA_256) und bietet die Möglichkeit Signaturen ohne einen vorherigen Hash Vorgang zu berechnen (Signature.signPreComputedHash).

Ob diese Betriebssystem Eigenschaften von einem Anwender auch genutzt werden können hängt vom installierten Applet ab.

Das eingesetzte ISO-Applet (https://github.com/philipWendland/IsoApplet/) ist als fertig gebautes Applet verfügbar. Leider ist es für die Javacard API Version 2.2.2 gebaut und unterstützt damit nicht die geforderten Eigenschaften. Da das ISO-Applet aber Open Source ist, kann und muß der Source Code in der Version 0.6.1 vom angegebenen GIT Repro heruntergeladen werden, modifiziert und neu gebaut werden (siehe Code Änderungen)

Vorbereitung der Smartcard für den Betrieb

Sind die Eigenschaften der Karte erfüllt, muß sie für den ersten Einsatz vorbereitet werden.

Bei einer neuen Karte ohne installiertem Applet muß zunächst das Applet mit JLoad auf die Karte geladen und installiert werden. Alternativ kann ein anderes Ladeprogramm für Applets verwendet werden.

Wird ein Applet neu geladen und installiert, hat es noch kein Filesystem. Daher MUSS immer zuerst ein Filesystem in dem Applet angelegt werden. Beim Löschen eines existierenden Applets wird das Filesystem ebenfalls gelöscht und muß ebenfalls neu angelegt werden.

Mit dem Tool pkcs15-init wird ein PKCS#15 Filesystem angelegt auf dem später Objekte (Schlüssel, Zertifikate, Daten) permanent gespeichert werden können:

```
pkcs15-init --create-pkcs15
Using reader with a card: Identive Identive CLOUD 4500 F Dual Interface
Reader [CLOUD 4700 F Contact Reader] (53201327201374) 00 00
New User PIN.
Please enter User PIN: 123456
Please type again to verify: 123456
Unblock Code for New User PIN (Optional - press return for no PIN).
Please enter User unblocking PIN (PUK): 01234....abcdef
Please type again to verify: 01234....abcdef
```

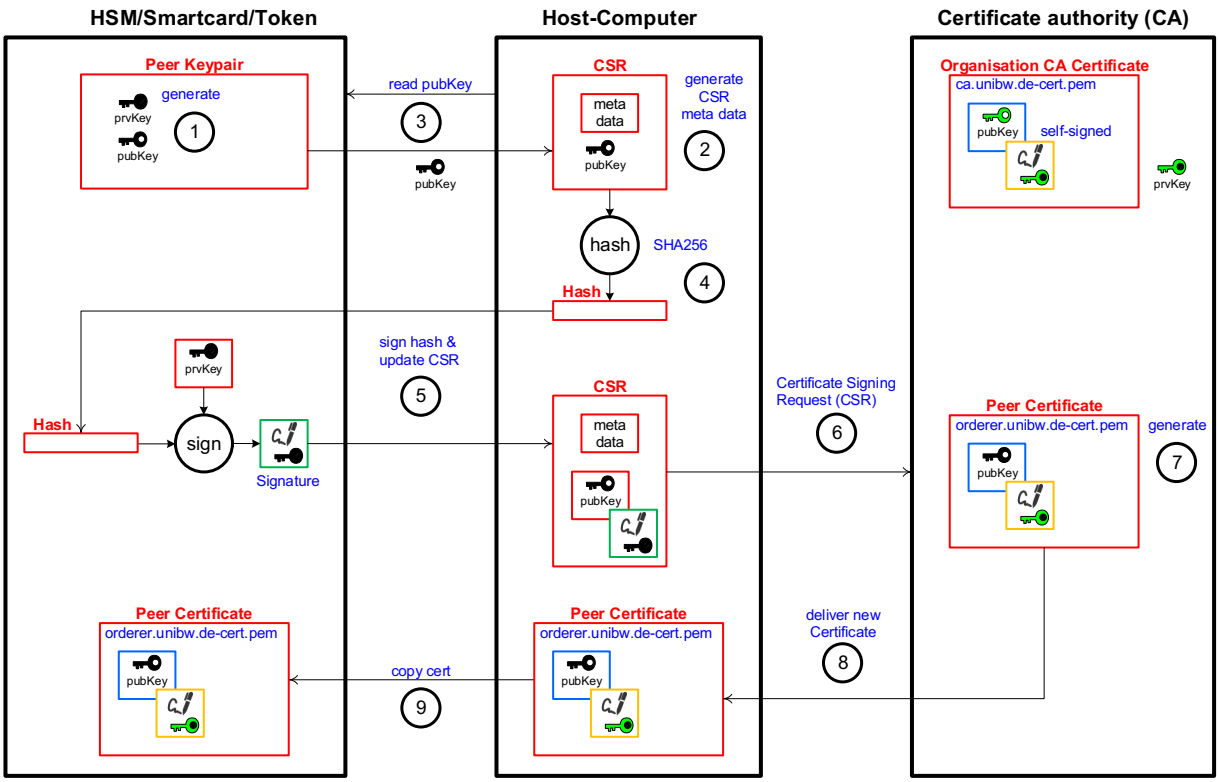
Anschließend an die Vorbereitungen kann die Schlüssel- und Zertifikatserzeugung erfolgen.

3.3.4 Schlüssel- und Zertifikatserzeugung

Es gibt sichere und unsichere Verfahren um Schlüsselmaterial (Private- und Public Key) zu erzeugen und in einem sicheren Schlüsselspeicher (HSM, Smartcard oder USB-Token) zu speichern.

1. Das Schlüsselmaterial wird auf einem Host-System mit Tools wie OpenSSL erzeugt und erst anschließend in den sicheren Schlüsselspeicher kopiert. Der Vorteil ist, daß die Erzeugung eines Zertifikates mit Hilfe eines CSRs einfach ist, da der Private Key zum Signieren des CSR auch auf dem Host-System verfügbar ist. Die Sicherheit hängt stark davon ab wie gut das Host-System auf dem die Generierung erfolgt abgesichert ist. Der Nachteil ist, das vor allem der Private Key auf einem meist relativ unsicheren System erzeugt wird und dort auch entwendet werden kann bevor oder nachdem er auf den sicheren Schlüsselspeicher kopiert wurde. Beim Einsatz des SoftHSM wurde diese Methode verwendet.
2. Das Schlüsselmaterial wird direkt auf dem sicheren Schlüsselspeicher generiert. Der Private Key verlässt dabei niemals diesen Speicher. Dieses Verfahren ist aufwendiger, speziell für die CSR Generierung. Eine Smartcard z.B. ist gar nicht in der Lage einen CSR zu generieren. Also muß der CSR mit den Meta Daten (Info über den Antragsteller) mit dem Public Key des Antragsstellers (der aus dem sicheren Schlüsselspeicher lesbar ist) ohne die Signatur des Antragsstellers auf dem Host-System vorbereitet werden. Über diesen vorbereiteten CSR wird dann auf dem sicheren Schlüsselspeicher die Signatur berechnet.
   Beim Einsatz einer Smartcard/USB-Token ist dieses Verfahren meistens sogar zwingend, da es nicht möglich ist Private Keys in den Schlüsselspeicher zu schreiben. Deshalb wird dieses Verfahren mit der Giesecke+Devrient Smartcard auch verwendet.

Am Beispiel der Schlüssel- und Zertifikatsbildung für die Nutrisafe Orderer Organisation "UniBw" ist der Ablauf im folgenden Bild nochmal detailliert dargestellt:



Schritt 1:

Das Schlüsselpaar mit der ID 707070 wird auf der Smartcard mit dem Tool pkcs11-tool generiert.

```
pkcs11-tool --module /usr/lib/x86_64-linux-gnu/opensc-pkcs11.so --pin
123456 --keypairgen --key-type EC:prime256v1 --id 707070
Using slot 0 with a present token (0x0)
Key pair generated:
Private Key Object; EC
label: Private Key
ID: 990099
Usage: sign, derive
Public Key Object; EC EC_POINT 256 bits
EC_POINT:
0441044997088c592b7a21e21acf79cb92b115d2874b32fb38225d28e13d8072a4a38e80415
2ef6393d290a374c9eb620ae4ef57ab471c3db324cd64dc5464695248c2
EC_PARAMS: 06082a8648ce3d030107
label: Private Key
ID: 990099
Usage: verify, derive
```

Die folgenden Schritte 2 bis 4 werden während einer OpenSSL Kommandobearbeitung ausgeführt.

Schritt 2:

Auf dem Host-System wird mit OpenSSL ein unvollständiger "Rumpf"-CSR mit Metadaten (Zertifikatsversion, Name des Antragsstellers, ...) erzeugt.

Schritt 3:

Anschließend wird der Public Key aus der Smartcard gelesen und in den "Rumpf"-CSR eingefügt.

Schritt 4:

Über den "Rumpf"-CSR wird ein SHA256 Hash berechnet.

Schritt 5:

Dieser Hash wird von der Smartcard signiert und in den CSR final eingebaut.

```
PKCS11_MODULE_PATH vor dem Aufruf setzen:
Dieser Schritt ist notwendig, da OpenSSL die Angabe des Parameters
MODULE_PATH=Library im folgenden Befehl anscheinend ignoriert!
export PKCS11_MODULE_PATH=/usr/lib/opensc-pkcs11.so

Eine externe PKCS#11 "Engine" in OpenSSL registrieren:
Wenn in folgenden Befehlen eine registrierte Engine angegeben wird, weiß
OpenSSL, das z.B. bestimmte Crypto-Operationen wie z.B. der Signiervorgang
des CSR Hashes nicht intern per Software, sondern durch die entsprechende
"Engine" ausgeführt werden sollen. Über diesen Mechanismus kann OpenSSL
bestimmte Funktionalität in eine externe Hardware - hier die Smartcard -
auslagern.
openssl engine dynamic -pre SO_PATH:/usr/local/lib/engines-1.1/libpkcs11.so
-pre ID:pkcs11 -pre LIST_ADD:1 -pre LOAD -pre MODULE_PATH:/usr/lib/opensc-
pkcs11.so

CSR mit Hilfe von OpenSSL, der Middleware OpenSC und der Smartcard
generieren:
Das Ergebnis ist der CSR mit dem Namen ec_Smartcard_csr707070.pem
openssl req -engine pkcs11 -new -key id_707070 -keyform engine -out
ec_Smartcard_csr707070.pem
```

Um sicherzustellen, dass der CSR fehlerfrei erstellt wurde, kann er folgendermaßen mit OpenSSL geprüft und angezeigt werden:

```
openssl req -text -verify -in ec_Smartcard_csr707070.pem
verify OK
Certificate Request:
Data:
Version: 1 (0x0)
Subject: C = DE, ST = Bavaria, L = Munich, O = UniBw Neubiberg, OU = R&D,
CN = Nutrisafe Orderer
Subject Public Key Info:
Public Key Algorithm: id-ecPublicKey
Public-Key: (256 bit)
pub:
04:6c:42:94:35:0d:d7:94:9b:fe:73:00:d8:d3:ee:
81:99:1d:5d:09:00:6f:73:bc:0a:69:e7:e2:f7:19:
d5:e7:e3:67:c3:6a:97:91:d5:e9:6e:2d:45:06:dc:
dc:6e:2c:93:e3:93:d2:3a:01:c2:f3:ee:de:1b:e7:
0d:c4:fd:10:41
ASN1 OID: prime256v1
```

```
NIST CURVE: P-256
Attributes:
unstructuredName :unibw
challengePassword :witz
Signature Algorithm: ecdsa-with-SHA256
30:44:02:20:40:15:82:c3:59:32:f9:a9:37:5b:da:22:cb:c4:
05:66:ab:1c:aa:49:10:56:c6:61:5e:fb:01:18:2c:f3:b3:a5:
02:20:56:59:48:97:76:a6:3c:fa:89:d4:d1:fb:11:e7:c7:9e:
a7:3b:6f:82:71:5a:52:12:df:ee:89:ec:c6:6c:0f:d9
-----BEGIN CERTIFICATE REQUEST-----
MIIBYTCCAQgCAQAwfDELMAkGA1UEBhMCREUxDDAKBgNVBAgMA0JheTEMMAoGA1UE
BwwDTXVjMRMwEQYDVQQKDApCYXN0ZWxidWRlMRAwDgYDVQQLDAdidWRlLmRlMQww
CgYDVQQDDANSJkQxHDAaBgkqhkiG9w0BCQEWDWFkbWluQGJ1ZGUuZGUwWTATBgcq
hkjOPQIBBggqhkjOPQMBBwNCAARsQpQ1DdeUm/5zANjT7oGZHV0JAG9zvApp5+L3
GdXn42fDapeR1eluLUUG3NxuLJPjk9I6AcLz7t4b5w3E/RBBoCowEwYJKoZIhvcN
AQkCMQYMBGJ1ZGUwEwYJKoZIhvcNAQkHMQYMBHdpdHowCgYIKoZIzj0EAwIDRwAw
RAIgQBWCw1ky+ak3W9oiy8QFZqscqkkQVsZhXvsBGCzzs6UCIFZZSJd2pjz6idTR
+xHnx56nO2+CcVpSEt/uiezGbA/Z
-----END CERTIFICATE REQUEST-----
```

Schritt 6:

Der Certificate Signing Request wird an die CA weitergeleitet.

Schritt 7:

Die CA erstellt nach Prüfung des CSRs ein neues Zertifikat und signiert dieses mit ihrem eigenen Private Key. Bei der Prüfung wird verifiziert, ob der Public Key des CSR zur Signatur des CSR passt. In einer echten CA wird zusätzlich noch die Identität des Antragsstellers über andere Wege geprüft (Ausweis, …). Im Nutrisafe Projekt sind die Credentials des CA (Zertifikat und Private Key) bekannt und werden von OpenSSL, in der Funktion einer CA, benutzt, um das Zertifikat zu generieren.

```
Zertifikat mit OpenSSL als CA generieren:
Die Datei ca.unibw.de-cert.pem enthält das Zertifikat, die Datei
ca.unibw.de-keypair.pem enthält den Private Key der CA. Das neue Zertifikat
hat den Namen ec_Smartcard_cert707070.pem.
openssl x509 -req -in ec_Smartcard_csr707070.pem -CA ca.unibw.de-cert.pem -
CAkey ca.unibw.de-keypair.pem -CAcreateserial -out
ec_Smartcard_cert707070.pem
```

Nach der Generierung sollte vor dem Einsatz das Zertifikat noch mit OpenSSL angezeigt und geprüft werden.

```
openssl x509 -noout -text -in ./ec_Smartcard_cert707070.pem
Certificate:
Data:
Version: 1 (0x0)
Serial Number:
50:e2:0c:b2:86:0b:44:43:c0:91:b6:ca:44:35:44:55:22:46:2b:5b
Signature Algorithm: ecdsa-with-SHA256
Issuer: C = US, ST = California, L = San Francisco, O = unibw.de, CN =
ca.unibw.de
```

```
Validity
Not Before: Nov 13 16:17:31 2020 GMT
Not After : Dec 13 16:17:31 2020 GMT
Subject: C = DE, ST = Bavaria, L = Munich, O = UniBw Neubiberg, OU = R&D,
CN = Nutrisafe Orderer
Subject Public Key Info:
Public Key Algorithm: id-ecPublicKey
Public-Key: (256 bit)
pub:
04:6c:42:94:35:0d:d7:94:9b:fe:73:00:d8:d3:ee:
81:99:1d:5d:09:00:6f:73:bc:0a:69:e7:e2:f7:19:
d5:e7:e3:67:c3:6a:97:91:d5:e9:6e:2d:45:06:dc:
dc:6e:2c:93:e3:93:d2:3a:01:c2:f3:ee:de:1b:e7:
0d:c4:fd:10:41
ASN1 OID: prime256v1
NIST CURVE: P-256
Signature Algorithm: ecdsa-with-SHA256
30:46:02:21:00:a4:e9:6f:ff:62:cc:5a:3e:15:e3:06:59:a3:
1a:37:0d:ea:0f:f8:0f:84:67:76:6e:bb:0f:57:8d:11:d0:bc:
3e:02:21:00:aa:00:ab:ad:16:43:9a:9f:dc:a5:76:98:2b:67:
29:4c:23:20:5a:29:60:d7:9c:ee:bc:74:d5:4b:53:78:be:7f
```

Schritt 8:

Das Zertifikat wird an den Antragsteller ausgeliefert.

Schritt 9:

Das neue erstellte Zertifikat wird vom Host-System in die Smartcard kopiert.

```
Zertifikat mit pkcs11-tool als neues Objekt mit der ID 707070 auf die
Smartcard schreiben:
Dabei muss die Objekt ID die gleiche sein wie die des vorher erzeugten
Schlüsselpaares.
pkcs11-tool --module /usr/lib/opensc-pkcs11.so --pin 123456 --write-
object ./ec_Smartcard_cert707070.pem --type cert --id 707070
```

3.3.5 Anpassung Credendials für Hyperledger Fabric

Im Abschnitt Schlüssel- und Zertifikatserzeugung ist der genaue Ablauf für die Erzeugung beschrieben. Auf der Smartcard befinden sich nach Abschluß ein Zertifikat und der Public- sowie Private Key, alle mit der ID 707070.

Leider läßt sich das so erzeugte Schlüsselmaterial in Hyperledger Fabric nicht nutzen. Für den Zugriff auf das Schlüsselmaterial berechnet Hyperledger Fabric nämlich selbst den Subject Key Identifier (SKI), unabhängig vom SKI, der im erzeugten Zertifikat steht. Bei der Erzeugung eines Zertifikats kann über die X509v3 Extentions ein SKI in Zertifikat mit generiert werden - leider ist die Art wie der SKI erzeugt wird aber sehr begrenzt. Der SKI kann bei der Zertifikatsgenerierung durch OpenSSL entweder "hard coded" auf einen festen Wert gesetzt werden oder auf Hash gesetzt werden. Hash bedeutet hier immer, das ein SHA1 über den Public Key gerechnet wird und als SKI in das Zertifikat eingesetzt wird.

Hyperledger Fabric berechnet aber einen SHA256 über den Public Key und verwendet beim Zugriff auf das Schlüsselmaterial diesen Wert als ID. Deshalb muß die ID in der Smartcard für alle 3 Objekte nachträglich und auch vor der Nutzung mit Hyperledger folgendermaßen geändert werden:

Schritt 1:

Public Key aus der Smartcard auslesen:

```
pkcs11-tool --module /usr/lib/opensc-pkcs11.so --pin 123456 --read-object --id 707070 --type pubkey --output-file ec_Smartcard_pubkey707070.der
Using slot 0 with a present token (0x0)
```

Anzeige mit OpenSSL

```
openssl ec -inform DER -pubin -in ec_Smartcard_pubkey707070.der -text -noout
read EC key
Public-Key: (256 bit)
pub:
04:49:97:08:8c:59:2b:7a:21:e2:1a:cf:79:cb:92:
b1:15:d2:87:4b:32:fb:38:22:5d:28:e1:3d:80:72:
a4:a3:8e:80:41:52:ef:63:93:d2:90:a3:74:c9:eb:
62:0a:e4:ef:57:ab:47:1c:3d:b3:24:cd:64:dc:54:
64:69:52:48:c2
ASN1 OID: prime256v1
NIST CURVE: P-256
```

Schritt 2:

Hash SHA256 über den Public Key berechnen

```
openssl dgst -sha256 ec_Smartcard_pubkey707070.der
SHA256(ec_Smartcard_pubkey707070.der)=
a6dc775a30335342aaf0234bb59005a9412590119c0e6049ef38a2fb79d50ef5
```

Dieser Wert ist die neue ID ist das Schlüsselmaterial.

Schritt 3:

IDs des Zertifikats, des Public- und des Private-Keys auf den Wert des Hashes ändern

```
Zertifikat ID ändern:
pkcs11-tool --module /usr/lib/opensc-pkcs11.so -l --pin 123456 --type cert --id 707070 --set-id
a6dc775a30335342aaf0234bb59005a9412590119c0e6049ef38a2fb79d50ef5

Public Key ID ändern:

pkcs11-tool --module /usr/lib/opensc-pkcs11.so -l --pin 123456 --type pubkey --id 707070 --set-id
a6dc775a30335342aaf0234bb59005a9412590119c0e6049ef38a2fb79d50ef5

Private Key ID ändern:

pkcs11-tool --module /usr/lib/opensc-pkcs11.so -l --pin 123456 --type privkey --id 707070 --set-id
a6dc775a30335342aaf0234bb59005a9412590119c0e6049ef38a2fb79d50ef5
```

3.3.6 Code Änderungen

Der oben beschriebene Ablauf ist ohne Code-Änderungen an verschiedenen Teilen der beteiligten Software z.Z. leider nicht durchführbar. Aus diesem Grund wurden die meisten der beteiligten Software Pakete aus den Sourcen neu gebaut und mit Debug-Ausgaben versehen. An folgender Software sind Modifikationen erforderlich, die den oben beschriebenen Ablauf ermöglichen.

3.3.6.1 libp11

Die libp11 Software gehört als "Unterrepository" zum OpenSC Projekt. libp11 besteht aus zwei PKCS#11 Wrapper Libraries, die einen schmaler Layer über dem eigentlichen PKCS#11 API implementieren und helfen sollen, Anwendungen, die PKCS#11 Funktionalität nutzen, zu vereinfachen.

- pkcs11.so
  Ist das PKCS#11 "Engine" Plugin für OpenSSL, das es einfach macht Krypto-Funktionalität nicht intern in Software auszuführen, sondern in einer externen Implementierung - hier in ein Hardware-Implementierung. Sobald die PKCS11 Engine bei OpenSSL registriert ist, können Bearbeitungen via einer Middleware wie OpenSC über PC/SC an einen Kartenleser und damit an eine Smartcard "ausgelagert" werden.
- libp11.so
  Implementiert das eigentliche Wrapper Interface. Aus der SW Stack-Sicht bildet es die Schnittstelle nach oben zum Engine Plugin pkcs11.so und nach unten zur OpenSC Library opensc-pkcs11.so

Zuerst wird die ZIP Datei heruntergeladen und entpackt oder das GitHub Projekt gecloned.

https://github.com/OpenSC/libp11

Änderungen/Erweiterungen

Bei der Generierung eines Certificate Signing Requests (CSR) gibt es folgende Fehlermeldungen:

```
Shell:
p11_ec.c:454 pkcs11_ecdsa_sign() rv after CRYPTOKI_call C_Sign = 0x54
139930014323840:error:82068054:PKCS#11 module:pkcs11_ecdsa_sign:Function
not supported:p11_ec.c:458:
```

Die Suche nach der Ursache führt zu Änderungen im Software Paket libp11 und OpenSC (siehe OpenSC).

Der zu nutzende Mechanismus wird überschrieben:

- Alt: mechanism.mechanism = CKM_ECDSA
- Neu: mechanism.mechanism = CKM_ECDSA_SHA1

```
File:      src/p11_ec.c
Funktion:  pkcs11_ecdsa_sign()

memset(&mechanism, 0, sizeof(mechanism));
```

```
~~mechanism.mechanism = CKM_ECDSA~~
mechanism.mechanism = CKM_ECDSA_SHA1;
```

Build

Als erstes werden alle Abhängigkeiten geprüft und wenn notwendig heruntergeladen und installiert.

https://github.com/OpenSC/libp11

```
libp11-libp11-0.4.10.zip entpacken
cd <libp11 Dir>
./bootstrap
```

```
./configure

: :
libp11 has been configured with the following options:

Version: 0.4.10
libp11 directory: /usr/local/lib
Engine directory: /usr/local/lib/engines-1.1
Default PKCS11 module:
API doc support: no

Host: x86_64-pc-linux-gnu
Compiler: gcc
Preprocessor flags:
Compiler flags: -g -O2
Linker flags:
Libraries: -ldl

OPENSSL_CFLAGS: -I/usr/local/include
OPENSSL_LIBS: -L/usr/local/lib -lcrypto
```

Abschließend wird, nachdem die oben beschriebenen Code Änderungen durchgeführt wurden, libp11 gebaut und im System installiert:

```
make

make install
```

Beim make install werden die statische und die dynamische Library in das Verzeichnis kopiert:

/usr/local/lib/engines-1.1

- pkcs11.so
- pkcs11.la

Die Library libp11.so wird ins Verzeichnis /usr/local/lib kopiert.

3.3.6.2 OpenSC

Für den Zugriff auf Krypto-Token (HSMs, Smartcards, USB-Token, ...) wurde von OASIS die generische API PKCS#11 spezifiziert (http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/pkcs11-base-v2.40.html), die von den meisten Betriebssystemen unterstützt wird.

OpenSC implementiert dieses API für den Zugriff auf Smartcards als Library. Zusätzlich stellt das OpenSC Projekt zahlreiche Tools zur Verfügung um Smartcards über die Kommandozeile zu verwalten. Diese Tools können mit der eigenen Library oder auch mit kompatiblen Libraries von anderen Herstellern zusammenarbeiten.

Für die Nutzung im Projekt Nutrisafe, speziell für den Einsatz mit Hyperledger Fabric, muß die Software neu selbst gebaut und modifiziert werden. Daher wurde das Tool OpenSC nicht aus den Debian Paketquellen installiert, sondern aus den Sourcen neu gebaut. Sollte OpenSC schon über die Paketquellen installiert worden sein, so empfiehlt sich zuerst eine Deinstallation folgender Pakete:

- opensc
- opensc-pkcs11

Zuerst wird die ZIP Datei heruntergeladen und entpackt oder das GitHub Projekt gecloned.

https://github.com/OpenSC/OpenSC

Zusatzinfos/Abhängigkeiten

- Wenn OpenSC aus dem Standard-Debian Paket installiert wird, liegt die eigene Library unter /usr/lib/x86_64-linux-gnu/opensc-pkcs11.so
- Wenn OpenSC selbst gebaut wird, liegt die Library per default unter /usr/lib/opensc-pkcs11.so
- Wird OpenSC oder eines der Tools wie pkcs11-tool ohne --module <Lib> aufgerufen, wird immer die eigene Library verwendet.
- OpenSC unterstützt PIVKey und andere PIV Karten im "Read Only" Mode. Deshalb können keine Zertifikate auf die Karte geschrieben werden, sondern nur gelesen und benutzt werden.

Je nach Einsatz benötigt OpenSC noch weitere Software-Pakete:

- PCSC-Lite (runtime and development)
- OpenSSL (optional, runtime and development)
- OpenPACE (optional, runtime and development)
- GNU Readline (optional, runtime and development)

Installationanweisung:

```
sudo apt-get install pcscd libccid libpcsclite-dev libssl-dev libreadline-dev autoconf automake build-essential docbook-xsl xsltproc libtool pkg-config
```

Konfiguration

OpenSC hat eine Konfigurationsdatei, die bei Bedarf geändert werden kann. Zum Beispiel ist es für Tests hilfreich den Parameter "debug_file" auszukommentieren und auf eine eigene Datei zu konfigurieren. In dieser Datei werden viele Informationen über den Ablauf in OpenSC und Daten für der Zugriff gelogged.

Beispiel default Konfigurationsfile /etc/opensc/opensc.conf:

```
app default {
      # debug = 3;
      debug_file = opensc-debug.txt;
      framework pkcs15 {
            # use_file_caching = true;
      }
}
```

Änderungen/Erweiterungen

Bei der Generierung eines Certificate Signing Requests (CSR) gibt es folgende Fehlermeldungen:

```
Shell:
p11_ec.c:454 pkcs11_ecdsa_sign() rv after CRYPTOKI_call C_Sign = 0x54
139930014323840:error:82068054:PKCS#11 module:pkcs11_ecdsa_sign:Function
not supported:p11_ec.c:458:

OpenSC Logfile:
[opensc-pkcs11] card-isoApplet.c:1126:isoApplet_set_security_env: IsoApplet
only supports raw ECDSA.: -1408 (Not supported)
```

Die Suche nach der Ursache führte zu Änderungen im Software Paket OpenSC und libp11 (siehe libp11).

In OpenSC werden die Flags für den zu nutzenden Algorithmus (env.algorithm_flags) überschrieben:

- Alt: 0x00000200 = SC_ALGORITHM_ECDSA_HASH_SHA1
- Neu: 0x00100000 = SC_ALGORITHM_ECDSA_RAW

```
File:      src/libopensc/pkcs15-sec.c
Funktion:  sc_pkcs15_compute_signature()

senv.algorithm_flags = SC_ALGORITHM_ECDSA_RAW;        // Overwrite
sc_log(ctx, "sc_pkcs15_compute_signature(), overwite: env.algorithm_flags =
0x%8.8x", senv.algorithm_flags);

r = use_key(p15card, obj, &senv, sc_compute_signature, tmp, inlen, out,
outlen);
```

Ausgabe nach Änderung im OpenSC Logfile:

```
[opensc-pkcs11] pkcs15-sec.c:713:sc_pkcs15_compute_signature:
sc_pkcs15_compute_signature(), flags:0x00000200,
alg_info->flags:0x80100200, pad_flags:0x00000000, sec_flags:0x00000200

[opensc-pkcs11] pkcs15-sec.c:747:sc_pkcs15_compute_signature:
sc_pkcs15_compute_signature(), overwite: env.algorithm_flags = 0x00100000

: :

[opensc-pkcs11] card-isoApplet.c:1080:isoApplet_set_security_env:
isoApplet_set_security_env(), start

[opensc-pkcs11] card-isoApplet.c:1120:isoApplet_set_security_env:
isoApplet_set_security_env(), env->algorithm = SC_ALGORITHM_EC

[opensc-pkcs11] card-isoApplet.c:1121:isoApplet_set_security_env:
isoApplet_set_security_env(), env->algorithm_flags = 1048576

: :

[opensc-pkcs11] card-isoApplet.c:1175:isoApplet_set_security_env: returning
with: 0 (Success)
```

Build

Als nächstes werden alle Abhängigkeiten geprüft und wenn notwendig heruntergeladen und installiert.

https://github.com/OpenSC/OpenSC

```
tar xfvz opensc-*.tar.gz
cd opensc-*
./bootstrap
```

Per default werden folgende Verzeichnisse für die Konfiguration und als Installationsverzeichnis benutzt:

- /usr (Installation)
- /usr/local/etc (Konfiguration)

Die default Verzeichnisse können beim Aufruf von ./configure geändert werden:

```
./configure --prefix=/usr --sysconfdir=/etc/opensc

OpenSC has been configured with the following options:
Version: 0.21.0-rc1
User binaries: /usr/bin
Configuration files: /etc/opensc
OpenSSL support: yes
OpenSSL secure memory: no
PC/SC support: yes
PC/SC default provider: libpcsclite.so.1
PKCS11 default provider: /usr/lib/opensc-pkcs11.so
PKCS11 onepin provider: /usr/lib/onepin-opensc-pkcs11.so
```

Abschließend wird, nachdem die oben beschriebenen Code Änderungen durchgeführt wurden, OpenSC gebaut:

```
make
```

Es wird das Verzeichnis mit Links auf einige Libraries angelegt:

/usr/lib/pkcs11

3.3.6.3 ISO-Applet

Benötigte Tools

Das ISO-Applet ist ein Java-Applet, das zum Bau eine Java Card SDK Umgebung benötigt. Der Build setzt Apache Ant voraus.

- openjdk-11-jdk-headless
- ant

Als erster Schritt werden die Sourcen aus dem Repro gecloned.

```
git clone https://github.com/philipWendland/IsoApplet.git

git submodule init && git submodule update
Submodul 'ext/sdks' (https://github.com/martinpaljak/oracle_javacard_sdks)
für Pfad 'ext/sdks' in die Konfiguration eingetragen.
Klone nach '/home/fabric/Workspace/Projects/IsoApplet/ext/sdks' ...
Submodul-Pfad: 'ext/sdks': '494164ea8470f49f44be19f49dab3af7c83e39e2'
ausgecheckt
```

Änderungen/Erweiterungen

Die folgenden Änderungen am Applet stellen keine kommerzielle Lösung dar, sondern dienen ausschließlich dazu, die gewünschte Funktionalität schnell mit geringem Aufwand zu implementieren!

Das ISO-Applet unterstützt nur den Signatur-Algorithmus ALG_ECDSA_SHA (SHA1) und führt vor der Signaturberechnung immer ein Hashen der Eingangsdaten durch. Dieses Verhalten ist für die Erzeugung eines CSRs und für den Einsatz in Hyperledger Fabric ungeeignet, da die Daten außerhalb der Smartcard gehashed werden und die Smartcard nur noch die Signatur berechnen soll.

Der vorhandene Signatur-Algo wird als durch ALG_ECDSA_SHA_256 (SHA256) ersetzt. Im vorliegenden Fall ist der neue Algo nur für das Anlegen eines Signatur-Objektes mit ausreichend großem Speicher notwendig (für einen 32 Byte Hash). Das Applet wird trotzdem mit dem Algorithmus ALG_ECDSA_SHA1 angesprochen. Die Implementierung ist aber folgendermaßen im Vergleich zum Original geändert:

- Es werden max. 32 Byte Eingangsdaten verarbeitet
- Die Eingangsdaten werden nicht gehashed (da sie bereits der Hash sind!)
- Die Eingangsdaten werden nur signiert

```
File:       src/net/pwendland/javacard/pki/isoapplet/IsoApplet.java
Funktion:   IsoApplet()

protected IsoApplet() {

      :           :

try {

//ecdsaSignature = Signature.getInstance(Signature.ALG_ECDSA_SHA, false);

ecdsaSignature = Signature.getInstance(Signature.ALG_ECDSA_SHA_256, false);
```

File: src/net/pwendland/javacard/pki/isoapplet/IsoApplet.java
Funktion: computeDigitalSignature()

```
private void computeDigitalSignature(APDU apdu) throws ISOException {

      :           :

case ALG_ECDSA_SHA1:

// Get the key - it must be a EC private key,
// checks have been done in MANAGE SECURITY ENVIRONMENT.
ECPrivateKey ecKey = (ECPrivateKey) keys[currentPrivateKeyRef[0]];

// Initialisation
ecdsaSignature.init(ecKey, Signature.MODE_SIGN);

short recvLen = apdu.setIncomingAndReceive();
offset_cdata = apdu.getOffsetCdata();

// Generates the signature of the precomputed hash because we have already
SHA256 hashed data!
sigLen = ecdsaSignature.signPreComputedHash(buf, offset_cdata, recvLen,
buf, (short) 0);

apdu.setOutgoingAndSend((short) 0, sigLen);

break;
```

Konfiguration

Damit die Änderungen am Code des ISO-Applets bei Bau nicht zu Fehlern führen, muß noch das Ziel-Java-API konfiguriert werden. Wenn z.B. ein zu "altes" Ziel API konfiguriert ist (wie bei ISO-Applet - default API 2.2.2), dann sind manche Methoden oder Definitionen nicht vorhanden und der Build ist fehlerhaft!

Die Konfiguration welches API für das Ziel verwendet wird befindet sich in der XML-Datei build.xml

```
Die Zeile muß entsprechend angepasst werden:

~~<cap targetsdk="ext/sdks/jc222_kit" aid="f2:76:a2:88:bc:fb:a6:9d:34:f3:10"
output="IsoApplet.cap" sources="src" version="1.0">~~
<cap targetsdk="ext/sdks/jc304_kit" aid="f2:76:a2:88:bc:fb:a6:9d:34:f3:10"
output="IsoApplet.cap" sources="src" version="1.0">
```

Die Giesecke+Devrient Javacard Smart Cafe Expert (SCE 7) implemetiert Javacard 3.0.4!

Build

ISO-Applet mit Hilfe von Ant neu bauen:

```
ant
Buildfile: /home/fabric/Workspace/Projects/IsoApplet/build.xml
```

```
[get] Destination already exists (skipping):
/home/fabric/Workspace/Projects/IsoApplet/ant-javacard.jar

dist:
[cap] INFO: using JavaCard 3.0.5 SDK in
/home/fabric/Workspace/Projects/IsoApplet/ext/sdks/jc305u3_kit
[cap] INFO: targeting JavaCard 2.2.2 SDK in
/home/fabric/Workspace/Projects/IsoApplet/ext/sdks/jc222_kit
[cap] INFO: Setting package name to net.pwendland.javacard.pki.isoapplet
[cap] Building CAP with 1 applet from package
net.pwendland.javacard.pki.isoapplet (AID: F276A288BCFBA69D34F310)
[cap] net.pwendland.javacard.pki.isoapplet.IsoApplet
F276A288BCFBA69D34F31001
[compile] Compiling files from
/home/fabric/Workspace/Projects/IsoApplet/src
[compile] Compiling 14 source files to /tmp/jccpro4786043255311220653
[convert] [ INFO: ] Converter [v3.0.5]
[convert] [ INFO: ] Copyright (c) 1998, 2018, Oracle and/or its affiliates.
All rights reserved.
[convert]
[convert]
[convert] [ INFO: ] conversion completed with 0 errors and 0 warnings.
[verify] Verification passed
[cap] CAP saved to /home/fabric/Workspace/Projects/IsoApplet/IsoApplet.cap

BUILD SUCCESSFUL
Total time: 5 seconds
```

3.4 Hyperledger Fabric Basic Installation

Wer Hyperledger Fabric mit dem in der Dokumentation beschriebenen Beispielcode benutzen möchte, braucht Hyperledger nicht neu bauen, sondern nur den Beispielcode herunterladen.
Für die Installation wird ein Curl-Script aufgerufen, dass das aktuellste Production-Release installiert. Dabei werden die **Hyperledger Fabric Docker Image** heruntergeladen und installiert. Zusätzlich werden auch Beispiele, die sogenannten ***fabric-samples***, gespeichert, die auch die Plattform spezifischen Binaries (Tools + **Hyperledger Fabric Images**) enthalten. Die Images werden im Linux Dateisystem installiert, die Beispiele in das aktuelle Verzeichnis beim Aufruf des Scripts.

Step1:
Zunächst wechselt man in das Verzeichnis, in dem die Hyperledger Beispiele installiert werden sollen.

```
cd /home/fabric/Workspace/Projects/HyperledgerFabric/fabric-samples_V1.4
```

Step2:

Entweder wird die letzte Release installiert oder eine spezifische Version (hier V1.4.8)

```
curl -sSL https://bit.ly/2ysbOFE | bash -s
curl -sSL http://bit.ly/2ysbOFE | bash -s -- 1.4.8
```

3.5 Hyperledger Fabric Basic Build

Die folgenden Beschreibungen gehen davon aus, das eine Linux VM und die notwendigen Tools installiert wurden.

Syntax für Pfadangaben: „…“ entspricht dem Verzeichnis in das Hyperledger Fabric kopiert/gecloned wurde. Zum Beispiel:

```
/home/fabric/fabric-build_V1.4.x/src/github.com/hyperledger/fabric
```

3.5.1 Vorbereitungen/Zusatzinformationen

Wenn Hyperledger neu gebaut werden soll (z.B. mit PKCS#11 Support), sollte als Vorbereitung zuerst immer die PKCS#11 Implementierung, also das HSM, eingerichtet und auf Funktionstüchtigkeit geprüft worden sein (siehe Einrichtung eines HSMs). Unabhängig von der Build Variante werden anschließend die Hyperledger Fabric Sourcen heruntergeladen bzw. das Repro gecloned.

3.5.2 Download der Hyperledger Fabric Sourcen

Im Folgenden wird beschrieben, wie Hyperledger Fabric vom GitHub Repro gecloned wird.

Schritt 1:

Da hier Hyperledger Fabric V1.4.x gebaut werden soll, wird als Beispiel folgendes Projektverzeichnis angelegt:

```
/home/fabric/Workspace/Projects/HyperLedgerFabric/fabric-build_V1.4.x
```

Dieses Projektverzeichnis entspricht einem Go Workspace, der durch die Umgebungsvariable $GOPATH referenziert wird. Jeder Workspace besteht aus 2 bzw. 3 Ordnern.

1. bin (von go install kompilierte Binärdateien)
2. src (hier befindet sich der Quellcode)
3. pkg (optional für die Aufnahme von Bibliotheken und Ähnlichem)

Schritt 2:

Als nächstes wird ein Unterverzeichnis angelegt, in das die Sourcen gecloned werden:

```
mkdir -p $GOPATH/src/github.com/hyperledger
```

Schritt 3:

In das angelegte Unterverzeichnis wechseln und das Repro clonen:

```
cd $GOPATH/src/github.com/hyperledger
git clone https://github.com/hyperledger/fabric.git
Klone nach 'fabric' ...
remote: Enumerating objects: 12, done.
remote: Counting objects: 100% (12/12), done.
remote: Compressing objects: 100% (11/11), done.
remote: Total 129216 (delta 3), reused 2 (delta 1), pack-reused 129204
Empfange Objekte: 100% (129216/129216), 103.86 MiB | 1.01 MiB/s, Fertig.
Löse Unterschiede auf: 100% (87900/87900), Fertig.
```

Die Sourcen werden unter $GOPATH/src/github.com/hyperledger/fabric angelegt (siehe folgendem Verzeichnisbaum):

```
├── bccsp
├── ci
├── cmd
├── common
├── core
├── devenv
├── discovery
├── docs
├── examples
├── gossip
├── idemix
├── images
├── integration
├── libs
├── msp
├── orderer
├── peer
├── protos
├── release
```

```
├── release_notes
├── sampleconfig
├── scripts
├── token
├── unit-test
└── vendor
```

3.5.3 Go Konfiguration

Da ein Großteil des Hyperledger Fabric Codes in Go geschrieben ist, muß für den späteren Bau von Hyperledger Fabric die Go Buildumgebung konfiguriert werden.

Es müssen 3 Umgebungsvariablen für die Go Umgebung gesetzt bzw. erweitert werden.

Am Einfachsten werden sie in der ./bashrc eingetragen:

#GOROOT ist das Verzeichnis, indem das Go Package auf dem Host System (hier Debian in der VM) installiert ist.

#GOPATH ist das Arbeitsverzeichnis des aktuellen Go Projekts.

```
export GOROOT=/usr/local/go
```

Um verschiedene Go Installationen parallel ausprobieren zu können, kann einfach das Installationsverzeichnis umbenannt werden und dann mit dem neuen Namen hier selektiert werden

```
export GOROOT=/usr/local/go_1.14.2
```

Der GOPATH wird auf den Pfad gesetzt, wo z.B. das Hyperledger Fabric Projekt selbst gebaut wird oder wo Go Applikationen gebaut werden.

```
export GOPATH=/home/fabric/Workspace/Projects/HyperLedgerFabric/fabric-
build_V1.4.x
```

Abschließend wird noch der PATH erweitert:

Für den Fall, dass man mit verschiedenen plattform spezifischen Tool- oder Hyperledger-Image Varianten arbeitet, kann zuerst der Pfad z.B auf eine andere fabric-samples Version gesetzt werden

```
PATH=$PATH:/home/fabric/Workspace/Projects/HyperLedgerFabric/fabric-
samples_V1.4/bin
```

Als letztes werden noch die Go Pfade hinzugefügt

```
export PATH=$PATH:$GOPATH/bin:$GOROOT/bin
```

3.5.4 Unterschied „native binary“ und „docker version image“

Ergänzende Zusatzinfos unter:

https://stackoverflow.com/questions/56764476/difference-between-hyperledger-fabric-native-setup-and-hyperledger-fabric-docker/56765106#56765106

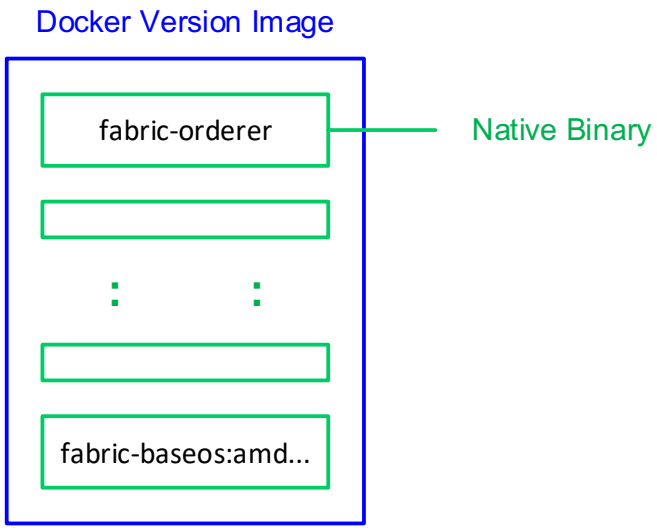
https://stackoverflow.com/questions/56819137/where-to-include-core-yaml-in-hyperledger-fabric/56820397#56820397

Beim Bau von Hyperleger wird der Makefile im Hyperledger Root-Verzeichnis benutzt:

In diesem Makefile stehen für das Target „peer“ 2 Build-Anweisungen:

```
.PHONY: peer
peer: $(BUILD_DIR)/bin/peer
peer-docker: $(BUILD_DIR)/image/peer/$(DUMMY)
```

Es werden 2 Binaries gebaut. Der Grund ist, das Hyperledger Fabric Binaries auch native auf einem Linux System ohne Docker lauffähig sind (wurde noch nicht probiert!). Die erste build Anweisung erzeugt genau dieses „native binary“, die zweite ein „docker version image“,ein Docker Image mit Layern aller benötigten Tools und einem „Base OS“.



Je nachdem wo Änderungen gemacht werden, muß das Native Binary und das Docker Version Image oder nur das Docker Version Image gebaut werden.

Die Binaries werden beim Build folgendermaßen abgelegt:

Native Binaries (pre-built executables):

```
…/.build/bin
├── chaintool
├── configtxgen
├── configtxlator
├── cryptogen
├── discover
├── idemixgen
├── orderer
```

```
└──── peer
```

Docker Version Images:

```
…/.build/image
├──── buildenv
│     ├──── Dockerfile   <== Das ist die Bauanweisung (Dockerfile) zum Bau
│     └──── payload          des Containers (aus Basis-Image)
│           ├──── gotools.tar.bz2
│           └──── protoc-gen-go
├──── ccenv
│     ├──── Dockerfile
│     └──── payload
│           ├──── chaintool
│           ├──── goshim.tar.bz2
│           └──── protoc-gen-go
├──── orderer
│     ├──── Dockerfile
│     └──── payload
│     ├──── orderer
│     └──── sampleconfig.tar.bz2
├──── peer
│     ├──── Dockerfile
│     └──── payload
│           ├──── peer
│           └──── sampleconfig.tar.bz2
└──── tools
      └──── Dockerfile
```

3.5.5 Build

Im "Basic Build" wird der Bau des Hyperledger Fabric Frameworks in der Default-Konfiguration (ohne Änderungen an der Konfiguration und an den Sourcen) beschrieben. Das bedeutet, daß für den Blockchain Crypto Service Provider (BCCSP) die in Hyperledger implementierte Software bccsp/sw konfiguriert und genutzt wird und somit die Credentials ungeschützt im Filesystem des Host-Systems abgelegt werden.

Für die Anbindung eines SoftHSM bzw. HW-Devices wie einer Smartcard/USB-Token sind Änderungen erforderlich, die in den nächsten Kapiteln beschrieben sind. Für den SoftHSM siehe *Build mit PKCS#11 Support für SoftHSM*.

Nach dem Download der Sourcen wird zunächst das Framework ohne Änderungen gebaut.

Leider führen die Unit-Test beim ersten Bauen meistens zu Fehlern, was aber teilweise (Tools und Unit-Tests) ignoriert werden kann.

Manchmal fehlen Tools, die noch installiert werden müssen. Siehe auch https://hyperledger-fabric.readthedocs.io/en/release-1.4/dev-setup/devenv.html)

Der Build wird mit folgenden Parametern ausgeführt:

```
make dist-clean all
```

Der Build Vorgang kann eine Weile dauern, da alle benötigten Base-Images vom Docker-Hub geladen werden.

Wenn beim Bau aufgrund von Fehlern „persistent states“ entstanden sind, die nicht zu beheben sind, hilft oft folgende Build Anweisung:

```
make clean all
```

Wichtig ist, das folgende Ausgaben im Build vorhanden sein:

```
Building docker peer-image
docker build  -t hyperledger/fabric-peer .build/image/peer
Sending build context to Docker daemon  48.71MB
Step 1/7 : FROM hyperledger/fabric-baseos:amd64-0.4.18
 ---> c256a6aad46f
      :           :
Successfully built c56915d4218f
Successfully tagged hyperledger/fabric-peer:latest
      :           :
Building docker orderer-image
docker build  -t hyperledger/fabric-orderer .build/image/orderer
Sending build context to Docker daemon  40.98MB
Step 1/9 : FROM hyperledger/fabric-baseos:amd64-0.4.18
 ---> c256a6aad46f
      :           :
Successfully built 5388d5859d05
Successfully tagged hyperledger/fabric-orderer:latest
```

Nach dem Bau von Peer und Orderer werden noch „dockerized gotools“ gebaut. Hier tritt aus unerklärlichen Gründen bei Giesecke+Devrient folgender Fehler auf:

```
Building github.com/vektra/mockery/cmd/mockery -> mockery
package github.com/vektra/mockery/cmd/mockery: cannot find package
"github.com/vektra/mockery/cmd/mockery" in any of:
      /opt/go/src/github.com/vektra/mockery/cmd/mockery (from $GOROOT)
      /opt/gotools/obj/src/github.com/vektra/mockery/cmd/mockery (from
$GOPATH)
gotools.mk:59: recipe for target 'gotool.mockery' failed
make[1]: *** [gotool.mockery] Error 1
```

Damit bricht der Build ab. Die Hyperledger Fabric Tools wie auch die Peer und Orderer Images sind aber einwandfrei gebaut!

Welche Images wann gebaut wurden zeigt der Docker Befehl:

```
docker image ls
```

3.6 Hyperledger Fabric Build mit PKCS#11 Support

Für die Anbindung eines SoftHSM bzw. Hardware-Devices wie einer Smartcard/USB-Token sind Änderungen/Ergänzungen notwendig, die im Folgenden beschrieben sind. Auch unterscheiden sich die Konfiguration und evtl. Codeänderungen bei den verschiedenen PKCS#11 Implementierungen.

3.6.1 Konfigurations-Varianten

Abhängig von der Art der PKCS#11 Konfiguration, die benutzt werden soll, wird der Build unterschiedlich vorbereitet.

Um die Konfiguration der Images zu beeinflussen, gibt es 2 verschiedene Möglichkeiten:

1. Überschreiben der "default" Konfiguration über Umgebungsvariablen im eigenen yaml Konfigurationsfile, der beim Start des Containers mit docker-compose angegeben wird.
2. Änderung der "default" Konfiguration im Verzeichnis .../sampleConfig

Per default wird beim Bau eines Images folgende Datei mit in das Image kopiert und bei Start des Containers entpackt und ausgewertet:

```
…/.build/image/orderer/payload/sampleconfig.tar.bz2
```

Sie enthält die default Dateien configtx.yaml, core.yaml, orderer.yaml und default Zertifikate.

Diese Dateien können für die PKCS#11 Konfiguration entweder „statisch“ vor dem Bau im Template in den Sourcen direkt geändert werden und sind dann schon im Archiv `sampleconfig.tar.bz2` enthalten, oder die Dateien werden nicht geändert und verbleiben im Originalzustand. Bei der 2. Variante wird die default Konfiguration dynamisch - über die Umgebungsvariablen im yaml File - erst beim Start des Containers überschrieben.

Beispiel für das Quell-Verzeichnis, das für die Erzeugung von `sampleconfig.tar.bz2` für das Peer Image benutzt wird:

```
├─── configtx.yaml
├─── core.yaml
├─── msp
│     ├─── admincerts
│     │     └─── admincert.pem
│     ├─── cacerts
│     │     └─── cacert.pem
│     ├─── config.yaml
│     ├─── keystore
│     │     └─── key.pem
│     ├─── signcerts
│     │     └─── peer.pem
│     ├─── tlscacerts
│     │     └─── tlsroot.pem
│     └─── tlsintermediatecerts
│           └─── tlsintermediate.pem
```

```
└─── orderer.yaml
```

3.7 Hyperledger Fabric Build für SoftHSM

3.7.1 Orderer

Als Beispiel wird der Bau des Orderers mit HSM Support beschrieben.

3.7.1.1 Konfigurations-Variante „statisch“

Schritt 1: Änderung der Container Konfigurations-Templates

Diese Dateien (configtx_yaml, core.yaml, orderer.yaml) sind im Verzeichnis .../fabric/sampleConfig abgelegt. Bei Bau des Docker Images werden diese Dateien in ein komprimiertes TAR-Archiv gepackt und dieses dann in das Verzeichnis .../fabric/.build kopiert. Dieses wird anschließend noch nach .../fabric/.build/image/orderer/payload kopiert.

Dieses TAR-File wird beim Bau des Docker-Images ins Image kopiert. Durch Änderungen an den Konfigurations-Template-Dateien können Konfigurationänderungen des Orderers direkt beim Bau durchgeführt werden.

```
…/sampleconfig/orderer.yaml

General:
        :           :
    BCCSP:
        # Default specifies the preferred blockchain crypto service
        # provider
        # to use. If the preferred provider is not available, the software
        # based provider ("SW") will be used.
        # Valid providers are:
        #  - SW: a software based crypto provider
        #  - PKCS11: a CA hardware security module crypto provider.
        Default: PKCS11

        # Settings for the PKCS#11 crypto provider (i.e. when DEFAULT:
        # PKCS11)
        PKCS11:
            # Location of the PKCS11 module library
            Library: /etc/hyperledger/fabric/HSM_lib/libsofthsm2.so
            # Token Label
            Label: ForFabric
            # User PIN
            Pin: "98765432"     <== hier mit "" !!!
            Hash: SHA2
            Security: 256
            Immutable: false
```

Der Eintrag unter SW: wird komplett gelöscht!

Das Label und der Pin müssen individuell angepaßt werden. Im angegebenen Library Path erwartet der Container die Middleware Library des HSM.

Dieser Schritt wird nur einmalig vor der ersten Build durchgeführt!

Step 2: Änderung am Dockerfile Template

Das Template für den Dockerfile (Dockerfile.in), der beim Bau des Orderers benutzt wird, liegt im Verzeichnis

```
.../fabric/images/orderer
```

Dieser File wird beim Build Prozess zuerst in das Verzeichnis .../fabric/.build/image/orderer kopiert und anschließend beim Build benutzt. Durch Änderungen am Template kann man damit einfach weitere Befehle beim Bau des Images ausführen.

Damit der Docker-Container auf den HSM zugreifen kann, benötigt der Container eine Sqlite Datenbank-Library.

Als eine mögliche Lösung wird diese vom Host beim Bau des Docker-Containers mit Hilfe des Dockerfile in das Image kopiert.

Zusätzliche Anweisung im Dockerfile:

```
COPY libsqlite3.so.0 /usr/lib/
```

Die Library muß ins gleiche Verzeichnis kopiert werden, in dem sich auch der Dockerfile befindet. Dies wird durch eine Änderung im zentralen Hyperledger Fabric Makefile bewerkstelligt:

```
$(BUILD_DIR)/image/%/Dockerfile: images/%/Dockerfile.in
	mkdir -p $(@D)
	cp libs/libsqlite3.so.0 $(@D)    <== neu eingefügt !!
	@cat $< \
```

Die Datei libsqlite3.so.0 selbst wird vom Host Rechner aus dem Verzeichnis /usr/lib/x86_64-linux-gnu (bei Debian) in ein neu angelegtes Verzeichnis im Hyperledger Fabric Verzeichnis .../libs mit dem Namen libsqlite3.so.0 kopiert. Ist die Datei ein Link wird das Ziel kopiert und umbenannt.

3.7.2 Peer Node

Als Beispiel wird der Bau des Peers Pinzgauer mit HSM Support beschrieben.

3.7.2.1 Konfigurations-Variante „dynamisch“

Schritt 1: Änderung am Dockerfile Template

Wie unter ***Konfigurations-Variante „statisch“*** beschrieben.

3.7.3 Build

Zuerst wird das Native Binary gebaut:

```
GO_TAGS=pkcs11 make -B orderer
```

Anschließend das Docker Image:

```
GO_TAGS=pkcs11 make -B orderer-docker
```

Der Parameter -B bewirkt, daß immer gebaut wird.

Test, ob Image neu gebaut und im System registriert wurde

```
fabric@SwarmManager:~/Workspace/Projects/HyperLedgerFabric/fabric-
build_V1.4.x/src/github.com/hyperledger/fabric$ docker images

REPOSITORY                    TAG                         IMAGE ID       CREATED          SIZE
hyperledger/fabric-orderer    amd64-1.4.7-snapshot-e1db6   c52dd6b71956   22 seconds ago   121MB
hyperledger/fabric-orderer    amd64-latest                c52dd6b71956   22 seconds ago   121MB
hyperledger/fabric-orderer    latest                      c52dd6b71956   22 seconds ago   121MB
```

Neues Image taggen

```
docker tag c52dd6b71956 hyperledger/fabric-orderer:softhsm_orderer
```

Wenn eine lauffähige Version der Images „archiviert“ werden soll, d.h. nicht bei jedem Build überschieben werden soll, ist es hilfreich das gebaute Image zu taggen.

In den yaml Files beim Start der Container kann das Tag folgendermaßen mitgegeben werden:

```
image: "hyperledger/fabric-orderer:${IMAGE_TAG}"
```

3.8 Hyperledger Fabric Build für Smartcard(-HSM)

Als Beispiel wird der Bau des Orderers mit PKCS#11 Smartcard-Support beschrieben.

3.8.1 Orderer

3.8.1.1 Konfigurations-Variante “statisch”

Schritt 1: Änderung der Container Konfigurations-Templates

Diese Dateien (configtx_yaml, core.yaml, orderer.yaml) sind im Verzeichnis .../fabric/sampleConfig abgelegt. Bei Bau des Docker Images werden diese Dateien in ein komprimiertes TAR-Archiv gepackt und dieses dann in das Verzeichnis .../fabric/.build kopiert. Dieses wird anschließend noch nach .../fabric/.build/image/orderer/payload kopiert.

Dieses TAR-File wird beim Bau des Docker-Images ins Image kopiert. Durch Änderungen an den Konfigurations-Template-Dateien können Konfigurationsänderungen des Orderers direkt beim Bau durchgeführt werden.

```
…/sampleconfig/orderer.yaml
```

```
General:
        :           :
    BCCSP:
        # Default specifies the preferred blockchain crypto service
        # provider
```

```
        # to use. If the preferred provider is not available, the software
        # based provider ("SW") will be used.
        # Valid providers are:
        #  - SW: a software based crypto provider
        #  - PKCS11: a CA hardware security module crypto provider.
        Default: PKCS11

        # Settings for the PKCS#11 crypto provider (i.e. when DEFAULT:
        # PKCS11)
        PKCS11:
            # Location of the PKCS11 module library
           Library: /etc/hyperledger/fabric/HSM_lib/opensc-pkcs11.so
            # Token Label
            Label: ForFabric
            # User PIN
            Pin: "123456"     <== hier mit "" !!!
            Hash: SHA2
            Security: 256
            Immutable: false
```

Im Gegensatz zum SoftHSM wird beim Smartcard-Einsatz die OpenSC Middleware eingesetzt. Deshalb muß die Library auf die OpenSC Library opensc-pkcs11.so geändert werden.

Der Eintrag unter SW: wird komplett gelöscht!

Das Label und der Pin müssen individuell angepaßt werden. Im angegebenen Library Path erwartet der Container die Middleware Library des HSM.

Dieser Schritt wird nur einmalig vor der ersten Build durchgeführt!

Step 2: Änderung am Dockerfile Template

Das Template für den Dockerfile (Dockerfile.in), der beim Bau des Orderers benutzt wird, liegt im Verzeichnis

```
.../fabric/images/orderer
```

Dieser File wird beim Build Prozess zuerst in das Verzeichnis .../fabric/.build/image/orderer kopiert und anschließend beim Build benutzt. Durch Änderungen am Template kann man damit einfach weitere Befehle beim Bau des Images ausführen.

Damit der Docker-Container auf die Smartcard zugreifen kann, benötigt der Container folgende Änderungen im Dockerfile:

```
# for OpenSC lib
COPY libopensc.so.6 /lib
COPY opensc.conf /tmp
ENV OPENSC_CONF /tmp/opensc.conf
```

```
# install PC/SC + Tools
RUN apt-get -y update && apt-get install libpcsclite1
RUN apt-get -y install pcsc-tools
RUN apt-get -y install pcscd
```

1. Die von opensc-pkcs11.so benötigte Library libopensc.so.6 befindet sich zwar im gleichen Verzeichnis auf dem Host - wird aber nicht gefunden, da opensc-pkcs11.so die Library im Verzeichnis /lib des Containers sucht, wo sie von Docker nicht gefunden wird. Als Lösung wird die Library ins (selbst angelegte) Lib Verzeichnis kopiert und beim Bau des Containers dann in den Container kopiert. Sind die Dateien Links wird das Ziel kopiert und umbenannt.
   Die Dateien befinden sich auf dem Host unter /usr/lib/
2. Optional: Damit auch OpenSC Debug-Informationen in eine Datei auf dem Container geschrieben werden, muß die OpenSC Konfigurationsdatei opensc.conf so abgeändert werden, das die Variable "debug_file" auf ein im Container vorhandenes Verzeichnis zeigt. Hier wurde das für alle User schreibbare Verzeichnis /tmp gewählt. Durch Setzen der Umbebungsvariable OPENSC_CONF findet OpenSC den neuen Konfigurationsfile auch im Container.

   ```
   app default {
       debug = 3;
       debug_file = /tmp/opensc-debug.txt;
       framework pkcs15 {
           # use_file_caching = true;
       }
   }
   ```

3. Die Middleware OpenSC setzt ein PC/SC API mit entsprechender Implementierung voraus, um über einen HW-Treiber auf die Smartcard zugreifen zu können (siehe Einrichtung eines HSM/Smartcard). Deshalb wird beim Containerstart PC/SC und benötigte Tools im Container installiert.

Folgende Dateien müssen ins gleiche Verzeichnis kopiert werden, in dem sich auch der Dockerfile befindet. Dies wird durch eine Änderung im zentralen Hyperledger Fabric Makefile bewerkstelligt:

```
$(BUILD_DIR)/image/%/Dockerfile: images/%/Dockerfile.in
	mkdir -p $(@D)
	 cp libs/libopensc.so.6 $(@D)          <== neu eingefügt !!
	 cp libs/opensc.conf $(@D)             <== neu eingefügt !!
	@cat $< \
```

3.8.1.2 Code-Änderungen

Damit beim Hochlauf der Orderer auf PC/SC und damit auf die Smartcard zugreifen kann, muss der Daemon pcscd vor oder beim Ordererstart gestartet werden.

Das stellt ein Problem dar, da im Dockerfile immer nur ein Programm über ENTRYPOINT oder CMD angegeben werden kann. Der Start eines Shell-Scripts wäre eine Lösung, das dann erst

den PC/SC Daemon und anschließend den Orderer started (Versuche nur eine bash zu starten statt den Orderer waren nicht erfolgreich - es wurde trotzdem immer der Orderer gestartet). Diese Vorgehensweise hat auch Nachteile. Vermutlich würde die Kommunikation zwischen Container und Docker Daemon nicht mehr funktionieren. Siehe auch folgende Links dazu:

https://riptutorial.com/de/docker/example/2700/unterschied-zwischen-entrypoint-und-cmd
https://www.ctl.io/developers/blog/post/dockerfile-entrypoint-vs-cmd

Die Ausführung im Dockerfile mittels RUN ["service pcscd start"] ist auch nicht möglich, da während des Image-Build kein OS läuft, bei dem der Service gestartet werden kann! Eine Variante mit CMD [service pcscd start && orderer] läßt sich zwar bauen, funktioniert aber nicht! Abhilfe schafft eine Änderung im Hochlauf/Start des Orderers, in dem jetzt der Daemon gestartet wird:

Änderung im Modul:

```
.../fabric/orderer/common/server/main.go

func Main() {
    cmd := exec.Command("service", "pcscd", "start")
    fmt.Printf("server.Main() start pcscd and waiting for it to finish...\n")
    err := cmd.Run()
    if err != nil {
        fmt.Printf("server.Main() start pcscd finished, err = %v\n")
    } else {
        fmt.Printf("server.Main() start pcscd finished successful!\n")
    }

    :                  :
```

Ausgabe beim Start des Orderers:

```
Creating orderer.unibw.de ...
: :
Creating orderer.unibw.de ... done
Attaching to orderer.unibw.de
orderer.unibw.de | main.main() - called from runtime.main
orderer.unibw.de | server.Main() - called from main.main
orderer.unibw.de | server.Main() start pcscd and waiting for it to
finish...
orderer.unibw.de | server.Main() start pcscd finished successful!
```

3.8.2 Build

Zuerst wird das Native Binary gebaut. Der Parameter -B bewirkt, daß immer gebaut wird.

```
GO_TAGS=pkcs11 make -B orderer
```

Anschließend das Docker Image:

```
GO_TAGS=pkcs11 make -B orderer-docker
```

Test, ob Image neu gebaut und im System registriert wurde

```
fabric@SwarmManager:~/Workspace/Projects/HyperLedgerFabric/fabric-
build_V1.4.x/src/github.com/hyperledger/fabric$ docker images

REPOSITORY                    TAG                         IMAGE ID       CREATED          SIZE
hyperledger/fabric-orderer    amd64-1.4.7-snapshot-e1db6   c52dd6b71956   22 seconds ago   121MB
hyperledger/fabric-orderer    amd64-latest                c52dd6b71956   22 seconds ago   121MB
hyperledger/fabric-orderer    latest                      c52dd6b71956   22 seconds ago   121MB
```

3.9 Nutzung des SoftHSM vom Docker Container

3.9.1 Orderer

3.9.1.1 Konfigurations-Variante „statisch“

Der Orderer wird mittels Docker-Compose über folgenden Aufruf gestartet:

```
docker-compose -f docker_compose_orderer_unibw.yaml up -d orderer.unibw.de
```

Der Konfigurationsfile docker_compose_orderer_unibw.yaml wird folgendermaßen erweitert:

```
Version: "3.5"

networks:
  OverlayNetwork:
    external: true
    driver: overlay
    attachable: true

services:
  orderer.unibw.de:
    container_name: orderer.unibw.de
    image: "hyperledger/fabric-orderer:${IMAGE_TAG}"
    environment:
      - SOFTHSM2_CONF=/etc/hyperledger/fabric/HSM_config/softhsm2.conf
      - FABRIC_LOGGING_SPEC=info
      - ORDERER_GENERAL_LISTENADDRESS=0.0.0.0
      - ORDERER_GENERAL_GENESISMETHOD=file
      - ORDERER_GENERAL_GENESISFILE=/etc/hyperledger/configtx/genesis.block
      - ORDERER_GENERAL_LOCALMSPID=UnibwMSP
      - ORDERER_GENERAL_LOCALMSPDIR=/etc/hyperledger/msp/orderer/msp
    working_dir: /opt/gopath/src/github.com/hyperledger/fabric/orderer
    command: orderer
    ports:
      - 7050:7050
    volumes:
      - $PROJECT_DIR/configTransactions/:/etc/hyperledger/configtx
      - $PROJECT_DIR/creatingCryptoMaterial/crypto-
config/ordererOrganizations/unibw.de/orderers/orderer.unibw.de/:/etc/hyperledger/ms
p/orderer
      ### mount the directory from the host where the PKCS#11 Library
      ### libsofthsm2.so can be found
      - /usr/lib/x86_64-linux-gnu/softhsm:/etc/hyperledger/fabric/HSM_lib
      ### The HSM config directory
      - /home/fabric/.config/softhsm2_docker:/etc/hyperledger/fabric/HSM_config
      ### The HSM token directory
      - /home/fabric/.config/softhsm2/tokens:/etc/hyperledger/fabric/HSM_tokens
    networks:
```

```
      - OverlayNetwork
```

Die HSM config und token Verzeichnisse müssen individuell angepaßt werden.

3.9.2 Peer Node

3.9.2.1 Konfigurations-Variante „dynamisch“

Der Peer Pinzgauer wird mittels Docker-Compose über folgenden Aufruf gestartet:

```
docker-compose -f create_pinzgauer_SoftHSM.yaml up -d
```

Der Konfigurationsfile create_pinzgauer_SoftHSM.yaml up wird folgendermaßen erweitert:

```
version: "3.5"
```

```
networks:
  OverlayNetwork:
    external: true
    driver: overlay
    attachable: true

services:
  peer0.pinzgauer.de:
    container_name: peer0.pinzgauer.de
    # Take defined build image version
    image: "hyperledger/fabric-peer:${IMAGE_TAG}"
    privileged: true
    environment:
      - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
      - CORE_PEER_ID=peer0.pinzgauer.de
      - FABRIC_LOGGING_SPEC=info
      - CORE_CHAINCODE_LOGGING_LEVEL=info
      - CORE_PEER_LOCALMSPID=PinzgauerMSP
      - CORE_PEER_MSPCONFIGPATH=/etc/hyperledger/msp/peer/
      - CORE_PEER_ADDRESS=peer0.pinzgauer.de:7051
      - CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=orderingservice_basic
      - CORE_LEDGER_STATE_STATEDATABASE=CouchDB
      - CORE_LEDGER_STATE_COUCHDBCONFIG_COUCHDBADDRESS=couchdb.pinzgauer.de:5984
      - CORE_LEDGER_STATE_COUCHDBCONFIG_USERNAME=
      - CORE_LEDGER_STATE_COUCHDBCONFIG_PASSWORD=
      ### New stuff for PKCS#11
      - SOFTHSM2_CONF=/etc/hyperledger/fabric/HSM_config/softhsm2.conf
      - CORE_PEER_BCCSP_DEFAULT=PKCS11
      - CORE_PEER_BCCSP_PKCS11_HASH=SHA2
      - CORE_PEER_BCCSP_PKCS11_SECURITY=256
      - CORE_PEER_BCCSP_PKCS11_LIBRARY=/etc/hyperledger/fabric/HSM_lib/libsofthsm2.so
      - CORE_PEER_BCCSP_PKCS11_PIN=12345678
      - CORE_PEER_BCCSP_PKCS11_LABEL=ForPinzgauer
    working_dir: /opt/gopath/src/github.com/hyperledger/fabric
    command: peer node start
    # command: peer node start --peer-chaincodedev=true
    ports:
      - 7081:7051
      - 7083:7053
    volumes:
      - /var/run/:/host/var/run/
      - $PROJECT_DIR/creatingCryptoMaterial/crypto-
config/peerOrganizations/pinzgauer.de/peers/peer0.pinzgauer.de/msp:/etc/hyperledger/ms
p/peer
```

```
      ### mount the directory from the host where the PKCS#11 Library
      ### libsofthsm2.so can be found
      - /usr/lib/x86_64-linux-gnu/softhsm:/etc/hyperledger/fabric/HSM_lib
      ### The HSM config directory
      - /home/fabric/.config/softhsm2_docker:/etc/hyperledger/fabric/HSM_config
      ### The HSM token directory
      - /home/fabric/.config/softhsm2/tokens:/etc/hyperledger/fabric/HSM_tokens
    depends_on:
      - couchdb.pinzgauer.de
    networks:
      - OverlayNetwork
```

ACHTUNG: Der PIN für den Zugriff auf die Smartcard wird ohne “ “ angegeben!

Die HSM config und token Verzeichnisse müssen individuell angepaßt werden.

3.10Nutzung Smartcard(-HSM) vom Docker Container

3.10.1Orderer

3.10.1.1 Konfigurations-Variante "statisch"

Der Orderer wird mittels Docker-Compose über folgenden Aufruf gestartet:

```
docker-compose -f docker_compose_orderer_unibw.yaml up -d orderer.unibw.de
```

Der Konfigurationsfile docker_compose_orderer_unibw.yaml wird folgendermaßen erweitert:

```
Version: "3.5"

networks:
  OverlayNetwork:
    external: true
    driver: overlay
    attachable: true

services:
  orderer.unibw.de:
    container_name: orderer.unibw.de
    image: "hyperledger/fabric-orderer:${IMAGE_TAG}"
    environment:
      - FABRIC_LOGGING_SPEC=info
      - ORDERER_GENERAL_LISTENADDRESS=0.0.0.0
      - ORDERER_GENERAL_GENESISMETHOD=file
      - ORDERER_GENERAL_GENESISFILE=/etc/hyperledger/configtx/genesis.block
      - ORDERER_GENERAL_LOCALMSPID=UnibwMSP
      - ORDERER_GENERAL_LOCALMSPDIR=/etc/hyperledger/msp/orderer/msp
    working_dir: /opt/gopath/src/github.com/hyperledger/fabric/orderer
    command: orderer
    ports:
      - 7050:7050
    volumes:
      - $PROJECT_DIR/configTransactions/:/etc/hyperledger/configtx
      - $PROJECT_DIR/creatingCryptoMaterial/crypto-
config/ordererOrganizations/unibw.de/orderers/orderer.unibw.de/:/etc/hyperl
```

```
edger/msp/orderer
      ### mount the directory from the host where the PKCS#11 Library
      ### from OpenSC can be found
      - /usr/lib:/etc/hyperledger/fabric/HSM_lib
    networks:
      - OverlayNetwork
    devices:
      - "/dev/bus/usb/003/005"
```

1. Bei einem selbst gebauten OpenSC befindet sich die Library opensc-pkcs11.so im Verzeichnis /usr/lib!
2. Da der Docker Container auf einem Host läuft (Linux-PC oder Linux VM) muß zunächst das HW Device ermittelt werden, das vom Host zum Container "durchgereicht" werden soll. Dieses wird dann als device tag beim Start des Containers angegeben (siehe Einrichtung eines HSMs/Smartcard).