powered by e-commerce monitoring gmbh

Beispielcode für Signatur in C#, Java und C(C++)

Programmieranleitung für atos Smartcards CardOS v5.0, v5.3

v1.3d Stand: 10. Mai 2016

GLOBALTRUST QUALIFIED
e-commerce monitoring GmbH, 1160 Wien, Redtenbacherg. 20
Geschäftsadresse: A-1010 Wien, Vorlaufstraße 5/6
Gerichtsstand Wien
info@globaltrust.eu
http://globaltrust.eu
Akkreditierter Zertifizierungsdiensteanbieter
☎ +43/01/5320944, Fax +43/1/5320974
UID: ATU54992708 HG Wien FN 224536 a
text.76231taa

Inhalt

1 ZIELE DES DOKUMENTS

Dieses Dokument ist eine Anleitung für den mitgelieferten Beispielcode. Der Beispielcode soll die Übertragung der für die Signaturfunktion notwendigen APDUs veranschaulichen und ist nicht für einen live-Betrieb geeignet.

1.1 ÄNDERUNGSHISTORIE

1.1.1 v1.0 Stammfassung

Stand: 24. August 2015

1.1.2 v1.2 Ergänzung Signatur mit ECDSA Schlüsseln

Stand: 15. Oktober 2015

1.1.3 v1.3 Ergänzung für RKS-CARD Test- und Echtbetrieb

- Zusätzliche Hinweise zum Erkennen und Ansteuern des Cardreaders (⇨ 3.7 Cardreaderabfrage Seite 12)
- Klarstellungen zur PIN-Codierung (⇨ 3.8.2 Verify Seite 13)
- Verbesserte Beschreibung zum Auslesen der Zertifikate (⇨ 3.8.7 READBINARY Seite 16)
- Zusätzliche Hinweise und Anpassungen für CardOS v5.3
- diverse redaktionelle Korrekturen auf Grund der Supportanfragen

Stand: 11. Februar 2016

1.1.4 v1.3a Ergänzung für RKS-CARD Test- und Echtbetrieb

- Zusätzliche Hinweise zur JSON-Codierung des Signaturergebnisses (⇨ 3.2.2 Schlüsseltyp ECDSA (c#, java, C) Seite 9)
- Bekanntgabe der ZDA-Kennung AT2 für GLOBALTRUST
- diverse redaktionelle Korrekturen auf Grund der Supportanfragen

Stand: 10. März 2016

1.1.5 v1.3b Ergänzung für RKS-CARD Test- und Echtbetrieb

- Hinweis zu VB.NET eingebaut
- diverse redaktionelle Korrekturen auf Grund der Supportanfragen

Stand: 15. März 2016

1.1.6 v1.3c Ergänzung für RKS-CARD Test- und Echtbetrieb

- Erweiterung der Kommentare zum C(C++)-Code, Bereinigung Ablauf
- diverse redaktionelle Korrekturen auf Grund der Supportanfragen

Stand: 27. April 2016

1.1.7 v1.3d Ergänzung für RKS-CARD Test- und Echtbetrieb

- Dokumentation Code C#, Java für Verify-Signature bei "Signature RAW" (p9): Erzeugen von zwei verschiedenen Signaturergebnissen "signature" und "dersignature"
- diverse redaktionelle Korrekturen auf Grund der Supportanfragen

Stand: 10. Mai 2016

2 Grundlagen

Die Parameter der beschriebenen APDUs beziehen sich auf Smartcards mit Betriebssystem atos CardOS v5.0 [kurz: **V50**] bzw. CardOS v5.3 [kurz: **V53**] mit einem GLOBALTRUST Zertifikat.

Hinweis!
Für die RKS-CARD ist nur der Teil für CardOS v5.3 [V53] wesentlich!

Die APDU-Befehle werden im ISO Standard ISO7816-4 zu entnehmen:
http://www.cardwerk.com/smartcards/smartcard_standard_ISO7816-4_6_basic_interindustry_commands.aspx

Weiters ist das standardisierte Framework für PC / SC (Smartcard) Interaktion erforderlich.

Unterstützte Schlüsseltypen:
- RSA Schlüssel vom Typ RSASSA-PKCS#1-V1_5, Länge 2048 [kurz: RSAKEY]
- ECDSA Schlüssel vom Typ ECDSA P-256 Curve [kurz: ECDSAKEY]

In beiden Fällen wird als Hash-Algorithmus SHA-256 verwendet.

2.1 Hinweis zum gelieferten Beispielcode

Unterstützte Programmiersprachen:
- Java ab JDK v1.7
- C# ab Framework .NET 4.5 oder Mono 3.2.8 oder höher
- C oder C++ mit Kompiler

Der Beispielcode erfüllt folgende Funktionen:
- Erzeugen aus einem String den für die Signatur erforderlichen Hash-Wert
- Verifizieren des PIN[1] der Smartcard
- Erstellen der Signatur
- Prüfen der Signatur

Die im Beispielcode ausgeführte Validierung der Signatur dient nur zum Test einer Signatur und ist für die Signaturerstellung nicht zwingend notwendig. Das Public-Key Zertifikat für die Signaturüberprüfung wird mit derSmartcard mitgeliefert.

---

[1] PIN wird in diesem Dokument als allgemeine Bezeichnung für alle drei Arten der PINs verwendet: TransportPIN, SignaturPIN und PUK. Beachten Sie das in zahlreichen Dokumentationen und fallweise auch in den Kommentaren im Demo-Code PIN auch stellvertretend für SignaturPIN verwendet werden kann.

Bei Produktivkarten ist das Zertifikat auch über den LDAP-Server abrufbar, sofern der Signator einer Veröffentlichung zugestimmt hat (siehe ⇨ 2.3 LDAP-Zugriff auf Zertifikatsinformationen).

Beispiele für die Verwendung von PC/SC siehe auch:
http://ludovicrousseau.blogspot.co.at/2010/04/pcsc-sample-in-different-languages.html

2.2 Hinweis zu VB (Visual Basic)

Zu VB Classic (bis VB 6) kann keine Unterstützung angeboten werden, wir empfehlen Entwicklern auf VB .NET umzusteigen und das .NET- Framework ab 4.5 zu verwenden. Der Code ist weitgehend ident mit dem Beispielcode für c#. Ein eigener Beispielcode ist nicht vorgesehen.

2.3 LDAP-Zugriff auf Zertifikatsinformationen

Host: ldap://ldap.globaltrust.eu:389
Es ist keine Authentifizierung notwendig.
Base DN: C=AT
Pfad zu Zertifikaten:
serialNumber=<dezimale Seriennummer>,OU=<CANAME>[2]*,O=GLOBALTRUST,C=AT*

Zertifikate einer Test-SubCA (zB GLOBALTRUST RKS-CARD 2015 TEST 1) werden nicht am LDAP-Server veröffentlicht!

2.4 Kennung des Zertifizierungsdiensteanbieters gemäß RKSV

Die Anlage der RKSV verlangt unter "2. Registrierkassenalgorithmuskennzeichen" die Bekanntgabe eines Kennzeichens des ZDA (Auszug aus der Verordnung siehe unten).

Die Kennung für die RKS-CARD lautet AT2, der entsprechende String daher: R1-AT2

---

2 <CANAME> steht für die jeweilige SubCA
GLOBALTRUST RKS-CARD 2015 SubCA 1 == Smartcards gemäß RKSV
GLOBALTRUST RKS-CARD 2015 TEST 1 == Test-Smartcards gemäß RKSV
GLOBALTRUST QUALIFIED 1 == Smartcards mit qualifiziertem Zertifikat
GLOBALTRUST QUALIFIED TEST ==Smartcards mit einfachem Zertifikat (Test-Smartcards)
Bei Unklarheiten welche SubCA für die ausgestellte Smartcard zutrifft, wenden Sie sich an den GLOBALTRUST-Support.
Der Mustercode enthält auch Informationen zum Auslesen der SubCA

Auszug BGBl. II Nr. 410/2015 - Registrierkassensicherheitsverordnung, RKSV
2. Registrierkassenalgorithmuskennzeichen
Dieses Kennzeichen definiert die verwendeten Algorithmen und den Zertifizierungsdiensteanbieter (ZDA). Sobald ein in den Registrierkassenalgorithmuskennzeichen verwendeter Algorithmus nicht mehr im Anhang der SigV 2008 genannt wird und daher als unsicher gilt, muss ein neues Registrierkassenalgorithmuskennzeichen mit sicheren Algorithmen definiert werden und darf dieses auch bei bestehenden Registrierkassen nicht mehr eingesetzt werden. Das Kennzeichen entspricht einer Zeichenkette, die wie folgt aufgebaut ist:
RN-CM:
- *„R": Fixes Präfix*
- *„N": Index für die verwendete Algorithmen-Suite startend mit 1*
- *„-„: Fixes Trennzeichen*
- *„C": Länderkennung des ZDAs*
- *„M": Index für verwendeten ZDA innerhalb der gegeben Länderkennung nach ISO 3166-1 startend mit 1*

Die folgenden Kennzeichen sind definiert:
R1-CM:
- *ZDA: CM wird als Platzhalter für die zur Verfügung stehenden ZDAs gesehen. Wenn ein geschlossenes System laut § 20 zum Einsatz kommt, muss AT0 als ZDA angebenden werden.*

3 DOKUMENTATION

3.1 SOFT- UND HARDWAREVORAUSSETZUNGEN

Die folgenden Softwarepakete müssen zur Ausführung des Beispielcodes installiert sein

3.1.1 CARDREADER + TREIBER

Grundsätzlich kann jeder ISO 7816-konformer Kartenleser verwendet werden.

Empfohlenes Standardgerät:
- Gemalto IDBridge CT40 ohne PIN-Eingabe
- Cherry SmartTerminal ST-2xxx USB mit PIN-Eingabe
- weitere geeignete Produkte sind auf der GLOBALTRUST-Website http://www.globaltrust.eu/produkte.html gelistet

Hardwaretreiber für das Kartenlesegerät:
- **Windows:** Treiber des Hardwareherstellers installieren
- **Linux:** *libccid* installieren (Universaltreiber für USB-Kartenleser)
  zB unter OpenSUSE: (sudo) zypper install pcsc-ccid

3.1.2 PC/SC-INSTALLATION

Windows:
PC/SC ist in Windows-OS ab 7.0 integriert.

Linux:
- pcsc-lite MUSS installiert werden (siehe https://pcsclite.alioth.debian.org/)
- einfacher Test ob Pakete korrekt installiert sind: '$> pcsc_scan'
  (liefert Liste der angeschlossenen Cardreader)
- Beispiel erforderliche Pakete für OpenSuse: pcsclite1, pcsclite-dev, pcscd, pcsc-tools

Ergänzender Implementierungshinweis für Java!
Auf 64bit Linux Systemen verwendet die Oracle JDK einen falschen Pfad zur pcsclite library. Dieses Problem wird mittels eines symlinks behoben.
sudo mkdir /usr/lib64
sudo ln -s /lib/x86_64-linux-gnu/libpcsclite.so.1 /usr/lib64/libpcsclite.so

3.1.3 SPRACHSPEZIFISCHE PC/SC ABSTRAKTION

- **Java:** javax.smartcardio (inkludiert in der JRE ab Version 1.6)
- **C#:** pcsc-sharp referenzieren (https://github.com/danm-de/pcsc-sharp)
- **C (C++):** pcsc lite (siehe https://alioth.debian.org/)

Im Abschnitt ⇨ 4.4 Übersicht interne Test des Mustercodes (Seite 41) sind einige von GLOBALTRUST getestete Installationen gelistet.

3.1.4 ZUSÄTZLICH ERFORDERLICHE PAKETE

- **Java, C#:** verwendet die Bibliothek *BouncyCastle* (https://www.bouncycastle.org/)[3]
- **C (C++):** verwendet die EVP-Bibliotheken von openssl (https://www.openssl.org/)

Hinweis! Abhängig von der bestehenden Entwicklerumgebung kann es notwendig sein zusätzliche Pakete zu installieren, z.B. zur Kommunikation mit dem Kartenleser.

3.2 SIGNATURFORMAT

Hinweis zu Mustercode java, c# !
Standardmäßig wird beim Mustercode RSA-Verschlüsselung verwendet.
Zur Verwendung von ECDSA muss das Musterprogramm mit dem Parameter args[0] = "ECDSA" aufgerufen werden

⇨ c# Codebeispiel: Seite 18
⇨ java Codebeispiel: Seite 25
⇨ C(C++) Codebeispiel: nicht zutreffend, da nur ECDSA Mustercode bereitgestellt

[3] Die Bibliothek *BouncyCastle* wird zu Hilfszwecken (z.B für die DER Kodierung von Datentypen, Signaturprüfung) eingesetzt und ist für die Verwendung der Smartcard nicht zwingend notwendig.

Das Rückgabeformat des Signaturwertes von der Smartcard ist abhängig vom verwendeten Schlüsseltyp.

3.2.1 SCHLÜSSELTYP RSA (C# UND JAVA ONLY)

Wird auf der Smartcard ein RSA Schlüssel verwendet, gibt die Smartcard einen gemäß PKCS#1 formatierten Signaturwert zurück, der ohne Anpassung weiter verwendet werden kann.

3.2.2 SCHLÜSSELTYP ECDSA (C#, JAVA, C)

Wird auf der Smartcard ein ECDSA Schlüssel verwendet (zB bei der **RKS-CARD**), gibt die Smartcard den Signaturwert als unformatierte Aneinanderreihung der beiden Signaturteile zurück ("Signature RAW", siehe Codeabschnitt in C# bzw. Java "// Create the signature unformatted").

Um die Signatur mit Standardsoftware weiter verwenden zu können, muss die Signatur formatiert werden. Dazu wird der Datentyp *ECDSA-Sig-Value* gemäß RFC5480 erzeugt (siehe Codeabschnitt in C# bzw. Java "// Encode ECDSA signature using DER".

Hinweis RKS-CARD!

Im Fall der Verwendung der Signatur als **RKS-CARD** MUSS die unformatierte Signaturversion verwendet, verkettet und Base64-codiert werden. Der Abschnitt "// Encode ECDSA signature using DER" entfällt.

Beispiel zur JSON-Kodierung Using ECDSA P-256 SHA-256: siehe JSON Web Signature (JWS) RFC7515 https://tools.ietf.org/html/rfc7515#appendix-A.3.1)

c#, java: Dabei werden die beiden Teile der Signatur als unsigned integer Werte in eine Sequenz verpackt.
C (C++): Split signature in two and write it as BIGNUM to ECDSA_SIG struct. Verify with the openssl ECDSA library, func used - ECDSA_do_verify.

⇨ c# Codebeispiel: Seite 22
⇨ java Codebeispiel: Seite 29
⇨ C(C++) Codebeispiel: Seite 39

Hinweis! Um eine unformatierte Signatur prüfen zu können (3.8.2 Verify, p13) MUSS sie vorher DER-codiert werden ("signature" liefert die unformatierte Signatur, "dersignature" liefert die DER-codierte Signatur).

3.3 SIGNATURPIN UND PUK

Hinweis!
SignaturPIN und PUK sind in der Zahl der Fehlversuche streng limitiert. Bei qualifizierten Zertifikaten ist die erlaubte Maximalzahl jeweils 3, ansonsten bei SignaturPIN 15.

GLOBALTRUST stellt für [V50] eine eigene Windows-Applikation zur Verfügung, die es erlaubt auch ohne technische Vorkenntnisse einen gesperrten SignaturPIN durch den PUK wieder zu aktivieren. Diese Applikation + Dokumentation können Sie downloaden.

Für [V53] sind die Befehle Verify PUK und Reset Counter SignaturPIN zu verwenden.

Downloadlinks:
- Software: http://www.globaltrust.eu/static/globaltrustapp.exe
- Doku: http://www.globaltrust.eu/static/doku-globaltrustapp.pdf

Einschränkungen in der Verwendung der GlobalTrustApp
Die Signaturfunktionen für pdf und XML sind derzeit nur für RSA-Schlüssel freigeschalten, nicht für ECDSA-Schlüssel, wie auf der RKS-CARD vorgesehen.

3.4 TRANSPORTPIN

Hinweis!
Der TransportPIN ist ausschließlich bei qualifizierten Zertifikaten erforderlich. Wird die Smartcard ohne TransportPIN ausgeliefert (z.B. Testkarten, RKS-CARD, ...), kann dieser Abschnitt ignoriert werden!

Neu ausgestellte Signaturkarten sind mit einer TransportPIN gesichert. Um die Smartcard freizuschalten müssen SignaturPIN und PUK der Smartcard erst unter Verwendung der TransportPIN gesetzt werden.
Zum Setzen von SignaturPIN und PUK werden folgende APDUs benötigt:
- Verify: Überprüfung der TransportPIN zur Autorisierung der PIN/PUK Änderung.
- ChangeReferenceData: setzen der SignaturPIN
- ChangeReferenceDate: setzen des PUK

ACHTUNG!
Der TransportPIN kann genau einmal validiert werden und autorisiert genau 2 weitere APDUs. Die oben angeführten APDUs sind daher direkt hintereinander auszuführen oder die Smartcard wird unbrauchbar.

3.5 ERSTELLEN HASH-WERT FÜR SIGNATUR

Damit eine Zeichenkette signiert werden kann, ist daraus ein HASH-Wert (in der Regel SHA-256) zu erzeugen.

Die Zeichenkette kann zum Beispiel eine Datei, ein Buchungsjournal oder ein einzelner Datensatz, etwa eines Barumsatzes, sein.

3.5.1 ERSTELLEN HASH-WERT FÜR C# IN DER KLASSE SIGNER

⇨ c# Codebeispiel: Seite 21

```
SHA256Managed hashAlgorithm = new SHA256Managed ();
byte[] hash = hashAlgorithm.ComputeHash (data);
```

Wobei "SHA256Managed" eine Standard-.NET-Klasse ist.

3.5.2 ERSTELLEN HASH-WERT FÜR JAVA IN DER KLASSE SIGNER

⇨ java Codebeispiel: Seite 28

```
MessageDigest digest = MessageDigest.getInstance("SHA-256");
byte[] hash = digest.digest(data);
```

Wobei "MessageDigest" eine Standard-Java-Klasse ist.

3.5.3 *Erstellen Hash-Wert für C(C++) in der Klasse sample*

⇨ C(C++) Codebeispiel: Seite 33

```
EVP_MD_CTX *mdctx;
const EVP_MD *md;
unsigned char md_value[256];
unsigned int md_len, i;
md = EVP_sha256();
mdctx = EVP_MD_CTX_create();
EVP_DigestInit_ex(mdctx, md, NULL);
EVP_DigestUpdate(mdctx, data, strlen(data));
EVP_DigestFinal_ex(mdctx, md_value, &md_len);
EVP_MD_CTX_destroy(mdctx);
```

3.6 Smartcardinformation im Überblick

Diese Informationen dienen vorrangig für versierte Programmierer, die die Smartcard zur Nutzung für qualifizierte Signaturen verwenden wollen.

Für die Programmierung bei einfachen Signaturen (etwa der RKS-CARD) empfehlen wir sofort zu Abschnitt ⇨ 3.8 APDU (Application Protocol Data Unit) weiter zu gehen.

Nutzung APDUs Kartenkommandos zum Signieren (gemäß ISO 7816-4):

- Verify (PIN-Eingabe)
- Manage Security Environment (MSE), nur wenn spezielle Signaturen mit der Smartcard gemacht werden sollen. Mit der Standard Security Environment wird mit externem Hash und PKCS#1 BT1 Padding (RSA) bzw. ohne Padding (ECDSA) signiert)
- Perform Security Operation Compute Digital Signature (PSO_CDS, Signatur berechnen)

erforderliche IDs der Atos-Applikation QES (RKS-CARD oder QUALIFIED, nur bei [V50])
Verwaltet Schlüssel, PINs und sonstige Card-Secrets
- Application-Identifier[4] (AID) #1[5]: 0x1F 0xFF
-- ID des Schlüsselpaars: 0x02
-- ID des SignaturPIN[6]: 0x01 [V50] + [V53]
-- ID des PUK: 0x0A [V50] / 0x02 [V53]
-- ID der TransportPIN: 0x0B [V50] / 0x1E [V53]

FIDs der DF[7] bzw EF[8] der Globaltrust-Applikation CERT (RKS-CARD oder QUALIFIED, nur bei [V50])
Verwaltet Zertifikate und erlaubt Auslesen der Zertifikaten
- Application-Identifier (AID): 0x11 0x60 (*Verzeichnis*)
-- FIDs für Root-Zertifikat, Zwischenzertifikate und zusätzliche Zertifikate hängt davon ab wieviele Zwischenzertifikate der CA vorhanden sind. Es sind 4 EFs für diese Zertifikate reserviert: 0x63 0x01, 0x63 0x02, 0x63 0x03, 0x63 0x04
-- FID für End-Zertifikat: 0xC0 0x01 (*Zertifikat*)

Zum Auslesen wird ein APDU READ BINARY command benötigt.

---

4 Fallweise auch als allgemein als ObjectIdentifier bezeichnet.

5 Die GLOBALTRUST-Karte kann - je nach Anwendung - weitere Application-Identifier enthalten. Daten dazu auf Anfrage. Für die RKS-CARD wird nur dieser dokumentierte Application-Identifier verwendet!

6 Hinweis zum Backtracking: MSB von den IDs ist auf 1 zu setzen!

7 DF = Dedicated Files

8 EF = Elementary Files

FIDs der Dateien im MF
-- FID des GDO (nur bei [V50] enthält Metadaten, z.B. Name des Karteninhabers): 0x2F 0x02

3.7 CARDREADERABFRAGE

Der Mustercode für C# und java erlaubt den Anschluss mehrere Cardreader und/oder die geräteabhängige Auswahl eines Cardreaders. Damit könnte zB auch die Ansteuerung mehrere Smartcards über ein Registrierkassen-Programm realisiert werden (zB die Verwaltung mehrere Unternehmenskennungen).

⇨ c# Codebeispiel: Seite 20
⇨ java Codebeispiel: Seite 27
⇨ C(C++) Codebeispiel: nicht zutreffend

3.8 APDU (APPLICATION PROTOCOL DATA UNIT)

Command APDUs sind Kommandos als Bytefolge, die an die Smartcard geschickt werden. Der generelle Aufbau von CommandAPDUs ist:
{CLA (1 Byte), **INS**(1 Byte), **P1**(1 Byte), **P2**(1 Byte), **LC**(0-3 Bytes), **Data**(n Bytes), **LE**(0-3 Bytes)**}**
⇨ ISO7816: http://www.cardwerk.com/smartcards/smartcard_standard_ISO7816-4_5_basic_organizations.aspx#chap5_3

Die Antwort der Smartcard (Response APDU) besteht aus einer Bytefolge:
{Data (n Bytes), **Statuscode**(2 Bytes)**}**
Im Allgemeinen bedeutet der Statuscode (0x9000) ein erfolgreich ausgeführtes Kommando und andere Statuscodes einen Fehler.

Im Folgenden werden die für die Signaturfunktion verwendeten APDUs erläutert.

3.8.1 SELECT

Selektiert eine Applikation auf der Smartcard. Im gegebenen Fall ist die Applikation die Signaturfunktion der Karte. Alle folgenden Kommandos (PIN Überprüfung, etc...) werden an die selektierte Applikation geschickt.

Feld	Bytes	Wert (hex)	Kommentar
CLA, INS	2	00A4	Applikationsselektion
P1, P2	2	0804	
LC	1	02	
Data	2	1FFF	Beispiel eines ObjectIdentifier (in diesem Fall des Application-Identifier #1)
LE	0		

Tabelle 1: Select

3.8.2 VERIFY

PIN Abfrage zur Autorisierung der Signaturfunktion.

Beim Setzen von PINs ist darauf zu achten, dass kurze PINs korrekt gepaddet werden.

Die minimale Länge für SignaturPINs (excl. Padding) ist 6 Bytes. Die minimale Länge für PUKs (excl. Padding) ist 8 Bytes.

Hinweis PIN-Eingabe!
Abhängig von der Kartenversion [V50] oder [V53] ist die PIN-Eingabe unterschiedlich!
⇨ c# Codebeispiel: Seite 19
⇨ java Codebeispiel: Seite 27
⇨ C(C++) Codebeispiel: Seite 32

[V50] UTF-8 encoded and padded with 0xff bytes to the maximal length of 12
Annahme: SignaturPIN=123456 ⇨ "313233343536" = ASCII-Hex-Value of "123456"
Bei 6-stelligem SignaturPIN sind weitere sechs Stellen "ff" hinzuzufügen!

[V53] BCD encoded and padded with 0xff bytes to 8 byte
Annahme: SignaturPIN=123456 ⇨ "26123456" = "2": fixed, "6": length of PIN, 123456: BCD-coded PIN
Bei 6-stelligem SignaturPIN sind weitere vier Stellen "ff" hinzuzufügen!

Feld	Bytes	Wert (hex)	Kommentar
CLA, INS	2	0020	PIN Überprüfung
P1, P2	2	00**	P1 = 00 P2 = PinId mit **: SignaturPIN: 81 [V50]+[V53] PUK: 8A [V50], 82 [V53] TransportPIN: 8B [V50], 9E [V53]
LC	1	0C[9] / 08[10]	Gesamtlänge des PIN in hex
Data	12 / 8		HexWert des PIN (+0xFF...)
LE	0		

Tabelle 2: Verify

Bei fehlerhafter Eingabe des PINs wird ein Zähler für den PIN dekrementiert. Der Statuscode bei falscher PIN Überprüfung ist *63Cx(hex)* wobei '**x**' die Anzahl der verbleibenden Eingabeversuche ist.

Erreicht der Zähler den Wert '0' wird der PIN gesperrt und eine weitere Verwendung der Signaturfunktion ist daher nicht möglich.

9 für [V50]
10 für [V53]

Der Wert des Zählers (**x** von *63C**x**(hex)*) kann durch die selbe APDU abgefragt werden, wenn das *Data* Feld leer bleibt (APDU: 0x00 20 00 81 entspricht APDU_PIN_COUNT im Mustercode). APDU_PIN_COUNT liefert kein 'OK' (0x9000) retour.

[V50] Beispielstring für PIN-Verify, wenn SignaturPIN 123456:
`002000810c313233343536ffffffffffff`

[V53] Beispielstring für PIN-Verify, wenn SignaturPIN 123456:
`0020008108 26123456ffffffff`

3.8.3 CHANGE REFERENCE DATA (CSR)

Ersetzt den Wert eines PIN Objekts und setzt den PIN Zähler auf seinen maximalen Wert.

Feld	Bytes	Wert (hex)	Kommentar
CLA, INS	2	0024	PIN Änderung
P1, P2	2	01**	P1 = 01: explizite Authentifizierung P2 = PinId mit **: SignaturPIN: 81 [V50]+[V53] PUK: 8A [V50], 82 [V53] TransportPIN: 8B [V50], 9E [V53]
LC	1	0C[9] / 08[10]	
Data	12 / 8		HexWert des PIN (+0xFF...)
LE	0		

Tabelle 3: Change Reference Data

3.8.4 RESET RETRY COUNTER (RRC)

Ersetzt den Wert eines PIN Objekts und setzt den PIN Zähler auf seinen maximalen Wert.

Feld	Bytes	Wert (hex)	Kommentar
CLA, INS	2	002c	Reset eines Retry Counters
P1, P2	2	03**	P1 = 03: Vorherige Verifizierung des PUK erforderlich! P2 = PinId mit **: SignaturPIN: 81 [V50]+[V53]
Data	0		
LE	0		

Tabelle 4: Reset Retry Counter

3.8.5 MANAGE SECURITY ENVIRONMENT (MSE)

Auswahl der Parameter und kryptografischen Algorithmen der Signaturfunktion.

Dieses Kommando ist NUR bei Verwendung eines RSA-Schlüssel erforderlich:
[V50]: RSA mit PKCS#1_BT1 Padding

Feld	Bytes	Wert (hex)	Kommentar
CLA, INS	2	0022	Manage Security Environment
P1, P2	2	F304	Laden eines vorinstallierten Environments P1 = F3: Restore (Laden eines vorinstallierten Environments) P2 = 04: Security Environment ID
LC	0		
Data	0		
LE	0		

Tabelle 5: MSE

Signaturen mit ECDSA verwenden das Default Einvironment!

3.8.6 PERFORM SECURITY OPERATION, COMPUTE DIGITAL SIGNATURE (PSO_CDS)

Der Hash (SHA-256) wird off-card erzeugt, und wie in RFC 3447 (⇨ https://tools.ietf.org/html/rfc3447#page-32) beschrieben in ein DigestInfo Objekt verpackt.

Feld	Bytes	Wert (hex)	Kommentar
CLA, INS	2	002A	Signaturerstellung
P1, P2	2	9E9A	Hashwert wird off-card erzeugt
LC	3	Data.Length	0x00 + Länge der DigestInfo in 2 Bytes
Data			DigestInfo (AlgorithmIdentifier + Hash) (RSA) bzw. Hashwert (ECDSA).
LE	2		

Tabelle 6: PSO_CDS

Nicht alle Cardreader unterstützen Extended-Length APDUs. Sollte bei diesem Kommando eine Exception ('Unexpected end of Transaction' oder ähnliches) auftreten ist dies eine mögliche Ursache.

3.8.7 READBINARY

Liest Daten aus einem Binärfile aus

Feld	Bytes	Wert (hex)	Kommentar
CLA, INS	2	00b0	read binary
P1, P2	2	0000	offset
LC			
Data			
LE		0100	blocklength = 256 Byte

Tabelle 7: READBINARY

Auslesen User/Signer-Zertifikat + Seriennummer:

⇨ c# Codebeispiel: Seite 18
⇨ java Codebeispiel: Seite 25
⇨ C(C++) Codebeispiel: Seite 37

```
// Muster Read the signer's certificate from the smartcard
X509Certificate userCertificate =
signer.ReadCertificateFromCard(Signer.APDU_SELECT_USER_CERTEF);
Console.WriteLine ("userCertificate :" +  userCertificate.ToString());
```

Auslesen SubCA-Zertifikat + Seriennummer:

⇨ c# Codebeispiel: Seite 19
⇨ java Codebeispiel: Seite 26
⇨ C(C++) Codebeispiel: --

4 MUSTERCODE

Den Mustercode können Sie auch in elektronischer Form erhalten bzw. können ihn bei GLOBALTRUST-Support anfordern. Beachten Sie, dass an verschiedenen Stellen (meist gelb markiert) Anpassungen für ihre ausgelieferte Smartcard und/oder Cardreader erforderlich sein können (ua Verschlüsselungsart, PIN-Verwendung, Signaturart, Prüfmethode der Signatur, Cardreader, ...). Der vorliegende Mustercode ist ohne geeignete Anpassungen an Ihre Anforderungen NICHT lauffähig! **Der dokumentierte Code in diesem Dokument kann vom gelieferten Mustercode abweichen!**

Hinweis!
Beispielcode für C# und Java enthält Code-Muster für [V50] und [V53], [RSAKEY] und [ECDSAKEY]
Beispielcode für C (C++) enthält Code-Muster für [V50] und [V53], [ECDSAKEY] only

4.1 C#

4.1.1 PROGRAM.CS

```
using System;
using System.Security.Cryptography;
using RegisterIntegration.Extensions;
using Org.BouncyCastle.Crypto.Parameters;
using Org.BouncyCastle.Crypto;
using Org.BouncyCastle.Security;
using System.Text;
using Org.BouncyCastle.X509;
using System.IO;

namespace Eu.GlobalTrust.SignatureSample
{
    public class Program
    {
        public static void Main(string[] args)
        {
            // Signature input
            string s = "crinoline";
            byte[] data = Encoding.UTF8.GetBytes (s);
```

```
// Determine key type (RSA vs. ECDSA)
// RSA by default, ECDSA if "ECDSA" is given as first argument to the program.
            Signer.KeyType keyType = Signer.KeyType.Rsa;
            if(args.Length > 0)
                if(args[0].ToUpper() == "ECDSA")
                keyType = Signer.KeyType.Ecdsa;

            // Sign data with Smartcard
            Signer signer = new Signer ();
            byte[] signature = signer.Sign (data, keyType);

// in case of RKS-CARD use Base64-Encoding for using in JSON-Frmat! uncomment this command!
//            String signatureBase64 = Convert.ToBase64String(signature);

            if(signature == null) {
              Console.WriteLine("Could not create signature.");
              return;
            }

// Read the signer's certificate from the smartcard
            X509Certificate userCertificate = signer.ReadCertificateFromCard(Signer.APDU_SELECT_USER_CERTEF);

//Load the signer's public key from the certificate
            var key = userCertificate.GetPublicKey();
            Console.WriteLine(key);

// Verify signature using the public key
            ISigner bcSigner = null;
            if(keyType == Signer.KeyType.Rsa)
               bcSigner = SignerUtilities.GetSigner("SHA256withRSA");
            else if(keyType == Signer.KeyType.Ecdsa) {
               bcSigner = SignerUtilities.GetSigner("SHA-256withECDSA");
// Extra-DER-Coding only for Verify Signature
               signature = Signer.DerEncodeSignature(signature);
            }
            bcSigner.Init(false, key);
            bcSigner.BlockUpdate(data, 0, data.Length);
            bool isValidSignature = bcSigner.VerifySignature (signature);
            Console.WriteLine("signatureVerified: " + isValidSignature);
            Console.WriteLine("user certificate serial#: " + userCertificate.SerialNumber);
```

```
// Read the SubCA certificate from the smartcard
            X509Certificate caCertificate = signer.ReadCertificateFromCard(Signer.APDU_SELECT_CA_CERTEF);
            Console.WriteLine("ca certificate serial#: " + caCertificate.SerialNumber);
        }
    }
}
```

4.1.2 SIGNER.CS

```
using System;
using System.Collections.Generic;
using System.Linq;
using PCSC;
using System.Text;
using System.Security.Cryptography;
using RegisterIntegration.Extensions;
using Org.BouncyCastle.Asn1;
using Org.BouncyCastle.Asn1.X509;
using Org.BouncyCastle.Asn1.Nist;
using Org.BouncyCastle.Asn1.BC;
using Org.BouncyCastle.X509;

namespace Eu.GlobalTrust.SignatureSample
{
    public class Signer
    {

    public enum KeyType
    {
        Rsa,
        Ecdsa,
    }

        // The PIN for CardOS v5.3 cards is BCD encoded and padded with 0xff bytes to 8 byte
        // "26123456" = 2: fixed, 6: length of PIN, 123456: BCD-coded PIN (Example only!! Replace with deliverd PIN!!)
        public static readonly string PIN_BCD = "26123456" + "ffffffff";

        // The PIN for CardOS v5.0 cards is UTF-8 encoded and padded with 0xff bytes to the maximal length of 12
        // "313233343536" = ASCII-Hex-Value of "123456" (Example only!! Replace with deliverd PIN!!)
```

```
        public static readonly string PIN_PADDED = "313233343536" + "ffffffffffff";

        public static readonly byte[] APDU_GETDATA_CARDOSVERSION = "00ca018202".ToByteArray();
        public static readonly byte[] APDU_SERIAL_NO = "00ca018108".ToByteArray();
        public static readonly byte[] APDU_SELECT_QESDF =  "00a40804021fff".ToByteArray();
        public static readonly byte[] APDU_SELECT_USER_CERTEF =  "00a40804041160c001".ToByteArray();
        public static readonly byte[] APDU_SELECT_CA_CERTEF =  "00a408040411606301".ToByteArray();
        public static readonly byte[] APDU_PIN_COUNT = "00200081".ToByteArray();
        public static readonly byte[] APDU_PIN_VERIFY_50 = ("002000810c" + PIN_PADDED).ToByteArray();
        public static readonly byte[] APDU_PIN_VERIFY_53 = ("0020008108" + PIN_BCD).ToByteArray();
        public static readonly byte[] APDU_MSE = "0022F304".ToByteArray();
        public static readonly byte[] APDU_READBINARY = "00B000000000100".ToByteArray();
        public static readonly byte[] APDUHEADER_PSO_CDS = "002a9e9a".ToByteArray();

        private string cardOsVersion;

        private SCardReader Reader { get; set; }
        private IntPtr Pci { get; set; }

        public Signer()
        {
            SCardContext context = new SCardContext();
            context.Establish(SCardScope.System);

            // Enumerate readers
            string[] readers = context.GetReaders();
            for (int i = 0; i < readers.Length; i++) {
                var foo = context.GetReaderStatus (readers [i]).CurrentState;

// Reader-Example ONLY!!
                if (readers[i].ToUpper().Contains("CHERRY") || readers[i].ToUpper().Contains("OMNIKEY")
                    || readers[i].ToUpper().Contains("GEMALTO")) {
                    Console.WriteLine("reader name: " + readers[i]);
                    // Create a reader object using the existing context
                    Reader = new SCardReader(context);
                    // Connect to the card and select protocol
                    Reader.Connect(readers[i], SCardShareMode.Shared, SCardProtocol.Any);
                    break;
                }
            }
            Pci = SCardPCI.GetPci(Reader.ActiveProtocol);
```

```
// Find out the CardOS version of the smartcard (5.0 and 5.3 are supported)
            byte[] cardOsVersionEncoded = TransmitApdu(APDU_GETDATA_CARDOSVERSION, null);
            if(cardOsVersionEncoded[0] == 0xc9 && cardOsVersionEncoded[1] == 0x03)
                cardOsVersion = "5.3";
            else if(cardOsVersionEncoded[0] == 0xc9 && cardOsVersionEncoded[1] == 0x01)
                cardOsVersion = "5.0";
        }

// Creates actual PSO_CDS APDU based on header and signature input.
        public static byte[] PsoCdsApdu(byte[] sigData) {
            byte[] apdu = new byte[4 + 3 + sigData.Length + 2];
            Array.Copy(APDUHEADER_PSO_CDS, 0, apdu, 0, APDUHEADER_PSO_CDS.Length);
            Array.Copy(new byte[] {0x00, 0x00, (byte)sigData.Length}, 0, apdu, 4, 3);
            Array.Copy(sigData, 0, apdu, 7, sigData.Length);
            Array.Copy(new byte[] { 0x01, 0x02 }, 0, apdu, 7 + sigData.Length, 2);
            return apdu;
        }

// Hashes and signs the input data using a Smartcard.
        public byte[] Sign(byte[] data, KeyType keyType = KeyType.Rsa)
        {
// Hash input data using SHA-256
            SHA256Managed hashAlgorithm = new SHA256Managed ();
            byte[] hash = hashAlgorithm.ComputeHash (data);
            byte[] signatureData = null;

// Prepare hash as input for signature operation
            if(keyType == KeyType.Rsa) {
                DigestInfo digestInfo = new DigestInfo (new AlgorithmIdentifier (NistObjectIdentifiers.IdSha256), hash);
                signatureData = digestInfo.GetDerEncoded();
            }else if (keyType == KeyType.Ecdsa) {
                signatureData = hash;
            }

// Create signature APDU
            byte[] psoCdsApdu = PsoCdsApdu(signatureData);
            Console.WriteLine ("psoCdsApdu: " + psoCdsApdu.ToHexString ());

// Transmit APDUs to card
            TransmitApdu (APDU_SERIAL_NO, "serialNumber");
```

```
//Select the digital signature application
            TransmitApdu (APDU_SELECT_QESDF, "select");

//Verify the PIN
            TransmitApdu (APDU_PIN_COUNT, "pinCount");
            if(cardOsVersion == "5.0")
              TransmitApdu (APDU_PIN_VERIFY_50, "verify");
            else if(cardOsVersion == "5.3")
              TransmitApdu (APDU_PIN_VERIFY_53, "verify");

// Restore MSE only needed for RSA cards (not for RKS-CARD, other ECDSA-Cards)
            if (keyType == KeyType.Rsa)
                TransmitApdu (APDU_MSE, "mse");

// Create the signature unformatted
            byte[] signature = TransmitApdu (psoCdsApdu, "signature");
            if (signature.Length == 0)
// Error while creating the signature
                return null;

// Info only
            Console.WriteLine ("well done");
            return signature;
      }

// Encode ECDSA signature using DER (not for RKS-CARD!)
         public static byte[] DerEncodeSignature (byte[] signature)
         {
              Org.BouncyCastle.Math.BigInteger s1 =
                   new Org.BouncyCastle.Math.BigInteger(1, signature.Take(signature.Length / 2).ToArray());
              Org.BouncyCastle.Math.BigInteger s2 =
                   new Org.BouncyCastle.Math.BigInteger(1, signature.Skip(signature.Length / 2).ToArray());

              var v = new Asn1EncodableVector();
              v.Add(new DerInteger(s1));
              v.Add(new DerInteger(s2));
              byte[] derSignature = new DerSequence(v).GetDerEncoded();
              Console.WriteLine("Formated signature: {0}", signature.ToHexString());
              return derSignature;
         }
```

```
        public X509Certificate ReadCertificateFromCard (byte[] selectApdu)
        {
// Read the certificate from the card
            TransmitApdu (selectApdu, "select cert");
            List<byte> certBin = new List<byte>();
            byte block = 0;
            bool blockRead = true;
            do {
            byte[] apdu = APDU_READBINARY;

// set the offset of the READ BINARY APDU
            apdu[2] = block;
            byte[] newBlock = TransmitApdu (APDU_READBINARY, null);
            if(newBlock.Length > 0) {
                certBin.AddRange(newBlock);
                blockRead = true;
                block++;
            } else
                blockRead = false;
            } while(blockRead);
            X509CertificateParser p = new X509CertificateParser();
            X509Certificate certificate = p.ReadCertificate (certBin.ToArray());
            return certificate;
        }

// Transmit an APDU to the Smartcard
        private byte[] TransmitApdu(byte[] apdu, string logPrefix)
        {
            byte[] responseBuffer = new byte[512];
            SCardError error = Reader.Transmit (Pci, apdu, ref responseBuffer);
            if (error != SCardError.Success) {
                throw new PCSCException (error, SCardHelper.StringifyError (error));
            }
            int rl = responseBuffer.Length;
            if (responseBuffer.Skip(rl - 2).Take(2).ToArray().ToHexString().Equals("9000")) {
                if (logPrefix != null) {
                    Console.WriteLine (logPrefix + ": OK: " + responseBuffer.ToHexString());
                }
            }
            else {
```

```
              if(logPrefix != null)
                Console.WriteLine ((logPrefix ?? "") + ": " + responseBuffer.ToHexString ());
            }
            return responseBuffer.Take(rl - 2).ToArray ();
        }
    }
}

namespace RegisterIntegration.Extensions
{
    public static class Extensions
    {
        public static byte[] ToByteArray(this string hex)
        {
            return Enumerable.Range(0, hex.Length)
                .Where(x => x % 2 == 0)
                    .Select(x => Convert.ToByte(hex.Substring(x, 2), 16))
                    .ToArray();
        }

        public static string ToHexString(this byte[] ba)
        {
            StringBuilder hex = new StringBuilder(ba.Length * 2);
            foreach (byte b in ba)
                hex.AppendFormat("{0:x2}", b);
            return hex.ToString();
        }
    }
}
```

4.2 JAVA

4.2.1 PROGRAM.JAVA

```
package eu.globaltrust.signaturesample;

import java.io.File;
import java.io.FileInputStream;
import java.nio.charset.Charset;
```

```
import java.security.Signature;
import java.security.cert.CertificateFactory;
import java.security.cert.X509Certificate;

public class Program {

    public static void main(String[] args) throws Exception {

        java.security.Security.addProvider(new org.bouncycastle.jce.provider.BouncyCastleProvider());

// Signature input
        String s = "crinoline";
        byte[] data = s.getBytes(Charset.forName("UTF-8"));

// Determine key type (RSA vs. ECDSA)
// RSA by default, ECDSA if "ECDSA" is given as first argument to the program.
        Signer.KeyType keyType = Signer.KeyType.Rsa;
        System.out.println(args.length);
        if (args.length > 0) {
            if (args[0].toUpperCase().equals("ECDSA"))
                keyType = Signer.KeyType.Ecdsa;
        }

// Sign data with Smartcard
        Signer signer = new Signer();
        byte[] signature = signer.sign(data, keyType);

// in case of RKS-CARD use Base64-Encoding! uncomment!
//        byte[] signatureBase64 = org.bouncycastle.util.encoders.Base64.encode(signature);

// Load public-key from user certificate on card
        X509Certificate certificate = signer.readCertificateFromCard(Signer.APDU_SELECT_USER_CERTEF);
        Signature verifier = null;

// Verify signature
        if (keyType == Signer.KeyType.Rsa)
            verifier = Signature.getInstance("SHA256withRSA");
        else if (keyType == Signer.KeyType.Ecdsa) {
            verifier = Signature.getInstance("SHA256withECDSA");
            signature = Signer.derEncodeSignature(signature);
        }
```

```
        verifier.initVerify(certificate.getPublicKey());
        verifier.update(data);
        boolean isSignatureVerified = verifier.verify(signature);
        System.out.println("signatureVerified: " + isSignatureVerified);

// Blankline Only
        System.out.println();

// serialnumber of user certifikate
        System.out.println("user certificate serial#: " + certificate.getSerialNumber());

// Load public-key from SubCA certificate on card
        X509Certificate caCertificate = signer.readCertificateFromCard(Signer.APDU_SELECT_CA_CERTEF);

// serialnumber of SubCA certifikate
        System.out.println("ca certificate serial#: " + caCertificate.getSerialNumber());
    }

}
```

4.2.2 SIGNER.JAVA

```
package eu.globaltrust.signaturesample;

import org.apache.commons.codec.DecoderException;
import org.apache.commons.codec.binary.Hex;
import org.bouncycastle.asn1.ASN1EncodableVector;
import org.bouncycastle.asn1.DERInteger;
import org.bouncycastle.asn1.DERSequence;
import org.bouncycastle.asn1.nist.NISTObjectIdentifiers;
import org.bouncycastle.asn1.x509.*;

import javax.smartcardio.*;
import java.math.BigInteger;
import java.security.MessageDigest;
import java.util.*;
import java.io.*;
import java.security.cert.*;

public class Signer {
```

```
    public enum KeyType {
        Rsa,
        Ecdsa,
    }

// The PIN for CardOS v5.3 cards is BCD encoded and padded with 0xff bytes to length off 8 byte
// "26123456" = 2: fixed, 6: length of PIN, 123456: BCD-coded PIN + "ffffffff" (Example only!! Replace with deliverd PIN!!)
// "2812345678" = 2: fixed, 8: length of PIN, 12345678: BCD-coded PIN + "ffffff" (Example only!! Replace with deliverd PIN!!)
    public static final String PIN_BCD = "26123456" + "ffffffff";

// The PIN for CardOS v5.0 cards is UTF-8 encoded and padded with 0xff bytes to the maximal length of 12
// "313233343536" = ASCII-Hex-Value of "123456" (Example only!! Replace with deliverd PIN!!)
    public static final String PIN_PADDED = "313233343536" + "ffffffffffff";

    public static final byte[] APDU_SERIAL_NO = toByteArray("00ca018108");
    public static final byte[] APDU_SELECT_QESDF = toByteArray("00a40804021fff");
    public static final byte[] APDU_SELECT_USER_CERTEF =  toByteArray("00a40804041160c001");
    public static final byte[] APDU_SELECT_CA_CERTEF =  toByteArray("00a408040411606301");
    public static final byte[] APDU_PIN_COUNT = toByteArray("00200081");
    public static final byte[] APDU_PIN_VERIFY_50 = toByteArray(("002000810c" + PIN_PADDED));
    public static final byte[] APDU_PIN_VERIFY_53 = toByteArray(("0020008108" + PIN_BCD));
    public static final byte[] APDU_MSE = toByteArray("0022F304");
    public static final byte[] APDU_GETDATA_CARDOSVERSION = toByteArray("00ca018202");
    public static final byte[] APDUHEADER_PSO_CDS = toByteArray("002a9e9a");
    public static final byte[] APDU_READBINARY = toByteArray("00B000000000100");

    private CardChannel channel;
    private String cardOsVersion = "";

    public Signer() throws Exception {

// Display the list of terminals
        TerminalFactory factory = TerminalFactory.getDefault();
        List<CardTerminal> terminals = factory.terminals().list();
        System.out.println("Terminals: " + terminals);
        for (int i = 0; i < terminals.size(); i++) {
            CardTerminal terminal = terminals.get(i);
// Reader-Example ONLY!!
            if (terminal.getName().toUpperCase().contains("CHERRY") || terminal.getName().toUpperCase().contains("OMNIKEY")
                    || terminal.getName().toUpperCase().contains("GEMALTO") ) {
```

```
                Card card = terminal.connect("*");
                System.out.println("card: " + card);
                this.channel = card.getBasicChannel();
                break;
            }
        }
// Find out the CardOS version of the smartcard (5.0 and 5.3 are supported)
        byte[] cardOsVersionEncoded = transmitApdu(APDU_GETDATA_CARDOSVERSION, null);
        if(cardOsVersionEncoded[0] == (byte)0xC9 && cardOsVersionEncoded[1] == (byte)0x03)
            cardOsVersion = "5.3";
        else if(cardOsVersionEncoded[0] == (byte)0xc9 && cardOsVersionEncoded[1] == (byte)0x01)
            cardOsVersion = "5.0";
    }

    /**
     * Creates actual PSO_CDS APDU based on header and signature input.
     *
     * @param sigData the hash/data that is signed
     * @return the PSO_CDS APDU
     */
    public static byte[] psoCdsApdu(byte[] sigData) {
        byte[] apdu = new byte[4 + 3 + sigData.length + 2];
        System.arraycopy(APDUHEADER_PSO_CDS, 0, apdu, 0, APDUHEADER_PSO_CDS.length);
        System.arraycopy(new byte[]{0x00, 0x00, (byte) sigData.length}, 0, apdu, 4, 3);
        System.arraycopy(sigData, 0, apdu, 7, sigData.length);
        System.arraycopy(new byte[]{0x01, 0x02}, 0, apdu, 7 + sigData.length, 2);
        return apdu;
    }

    /**
     * Hashes and signs the input data using a Smartcard.
     *
     * @param data    to be hashed and signed
     * @param keyType RSA or ECDSA
     * @return signature as byte-array
     * @throws Exception
     */
    public byte[] sign(byte[] data, KeyType keyType) throws Exception {

// Hash input data using SHA-256
        MessageDigest digest = MessageDigest.getInstance("SHA-256");
```

```
        byte[] hash = digest.digest(data);

// Prepare hash as input for signature operation
        byte[] signatureData = null;
        if (keyType == KeyType.Rsa) {
            DigestInfo digestInfo = new DigestInfo(new AlgorithmIdentifier(NISTObjectIdentifiers.id_sha256), hash);
            signatureData = digestInfo.getEncoded();
        } else if (keyType == KeyType.Ecdsa) {
            signatureData = hash;
        }

// Create signature APDU
        byte[] psoCdsApdu = psoCdsApdu(signatureData);
        System.out.println("hash: " + toHexString(hash));

// Transmit APDUs to card
        transmitApdu(APDU_SERIAL_NO, "serialNumber");
        transmitApdu(APDU_SELECT_QESDF, "select");
        transmitApdu(APDU_PIN_COUNT, "pinCount");
        if(cardOsVersion == "5.0")
            transmitApdu (APDU_PIN_VERIFY_50, "verify");
        else if(cardOsVersion == "5.3")
            transmitApdu (APDU_PIN_VERIFY_53, "verify");

// Restore MSE only needed for RSA cards
        if (keyType == KeyType.Rsa)
            transmitApdu(APDU_MSE, "mse");

// Create the signature unformatted
        byte[] signature = transmitApdu(psoCdsApdu, "signature");
        System.out.println("well done");
        return signature;
    }


// Encode ECDSA signature using DER (not for RKS-CARD!)
    public static byte[] derEncodeSignature(byte[] signature) throws IOException {
        byte[] sigPart1 = new byte[signature.length / 2];
        byte[] sigPart2 = new byte[signature.length / 2];
        System.arraycopy(signature, 0, sigPart1, 0, sigPart1.length);
        System.arraycopy(signature, sigPart1.length, sigPart2, 0, sigPart2.length);
```

```
        BigInteger s1 = new BigInteger(1, sigPart1);
        BigInteger s2 = new BigInteger(1, sigPart2);
        ASN1EncodableVector v = new ASN1EncodableVector();
        v.add(new DERInteger(s1));
        v.add(new DERInteger(s2));
        byte[] derSignature = new DERSequence(v).getEncoded();
        System.out.println("Formatted signature: " + toHexString(signature));
        return derSignature;
    }

    public X509Certificate readCertificateFromCard (byte[] selectApdu) throws CardException, CertificateException
        {
// Read the certificate from the card
        transmitApdu (selectApdu, "select cert");
        List<Byte> certBin = new ArrayList<Byte>();
        byte block = 0;
        boolean blockRead = true;
        do {
            byte[] apdu = APDU_READBINARY;
            // set the offset of the READ BINARY APDU
            apdu[2] = block;
            byte[] newBlock = transmitApdu (APDU_READBINARY, null);
            if(newBlock.length > 0) {
                for(Byte b : newBlock)
                    certBin.add(b);
                blockRead = true;
                block++;
            } else
                blockRead = false;
        } while(blockRead);
        byte[] certBinArray = new byte[certBin.size()];
        for(int i=0;i<certBin.size();++i)
            certBinArray[i] = certBin.get(i);
            CertificateFactory cf = CertificateFactory.getInstance("X.509");
            X509Certificate certificate = (X509Certificate) cf.generateCertificate(new ByteArrayInputStream(certBinArray));
            return certificate;
        }

    /**
     * Transmit an APDU to the Smartcard
     * @param apdu APDU that is transmitted
```

```
     * @param logPrefix prefix for log messages
     * @return response data, if possible
     * @throws CardException
     */
    private byte[] transmitApdu(byte[] apdu, String logPrefix) throws CardException {
        CommandAPDU cApdu = new CommandAPDU(apdu);
        ResponseAPDU rApdu = channel.transmit(cApdu);
        int sw = rApdu.getSW();
        if (sw == 0x9000) {
            if (logPrefix != null) {
                System.out.println(logPrefix + ": OK: " + toHexString(rApdu.getBytes()));
            }
        } else {
            if(logPrefix != null)
                System.out.println(logPrefix + ": " + toHexString(rApdu.getBytes()));
        }
        return rApdu.getData();
    }

    public static byte[] toByteArray(String hex) {
        try {
            return Hex.decodeHex(hex.toCharArray());
        } catch (DecoderException e) {
            throw new RuntimeException(e);
        }
    }

    public static String toHexString(byte[] ba) {
        return Hex.encodeHexString(ba);
    }
}
```

4.3 C (C++)

4.3.1 SAMPLE

```
#include <stdio.h>
#include <stdlib.h>
#include <winscard.h>
```

```
#include <openssl/ssl.h>
#include <openssl/evp.h>
#include <openssl/pem.h>
#include <openssl/x509.h>
#include <openssl/err.h>
#include <openssl/bn.h>
#include <string.h>

// Prüffunktion
// Checks the return value of the PC/SC card operations
static void check(LONG returnvalue, char *reason) {
  if (returnvalue != SCARD_S_SUCCESS) {
    printf("%s: %s - %lx\n", reason, pcsc_stringify_error(returnvalue),
           returnvalue);
  }
}

// Printfunktion
// Prints the response of the PC/SC operations
void print(DWORD dwRecvLength, BYTE *pbRecvBuffer, char *purpose) {
  unsigned int i;
  printf("response %s (%d byte): ", purpose, dwRecvLength);
  for (i = 0; i < dwRecvLength; i++) {
    printf("%02X", pbRecvBuffer[i]);
  }
  printf("\n");
}

// Start Programm
int main(int argc, char *argv[]) {

// Definition Variable
  LONG returnvalue;
  SCARDCONTEXT hContext;
  LPTSTR mszReaders;
  SCARDHANDLE hCard;
  DWORD dwReaders, dwActiveProtocol, dwRecvLength;
  SCARD_IO_REQUEST pioSendPci;
  BYTE pbRecvBuffer[512];
// The PIN for CardOS v5.0 cards is UTF-8 encoded and padded with 0xff bytes to the maximal length of 12
  BYTE verify_50[] = {0x00, 0x20, 0x00, 0x81, 0x0c, 0x31, 0x32, 0x33, 0x34,
```

```
                    0x35, 0x36, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff};
// The PIN for CardOS v5.3 cards is BCD encoded and padded with 0xff bytes to 8 byte
// Example for PIN 123456:
  BYTE verify_53[] = {0x00, 0x20, 0x00, 0x81, 0x08,
            0x26, 0x12, 0x34, 0x56, 0xff, 0xff, 0xff, 0xff};
// Example for PIN 547722:
  BYTE verify_53[] = {0x00, 0x20, 0x00, 0x81, 0x08,
            0x26, 0x54, 0x77, 0x22, 0xff, 0xff, 0xff, 0xff};
// Variable for Serialnumber of Smartcard
  BYTE serialno[] = {0x00, 0xca, 0x01, 0x81, 0x08};
// Variable for Application-Identifier Atos-Applikation QES 3.6  Smartcardinformation im Überblick
  BYTE select_qesdf[] = {0x00, 0xa4, 0x08, 0x04, 0x02, 0x1f, 0xff};
// Variable for Application-Identifier Globaltrust-Applikation CERT End-Zertifikat
// 3.6 Smartcardinformation im Überblick
  BYTE select_certef[] = {0x00, 0xa4, 0x08, 0x04, 0x04, 0x11, 0x60, 0xc0, 0x01};
// Variable for PIN-Count 3.8.2    Verify
  BYTE pincount[] = {0x00, 0x20, 0x00, 0x81};
// Variable for 3.8.5   Manage Security Environment (MSE)
// ONLY for RSA [V50]
  BYTE mse[] = {0x00, 0x22, 0xF3, 0x01};
// Variable for 3.8.6   Perform security operation, Compute digital signature (PSO_CDS)
// erforderlich für die eigentliche Signatur des Hash-Wertes
// Hinweis! Hash-Wert muss voraberzeugt sein!
  BYTE pso_cds[] = {0x00, 0x2a, 0x9e, 0x9a};
// length of pso_cds data
  BYTE length[] = {0x00, 0x00, 0x20};
// data to be signed with demovalue "Hello World!"
  const char data[] = "Hello World!";
// expected return length in byte
  BYTE le[] = {0x01, 0x20};
//
  int cardOsVersion;
  BYTE getdata_cardosversion[] = {0x00, 0xca, 0x01, 0x82, 0x02};
  BYTE readbinary[] = {0x00, 0xB0, 0x00, 0x00, 0x00, 0x01, 0x00};

// Step I: Create hash from data which should signed
// creation without using Smartcard
// hash - digest the data using SHA-256
  EVP_MD_CTX *mdctx;
  const EVP_MD *md;
  unsigned char md_value[256];
```

```
  unsigned int md_len, i;
  md = EVP_sha256();
  mdctx = EVP_MD_CTX_create();
  EVP_DigestInit_ex(mdctx, md, NULL);
  EVP_DigestUpdate(mdctx, data, strlen(data));
  EVP_DigestFinal_ex(mdctx, md_value, &md_len);
  EVP_MD_CTX_destroy(mdctx);
  printf("Digest is: ");
  for (i = 0; i < md_len; i++) {
    printf("%02x", md_value[i]);
  }
  printf("\n");
  EVP_cleanup();
```

// Step II: Einrichten Umgebung (inkl. Cardreader)

```
// Establish Context
  returnvalue =
      SCardEstablishContext(SCARD_SCOPE_SYSTEM, NULL, NULL, &hContext);
  check(returnvalue, "EstablishContext");
#ifdef SCARD_AUTOALLOCATE
  dwReaders = SCARD_AUTOALLOCATE;
// List available card readers
  returnvalue =
      SCardListReaders(hContext, NULL, (LPTSTR)&mszReaders, &dwReaders);
  check(returnvalue, "ListReaders1");
#else
  returnvalue = SCardListReaders(hContext, NULL, NULL, &dwReaders) mszReaders =
      malloc(sizeof(char) * dwReaders);
  check(returnvalue, "ListReaders2");
  returnvalue = SCardListReaders(hContext, NULL, mszReaders, &dwReaders)
      check(returnvalue, "ListReaders3");
#endif
  printf("reader name: %s\n", mszReaders);
// Connect to a card reader
  returnvalue = SCardConnect(hContext, mszReaders, SCARD_SHARE_SHARED,
                             SCARD_PROTOCOL_ANY, &hCard,
                             &dwActiveProtocol);
  check(returnvalue, "Connect");
  switch (dwActiveProtocol) {
  case SCARD_PROTOCOL_T0:
    pioSendPci = *SCARD_PCI_T0;
```

```
    break;
  case SCARD_PROTOCOL_T1:
    pioSendPci = *SCARD_PCI_T1;
    break;
  }

// Step IIIa: Find out the CardOS Version
  dwRecvLength = sizeof(pbRecvBuffer);
  returnvalue = SCardTransmit(hCard, &pioSendPci, getdata_cardosversion, sizeof(getdata_cardosversion),
                              NULL, pbRecvBuffer, &dwRecvLength);
  check(returnvalue, "Get CardOS Version");
  if(dwRecvLength == 4) {
     if(pbRecvBuffer[0] == 0xc9 && pbRecvBuffer[1] == 0x03)
        cardOsVersion = 53;
     else if(pbRecvBuffer[0] == 0xc9 && pbRecvBuffer[1] == 0x01)
             cardOsVersion = 50;
  }else {
     printf("ERROR: Could not read the CardOS version.\n");
     return 1;
  }
  printf("CardOS version: %d\n", cardOsVersion);

// Step IIIb: Find out the Smartcard-SerialNumber
// transmit request for Serial Number
  dwRecvLength = sizeof(pbRecvBuffer);
  returnvalue = SCardTransmit(hCard, &pioSendPci, serialno, sizeof(serialno),
                              NULL, pbRecvBuffer, &dwRecvLength);
  check(returnvalue, "TransmitSerialNO");
  print(dwRecvLength, pbRecvBuffer, "Serial Number");

// Step IIIc: transmit request to select Aplication
  dwRecvLength = sizeof(pbRecvBuffer);
  returnvalue = SCardTransmit(hCard, &pioSendPci, select_qesdf, sizeof(select_qesdf), NULL,
                              pbRecvBuffer, &dwRecvLength);
  check(returnvalue, "TransmitSelect");
  print(dwRecvLength, pbRecvBuffer, "Select QES DF");

// Step IIId: transmit request to get PIN count
  dwRecvLength = sizeof(pbRecvBuffer);
  returnvalue = SCardTransmit(hCard, &pioSendPci, pincount, sizeof(pincount),
                              NULL, pbRecvBuffer, &dwRecvLength);
```

```
  check(returnvalue, "TransmitPinCount");
  print(dwRecvLength, pbRecvBuffer, "PinCount");

// Step IIIe: transmit request to verify PIN
// Correct Variable verify_50 or verify_53 necessary!
  dwRecvLength = sizeof(pbRecvBuffer);
  BYTE* verify;
  int verifyLen;
  if(cardOsVersion == 50) {
    verify = verify_50;
    verifyLen = sizeof(verify_50);
  } else if (cardOsVersion == 53) {
    verify = verify_53;
    verifyLen = sizeof(verify_53);
  }
  returnvalue = SCardTransmit(hCard, &pioSendPci, verify, verifyLen, NULL,
                              pbRecvBuffer, &dwRecvLength);
  check(returnvalue, "TransmitVerify");
  print(dwRecvLength, pbRecvBuffer, "Verify");

 // Step IIIf: transmit request to manage security environment
// RSA only! not needed for ECDSA cards
// dwRecvLength = sizeof(pbRecvBuffer);
// returnvalue = SCardTransmit(hCard, &pioSendPci, mse, sizeof(mse), NULL,
//                             pbRecvBuffer, &dwRecvLength);
// check(returnvalue, "Transmitmse");
// print(dwRecvLength, pbRecvBuffer, "MSE");

// Step IV: Definition Signatur-Request (pso_cds)
// 3.8.6  Perform security operation, Compute digital signature (PSO_CDS)
// construct pso_cds request
  int SSL_libary_init(void);
  SSL_CTX *SSL_CTX_new(const SSL_METHOD *method);
  BYTE pso_cds_complete[sizeof(pso_cds) + sizeof(length) + md_len + sizeof(le)];
//
  size_t offset = 0;
  memcpy(pso_cds_complete + offset, pso_cds, sizeof(pso_cds));
  offset += sizeof(pso_cds);
//
  memcpy(pso_cds_complete + offset, length, sizeof(length));
  offset += sizeof(length);
```

```
//
  memcpy(pso_cds_complete + offset, md_value, md_len);
  offset += md_len;
//
  memcpy(pso_cds_complete + offset, le, sizeof(le));
  offset += sizeof(le);
//
  size_t v;
  printf("\nSending PSO_CDS Apdu: ");
  for (v = 0; v < (offset); v++)
    printf("%02x", pso_cds_complete[v]);
  printf("\n");
```

// Step IV: transmit Signatur-Request (pso_cds)

```
// 3.8.6  Perform security operation, Compute digital signature (PSO_CDS)
  dwRecvLength = sizeof(pbRecvBuffer);
  returnvalue = SCardTransmit(hCard, &pioSendPci, pso_cds_complete, offset,
                              NULL, pbRecvBuffer, &dwRecvLength);
  check(returnvalue, "Transmitpso_cds");
  print(dwRecvLength, pbRecvBuffer, "pso_cds");
  if(dwRecvLength == 0) {
    printf("ERROR: Signature is empty.\n");
    return 1;
  }
  unsigned char signature[dwRecvLength - 2];
  memcpy(signature, pbRecvBuffer, dwRecvLength - 2);
  print(sizeof(signature), signature, "signature");
```

// Step V: Check signature

```
// Step Va: get certificate from file and  public key from certificate
// read the certificate from the card
// Select the file holding the certificate
  dwRecvLength = sizeof(pbRecvBuffer);
  returnvalue = SCardTransmit(hCard, &pioSendPci, select_certef, sizeof(select_certef),
                              NULL, pbRecvBuffer, &dwRecvLength);
  check(returnvalue, "Select cert EF");
// read the first block and obtain the certificate's length from it
  dwRecvLength = sizeof(pbRecvBuffer);
  returnvalue = SCardTransmit(hCard, &pioSendPci, readbinary, sizeof(readbinary),
                              NULL, pbRecvBuffer, &dwRecvLength);
  check(returnvalue, "Read binary");
```

```
// the encoded certificate starts with 0x30 0x82 followed by 2 length bytes
  int certLength = 256*pbRecvBuffer[2]+pbRecvBuffer[3]+4;
  printf ("Certificate's length is %d byte.", certLength);
  unsigned char certRaw[certLength];
// read 256 byte blocks
//the blocksize is already specified in the APDU
  int blockSize = 256;
// store the first block
  memcpy(certRaw, pbRecvBuffer, blockSize);
  int blocksRead = 1;
  int bytesRead = 256;
  while( (certLength-bytesRead) >= 256 ) {
//P1 holds the hi byte of the offset
    readbinary[2] = blocksRead;
    returnvalue = SCardTransmit(hCard, &pioSendPci, readbinary, sizeof(readbinary),
              NULL, pbRecvBuffer, &dwRecvLength);
    check(returnvalue, "Read binary");
    memcpy(certRaw+bytesRead, pbRecvBuffer, blockSize);
    bytesRead += blockSize;
    blocksRead++;
  }
//read the remaining bytes
  int remainingBytes = certLength-bytesRead;
  if(remainingBytes > 0) {
//P1 holds the hi byte of the offset
    readbinary[2] = blocksRead;
    readbinary[5] = 0x00;
    readbinary[6] = remainingBytes;
    returnvalue = SCardTransmit(hCard, &pioSendPci, readbinary, sizeof(readbinary),
           NULL, pbRecvBuffer, &dwRecvLength);
    check(returnvalue, "Read binary");
    memcpy(certRaw+bytesRead, pbRecvBuffer, remainingBytes);
    bytesRead+=remainingBytes;
  }
  printf("\n");
  print(bytesRead, certRaw, "user certificate");
  unsigned char* certBuf = certRaw;
  X509 *cert = d2i_X509(NULL, &certBuf, certLength);
  char subject[512];
  X509_NAME_oneline(X509_get_subject_name(cert), subject, 512);
  printf("Subject: %s\n", subject);
```

```
  EVP_PKEY *pubkey;
//
  pubkey = X509_get_pubkey(cert);
  if (pubkey == NULL)
    printf("Public key was not parsed");
//
// Step Vb: construct ECDSA_SIG for sha_256 and ECDSA algorithm
  ECDSA_SIG *ecdsa_sig;
  ecdsa_sig = ECDSA_SIG_new();
// Split Signature in half and convert to BIGNUM
  BN_bin2bn(signature, 32, ecdsa_sig->r);
  BN_bin2bn(signature + 32, 32, ecdsa_sig->s);
// get the Elliptic Curve public key
  EC_KEY *eckey = EVP_PKEY_get1_EC_KEY(pubkey);
  if (eckey == NULL)
    printf("eckey is null\n");
//
// Step Vc: verify Signature with Publickey
  size_t ret = ECDSA_do_verify(md_value, md_len, ecdsa_sig, eckey);
  if (ret == -1) {
    printf("return -1 unknown \n");
  } else if (ret == 0) {
    printf("return 0 incorrect signature \n");
  } else {
    printf("return 1 correct signature \n");
  }

// Step X: Other
// write errors to file error.txt
  FILE *bp = fopen("error.txt", "w+");
  ERR_print_errors_fp(bp);
//
// free Variables
  X509_free(cert);
  //    fclose(fp);
  EVP_PKEY_free(pubkey);
  EVP_cleanup();
//
// Release connection
  returnvalue = SCardDisconnect(hCard, SCARD_LEAVE_CARD);
  check(returnvalue, "Disconnect");
```

```
//
// Release context
#ifdef SCARD_AUTOALLOCATE
  returnvalue = SCardFreeMemory(hContext, mszReaders);
  check(returnvalue, "FreeMemory");
#else
  free(mszReaders);
#endif
  returnvalue = SCardReleaseContext(hContext);
  check(returnvalue, "ReleaseContext");
  exit(1);
}
```

4.3.2 MAKEFILE

```
#Makefile
#Linux
PCSC_CFLAGS:=$(shell pkg-config --cflags libpcsclite)
LDFLAGS:=$(shell pkg-config --libs libpcsclite)

CFLAGS+=$(PCSC_CFLAGS)
CFLAGS+=-Wall
CFLAGS+=-lssl
CFLAGS+=-lcrypto
sample: sample.c

clean:rm -f sample
```

4.4 ÜBERSICHT INTERNE TEST DES MUSTERCODES

Unter folgenden Konfigurationen erfolgten bei GLOBALTRUST Detailtests.

4.4.1 STANDARDKONFIGURATION

- **Java auf Linux** *MIT USB CARDREADER*
  OS: Opensuse 13.1(i586)
  Reader: Cherry SmartTerminal ST-2000, 21.9.2015
  Treiber: libccid
  Java: OpenJDK 1.7.0_85
  Schlüsseltyp: ECDSA & RSA

- **C# auf Linux** *MIT USB CARDREADER*
  OS: Opensuse 13.1(i586)
  Reader: Cherry SmartTerminal ST-2000, 21.9.2015
  Treiber: libccid
  Mono: mono-3.2.8
  Schlüsseltyp: ECDSA & RSA

- **C/C++ auf Linux** MIT USB CARDREADER
  OS: Debian amd64
  Reader: Cherry SmartTerminal ST-2000, 02.10.15
  Software/Treiber: libusb-1.0.19, pcsc-lite-1.8.13, ctpcsc_v1.0.8_linux_64bit, ccid-1.4.18.
  Schlüsseltyp: ECDSA

4.4.2 SONDERKONFIGURATION - SERIELLER KARTENLESER

- **Java auf Linux** *MIT SERIELLEM CARDREADER*
  OS: Opensuse 13.2(i586)
  Reader: Omnikey 3111 Seriell, 14.08.15
  Treiber: ifdok_un 311_lnx_i686-3.6.0
  Java: OpenJDK 1.8.0_51
  Schlüsseltyp: RSA

- **C# auf Linux** *MIT SERIELLEM CARDREADER*
  OS: Opensuse 13.2(i586)
  Reader: Omnikey 3111 Seriell, 14.08.15
  Treiber: ifdok_un 311_lnx_i686-3.6.0
  Schlüsseltyp: RSA

- **Java auf Windows** *MIT SERIELLEM CARDREADER*
  OS: Windows 8.1 pro 64bit
  Reader: Omnikey 3111 Seriell, 14.08.15
  Treiber: Omnikey 1.2.20.0, 08.08.13
  Java: JRE 1.8.0_45
  Schlüsseltyp: RSA

- **C# auf Windows** *MIT SERIELLEM CARDREADER*
  OS: Windows 8.1 pro 64bit
  Reader: Omnikey 3111 Seriell, 14.08.15
  Treiber: Omnikey 1.2.20.0, 08.08.13
  Schlüsseltyp: RSA

VERZEICHNIS TABELLEN