Pametne kartice in PKCS #11

Jure Žbontar, 63040307

30. avgust 2008

1 Uvod

Pametne kartice postajajo, vse bolj, del našega vsakdanjega življenja. Da bi razvijalcem aplikacij omogočili enoten dostop do kriptografske strojne opreme potrebujemo uveljavljene standarde, ki nam omogočajo da do pametnih kartic dostopamo na, od proizvajalca, neodvisen način. Primer takšnega standarda je PKCS #11.

V tej seminarski nalogi bomo najprej na splošno spregovorili o pametnih karticah. Ogledali si bomo njihovo zgradbo in kako z njimi komuniciramo. Sledi opis java kartic, ki zaradi močne razširjenosti programskega jezika Java, postajajo vse bolj popularne. Nato bomo na kratko opisali standard PKCS #11, ter ogrodje MUSCLE, ki standard implementira. V naslednjem delu sledi opis različnih uporab ogrodja MUSCLE ter opis programa pkcs11, na koncu pa priložimo še navodila za namestitev ogrodja MUSCLE skupaj z razvojnim okoljem.

2 Pameten kartice

Pametne kartice so plastične kartice, po velikosti identične kreditnim, z zmožnostjo hranjenja in procesiranja podatkov. Vsaka pametna kartica namreč vsebuje prav poseben računalnik, sestavljen iz centralno procesne enote (CPE) in pomnilnika - tako obstojnega (ROM in EEPROM) kot neobstojnega (RAM). Spominski čipi pametnih kartic so izdelani v tehnologiji, ki preprečuje nedovoljen fizičen poseg v notranjost tiskanega vezja. Ob vsakem poizkusu odpiranja čipa se ta avtomatsko uniči in izbriše podatke, ki jih je vseboval. Ker so se pametne kartice prav posebej prijele v aplikacijah povezanih s kriptografijo, jih ima večina vgrajene tudi soprocesorje za izvajanje raznih kriptografskih funkcij (npr. generiranje naključnih števil, RSA, AES, 3DES, EC, ...). V grobem bi lahko pametne kartice razdelili v dve skupini:

Slika 1: Pametne kartice.

Pomnilniške kartice nimajo lastne CPE in zato ne morejo obdelovati podatkov. Namenjene so predvsem hranjenju podatkov. Za nadzor dostopa do pomnilnika skrbi posebna logika.

Mikroprocesorske kartice so inteligentne kartice z lastno CPE in zmožnostjo pisanja, branja in obdelovanja podatkov.

V tej seminarski nalogi se bomo ukvarjali izključno z mikroprocesorskimi karticami, saj lahko z njimi, kot bomo videli v nadaljevanju, storimo veliko več kot pa z navadnimi pomnilniškimi karticami.

Komunikacija s pametno kartico poteka z izmenjavo paketov, ki jih imenujemo APDU (Application Protocol Data Units). APDU vsebuje bodisi ukaz ali pa odgovor. Računalnik in pametna kartica sta v razmerju nadrejeni-podrejeni, kjer vlogo nadrejenega vedno igra računalnik. To pomeni, da je kartica pri komunikaciji pasivna in samo čaka, dokler ne sprejme APDU ukaza. Ukaz se nato na kartici izvrši, za tem pa se zgradi APDU z ustreznim odgovorom in se posreduje nazaj.

obvezna glava				izbirna vsebina		
CLA	INS	P1	P2	Lc	Data	Le

Slika 2: Zgradba APDU ukaza.

Osnovna zgradba APDU ukaza je prikazana na sliki 2. Glava paketa je obvezna in je sestavljena iz štirih polj velikosti 1 bajt:

CLA - Class: s tem bajtom ločimo pakete dveh različnih aplikacij, ki sta hkrati naloženi na kartici. Vsaki aplikaciji določimo svoj lasten CLA, nato pa nam ni več potrebno skrbeti ali je morda INS ukaz že uporabljen v kateri drugi aplikaciji.

INS - Instruction: bajt vsebuje vrednost ukaza, ki naj se izvrši.

P1, P2 - Parameters: s tema dvema bajtoma lahko bolj podrobno opredelimo ukaz ali pa ju uporabimo za prenos podatkov (podobno kot parametri procedur v proceduralnih programskih jezikih).

Glavi sledi telo paketa, ki pa ni obvezno. Sestavljajo ga tri polja: Le, Data in Lc. Le in Lc sta velika po en bajt vsak, polje Data pa je veliko Le bajtov.

Lc: določi velikost polja Data.

Data: vsebuje podatke.

Le: določi število bajtov, ki jih pričakujemo v odgovoru.

Po vsakem prejetem APDU ukazu, kartica odvrne z APDU odgovorom, katerega zgradbo prikazuje slika 3.

Data: polje ni obvezno in je pripeto odgovoru le če smo to zahtevali v ukazu. Velikost polja je Le bajtov.

izbirna vsebina	obvezen rep	
Data	SW1	SW2

Slika 3: Zgradba APDU odgovora.

SW1, SW2: skupaj tvorita status izvršenega ukaza. Vsako polje je veliko po en bajt, torej je status sestavljen iz dveh bajtov (status 0x9000 pomeni, da se je ukaz uspešno izvršil, medtem ko npr. status 0x64XX označuje, da je med izvajanjem prišlo do napake).

2.1 Java kartice

Java kartice so pametne kartice, na katerih lahko izvajamo programe napisane v programskem jeziku Java. Osnovna arhitektura kartic z zmožnostjo izvajanja javine bajtkode je prikazana na sliki 4. Kot je razvidno iz diagrama je javin navidezni stroj implementiran nad plastjo ope-

aplet 1	aplet 2	aplet 3
Javacard Framework		
Javacard Virtual Machine		
operacijski sistem		

Slika 4: Arhitektura java kartic.

racijskega sistema. Navidezni stroj skrije razlike med operacijskimi sistemi in različno strojno opremo in razvijalcu predstavi enoten pogled na kartico. Javacard Framework vsebuje vse javanske razrede definirane v javacard standardu [9], preko katerih dostopamo do funkcionalnosti kartice. Java kartica lahko hkrati vsebuje več java apletov, ki jih med seboj ločimo po številki AID.

Med J2SE, ki teče na domačih računalnikih in Java Card Platform Specification je kar nekaj razlik. Spomnimo se, da so računski viri pametnih kartic zaradi njihove velikosti omejeni, zato bi bilo zahtevati vse funkcionalnosti polne J2SE specifikacije nesmiselno. Jezik Java, ki se izvaja na karticah je zato le podmnožica tistega iz J2SE specifikacije. Izpuščeni so osnovni tipi float, double, long, char, delo z nitmi, sinhronizacija, dinamično nalaganje razredov, kloniranje objektov, ... Pomembna razlika je tudi to, da se prostor za objekte ustvarjene s ključno besedo `new` alocira v počasnem EEPROM pomnilniku namesto na kopici v RAM-u [1]. Prostor v RAM-u se alocira le za lokalne spremenljivke na skladu in pa za tabele ustvarjene s posebnimi metodami iz paketa `javacard.framework.JCSystem`.

2.1.1 Življenjski cikel apletov

Vsak aplet, ki ga želimo izvajati na kartici mora biti posredno ali neposredno izpeljan iz razreda `javacard.framework.Applet`. Pomembne metode, ki jih pri tem podeduje so:

- `void deselect()`
- `static void install(byte[], short, byte)`
- `abstract process(APDU)`
- `protected void register()`
- `boolean select()`

[1] Operacija pisanja na EEPROM se izvrši približno 1000 krat počasneje kot pisanje na RAM.

Življenje apleta se prične ko ga pravilno namestimo na kartico in registriramo. To storimo tako, da v telesu metode `install`, ki jo pokliče java VM ob namestitvi apleta, izvršimo ukaz `register()`. Registracija je uspešna, če se metoda `register()` izvrši brez napake. Po namestitvi je aplet v neizbranem stanju in zato še ne more sprejemati APDU ukazov. Aplet lahko izberemo tako, da kartici pošljemo poseben APDU ukaz, ki v polju Data vsebuje AID apleta, ki ga želimo izbrati, CLA paketa je enak 0x00, INS pa 0xA4. Aplet, ki smo ga izbrali je o tem obveščen - VM izvrši njegovo metodo `select()`. Ko metoda `select()` zaključi izvajanje, z izhodno vrednostjo `true`, je aplet v izbranem stanju in lahko od sedaj naprej sprejema APDU ukaze. Ko se apleta naveličamo, ga lahko 'odizberemo' ali pa kar izbrišemo iz EEPROM pomnilnika.

2.1.2 Ugani število

Ker je učenje na primeru zelo zabavno, si bomo sedaj ogledali preprost primer java apleta. Napisali bomo znano igrico iz začetniških programerskih knjig, *ugani število*. Pravila igre so preprosta: na kartici se ustvari naključno število r, veliko 4 bajte, ki ga je potrebno uganiti. V vsakem krogu igralec poizkusi srečo s številom n, kartica pa odgovori, ali je $n > r$, $n < r$ ali pa $n = r$, kar pomeni da smo zmagali.

Za igro *ugani število* definirajmo dva APDU ukaza (tabela 1). Prvi ukaz primerja število n z naključno izbranim r. Kartica odgovori z 0x0001 (v paketu odgovora je SW1 = 0x00 in SW2 = 0x01), `0xffff` oziroma 0x9000, če je $n < r$, $n > r$ ali $n = r$, zaporedoma. Ko pa se igranja naveličamo pa lahko z drugim ukazom število r kar preberemo iz kartice.

CLA	INS	P1	P2	Lc	Data	Le	opis
0xB0	0x10	-	-	0x04	n	-	Preveri število n
0xB0	0x20	-	-	-	-	0x04	Goljufaj - preberi število r

Tabela 1: Ukazi igre *ugani število*. '-' pomeni, da se vrednost ignorira.

Poglejmo si sedaj kako zgleda izvorna koda apleta.

```
package org.test;

import javacard.framework.APDU;
import javacard.framework.Applet;
import javacard.framework.ISO7816;
import javacard.framework.ISOException;
import javacard.framework.Util;
import javacard.security.RandomData;

public class Hello extends Applet {
  RandomData rand = RandomData.getInstance(RandomData.ALG_SECURE_RANDOM);
  byte[] secret= new byte[4];

  public static void install(byte[] a, short o, byte l) {
    new Hello().register();
  }

  public boolean select() {
    // zacni novo igro - izberi r
    rand.generateData(secret, (short) 0, (short) 4);
    return true;
  }
```

```
  public void process(APDU apdu) {
    // prejel APDU z INS = A4 - ignoriraj
    if (selectingApplet())
      return;

    byte[] buf = apdu.getBuffer();
    // preveri polje CLA
    if (buf[ISO7816.OFFSET_CLA] != (byte) 0xB0)
      ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);

    // izvedi ukaz
    switch (buf[ISO7816.OFFSET_INS]) {
    case (byte) 0x10: // ugibaj
      byte byteRead = (byte)(apdu.setIncomingAndReceive());
      // preveri dolzino vhodnega stevila
      if (byteRead != 4)
        ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
      /* primerja r in n. Metuda Util.arrayCompare tu ni uporabna,
       * saj interpretira stevila kot predznacena.
       */
      for (short i = 0; i < 4; i++) {
        // da preprecimo razsiritev predznaka je potreben & 0xff
        short res = (short)
          ((secret[i] & 0xff) - (buf[ISO7816.OFFSET_CDATA+i] & 0xff));
        if (res > 0)
          ISOException.throwIt((short)0x0001);
        else if (res < 0)
          ISOException.throwIt((short)0xffff);
      }
      break;
    case (byte) 0x20: // goljufaj
      apdu.setOutgoing();
      apdu.setOutgoingLength((short)4);
      Util.arrayCopy(secret, (short)0, buf, (short)0, (short)4);
      apdu.sendBytes((short)0, (short)4);
      break;
    default:
      /* tega ukaza ne poznamo */
      ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
    }
  }
}
```

Dialog APDU-jev med igralčevim računalnikom in kartico pa zgleda približno tako:

```
// Izberemo aplet (INS = A4, AID apleta pa 74 65 73 74 61 70 70)
=> 00 A4 04 00 07 74 65 73 74 61 70 70 00
<= 90 00
// Preveri stevilo 0x70000000
=> B0 10 00 00 04 70 00 00 00 00
<= 00 01
// Aha, stevilo 0x70000000 je premajhno, kaj pa 0xe0000000
=> B0 10 00 00 04 E0 00 00 00 00
<= FF FF
// 0xE0000000 je preveliko

...
```

```
// Nekaj poizkusov kasneje
=> B0 10 00 00 04 D3 CB 97 02 00
<= 90 00
// zmaga!!! 0xD3CB9702 je iskano stevilo
```

3 PKCS #11

Da bi razvijalcem kriptografskih aplikacij omogočili enoten dostop do kriptografskih naprav, so znanstveniki v podjetju RSA Laboratories razvili standard PKCS #11 [8]. Standard definira od operacijskega sistema neodvisen API, imenovan "cryptoki", za uporabo pametnih kartic in ostalih naprav z zmožnostjo šifriranja podatkov. Podatkovni tipi in funkcije so definirane v programskem jeziku ANSI C. Cilj standarda je tudi deljenje virov, kar pomeni, da lahko več aplikacij hkrati dostopa do iste pametne kartice. V cryptoki API-ju so definirane funkcije za:

- vodenje seje
- upravljanje z objekti
- šifriranje
- zgoščevanje
- podpisovanje

Podroben opis standarda presega okvire te seminarske naloge, zato bralca napotimo na [8], kjer si lahko prebere uvodna poglavja standarda PKCS #11, ki so napisana zelo čitljivo. Če pa vas zanima praktična uporaba standarda pa predlagam, da si ogledate datoteko *pkcs11.c*, programa pkcs11.

4 Ogrodje M.U.S.C.L.E.

MUSCLE - Movement for the Use of Smart Cards in a Linux Environment, je zbirka gonilnikov, knjižnic, programov in API-jev za uporabo pametnih kartic in čitalcev v GNU okolju [5]. MUSCLE je odprtokoden projekt, izdan pod GNU licenco, kar pomeni da lahko izvorno kodo svobodno spreminja vsak, ki si to želi. To je za nas pomembno, saj nekatere knjižnice in programi še zdaleč niso popolni in potrebujejo nekaj sprememb preden nas začnejo "ubogati" [2]. Hkrati pa je izvorna koda zelo koristna, saj nam razkriva notranje delovanje knjižnic in delno opravičuje pomakanje dobre dokumentacije v pisni obliki. Za nas so zanimive predvsem naslednje štiri knjižnice:

MCardApplet je aplet, ki ga naložimo na java kartico ter implementira standard "MUSCLE applet protocol definition" [7]. Deluje podobno kot program *ugani število*, ki smo ga napisali v poglavju 2.1.2, la de je malo bolj zakompliciran.

MCardPlugin implementira drugo stran protokola "MUSCLE applet protocol definition" [7]. Torej MCardPlugin in MCardApplet sta v razmerju nadrejeni-podrejeni, kjer je MCardPlugin nadrejeni in tvori APDU ukaze, MCardApplet pa ukaze izvršuje in pošilja APDU odgovore.

[2] Dober primer je knjižnica libmusclepkcs11, ki ne implementira podpore za EC. Če bi se razvijalci odločili, da izvorne kode ne bi priložili programom, bi bilo ogrodje MUSCLE za to seminarsko povsem neuporabno. Tako pa se je, z nekaj spremembami, izkazalo za povsem koristno.

MuscleCard Library je ovojnica knjižnice MCardPlugin, ki implementira "MUSCLE Framework API" [6].

libmusclepkcs11 pa je ovojnica za knjižnico MuscleCard Library, ki implementira PKCS #11 standard.

Uporaba ogrodja M.U.S.C.L.E.

V tem delu poročila si bomo podrobneje ogledali nekaj praktičnih uporab ogrodja MUSCLE. Videli bomo kako lahko na naši pametni kartici varno hranimo certifikate, kako te certifikate uporabimo v brskalniku Firefox za avtentikacijo in v odjemalcu pošte Thunderbird za podpisovanje elektronskih sporočil. Na koncu pa si bomo po bližje ogledali še program pkcs11, ki sem ga izdelal posebej za to seminarsko nalogo. Program je poučen, saj prikazuje tipično uporabo standarda PKCS #11.

Seveda s tem nismo pokrili vseh praktičnih uporab ogrodja MUSCLE. Med bolj zanimivimi smo izpustili:

- avtentikacija oddaljenemu računalniku s programom OpenSSH
- lokalna avtentikacija na operacijskem sistemu Linux in avtentikacija aplikacij preko mehanizma PAM [11]
- lokalna avtentikacija na operacijskem sistemu Windows
- povezava s popularnim odprtokodnim programom GnuPG

5 Varno hranjenje certifikatov

V najbolj osnovnem primeru lahko pametno kartico uporabljamo kot varno shrambo certifikatov. Ko enkrat naložimo certifikat na kartico smo lahko varni v prepričanju, da zasebni ključ kartice nikoli ne bo zapustil [3]. Shranjeni certifikat lahko nato uporabimo za avtentikacijo s spletnim strežnikom, podpisovanje elektronske pošte, lahko pa ga uporabimo tudi kot par javni/zasebni ključ v programu pkcs11.

Opisan postopek deluje, le če na računalnik namestimo nespremenjeno verzijo ogrodja MUSCLE (pri namestitvi ne izvedemo nobenega ukaza patch). Bolj konkretno: moti ga datoteka musclecardApplet.c knjižnice MCardPlugin, ki vsebuje naslednji kontroverzni if stavek:

```
/* FIX - Forcing Encrypt Mode */
if (cryptInit->cipherDirection == MSC_DIR_SIGN) {
    pBuffer[OFFSET_DATA+currentPointer] = MSC_DIR_ENCRYPT;
}
```

Ta nesrečni del kode spremeni vsako podpisovanje v šifriranje - na kartici se namesto objekta Signature uporabi objekt Cipher. Torej podpis se izvede kot šifriranje z zasebnim ključem.

[3] Seveda pod pogojem, da zaupamo ogrodju MUSCLE in verjamemo da ima EEPROM na kartici ustrezno strojno zaščito, ki preprečuje 'nepooblaščen' dostop do njegove vsebine.

V popravkih sem ta del kode zakomentiral, saj je moj program pkcs11 napisan tako, da izkorišča razliko med objektoma Signature in Cipher. Sploh pa bi bilo podpisovanje z eliptičnimi krivuljami sicer neizvedljivo.

Certifikat lahko na kartico uvozimo na več načinov. Tukaj bom opisal kako to storimo z brskalnikom Firefox. Najprej je potrebno naložiti nov Security Device, v katerega bomo kasneje uvozili certifikat.

Edit / Preferences / Advanced / Encryption / Security Devices / Load

Module Name:	muscle
Module filename:	/usr/local/lib/libmusclepkcs11.so

Edit / Preferences / Advanced / Encryption / View Certificates / Your Certificates
Odpre se dialog za vnos gesla. Če PIN kode na kartici nismo sami ročno spremenili, je prevzeta PIN koda nastavljena na 1234. Za uvoz novega gesla kliknemo gumb Import. Sedaj na disku poiščemo datoteko v kateri je shranjen certifikat (npr. fri-ca.p12). Firefox nas nato vpraša, kam želimo uvoziti izbran certifikat - izberemo MuscleCard Applet. Ponovno je treba vnesti PIN kodo kartice, torej 1234, nato pa še geslo ki je bilo uporabljeno za šifriranje certifikata. Firefox nas na koncu še obvesti ali je vnos certifikata uspel.

5.1 Avtentikacija s spletnim strežnikom

Ko smo enkrat vnesli certifikat, ga lahko uporabimo za avtentikacijo s spletnim strežnikom. Prepričajmo se, da je kartica v čitalcu, nato pa obiščimo stran, ki zahteva avtentikacijo s certifikatom (npr. `https://estudent.fri.uni-lj.si/`). Firefox nas bo še enkrat prosil za geslo, torej vpišemo 1234 in vprašal kateri certifikat naj uporabi. Izberem certifikat, ki smo ga naložili na kartico (tistega, s predpono 'MuscleCard Applet:') in stran bi se sedaj morala uspešno naložiti.

5.1.1 Težave

V starejši verziji brskalnika Firefox (2.*) so se certifikati po ponovnem zagonu brskalnika izgubili. Rešitev je, da certifikat še enkrat naložimo na kartico. To je potrebno storiti le enkrat. Naslednjič ko zaženemo brskalnik bo certifikat ostal na kartici. V novejši verziji brskalnika Firefox (3.*) pa tega problema ni več.

5.2 Podpisovanje elektronske pošte

Shranjen certifikat lahko uporabljamo tudi za podpisovanje elektronske pošte. Opisal bom postopek podpisovanja v odjemalcu pošte Thunderbird (2.0.0.16).

Najprej je potrebno v Thunderbirdu nastaviti, da zaupamo certifikatni agenciji, ki je ključ izdala. Ta korak je obvezen, saj nam sicer Thunderbird ne bo dovolil podpisati sporočila.

Edit / Preferences / Advanced / Certificates / View Certificates / Authorities
V seznamu poiščemo agencijo, ki nam je ključ izdala (npr. Univerza v Ljubljani - @friCA). Pritisnemo gumb Edit in odkljukamo opcijo 'This certificate can identify mail users'.

Edit / Account Settings... / Security
V odseku Digital Signing pritisnemo gumb 'Select...' in izberemo naš certifikat, nato pa odkljukamo še možnost 'Digitally sign messages (by default)'.

Thunderbird bo sedaj vsaki elektronski pošti pripel še digitalni podpis vsebine sporočila.

5.2.1 Težave

Če v seznamu Authorities ne najdete vaše certifikatne agencije, poizkusite ponovno uvoziti certifikat (tokrat iz Thunderbirda).

6 Program pkcs11

Program pkcs11 je bil razvit posebej za to seminarsko nalogo. Omogoča nam, da preko standarda PKCS #11 komuniciramo s katerokoli kartico, ki standard implementira, neodvisno od proizvajalca kartice. Program opravlja dejavnosti kot so: vodenje seje, generiranje ključev na kartici, šifriranje ter podpisovanje in preverjanje podpisov. Program trenutno podpira ključa RSA in EC, vendar pa je dodajanje novih trivialno. Program je napisen v prenosljivem ANSI C, kar pomeni, da bi se moral prevesti na vseh večjih operacijskih sistemih.

6.1 Generiranje ključev

Začnimo na začetku. Preden lahko uporabimo katerokoli kriptografsko operacijo, moramo na kartici najprej ustvariti par javni/zasebni ključ. To lahko storimo na dva načina: uporabimo ključ uvoženega certifikata ali pa s programom pkcs11 ustvarimo par ključev na kartici sami. Kako uvozimo certifikat smo že spoznali, sedaj pa si poglejmo, kako par ključev generiramo na kartici. Prednost tega pristopa je očitna: ker je zasebni ključ ustvarjen na kartici je fizično nikoli ne zapusti.

Najprej si poglejmo kako ustvarimo par ključev za RSA.

```
$ ./pkcs11 genkeypair rsa
PIN: 1234
```

Generiranje EC ključev je zelo podobno:

```
$ ./pkcs11 genkeypair ec
PIN: 1234
```

Program pkcs11 nas opozori, če na kartici že obstaja ključ določenega tipa. V tem primeru se nov ključ ne generira. Če na kartico uspemo spraviti dva ključa istega tipa (npr. enega generiramo s programom pkcs11, drugega pa uvozimo iz certifikata), bo pri šifriranju in podpisovanju uporabljen ključ, ki je bil generiran prej[4]. Da se izognemo zmešnjavi, je zato dobra ideja, da imamo na kartici vedno le en ključ istega tipa (lahko pa imamo seveda brez težav na kartici istočasno ključ RSA in EC).

Dolžine generiranih ključev so zapečene v programu. Za RSA se generira ključ dolžine 1024 bitov, za EC pa 192 bitov.

[4] V resnic bo uporabljen prvi ključ, ki ga vrne PKCS #11 funkcija `C_FindObjects`, to pa je v trenutni implementaciji tisti ključ, ki je bil prvi ustvarjen.

6.2 Šifriranje

S programom pkcs11 lahko šifriramo manjša števila[5]. Vhod je vedno število v šestnajstiškem zapisu. Šifriramo lahko za enkrat le z algoritmom RSA. Podatke lahko zašifriramo z javnim ali pa z zasebnim ključem. Poglejmo si primer, kjer z zasebnim ključem zašifriramo število 0x123456789abcdef v šestnajstiškem zapisu, rezultat pa shranimo v datoteko *sig*.

```
./pkcs11 encrypt private rsa > sig
PIN: 1234
123456789abcdef
```

Podatke lahko odšifriramo z naslednjim ukazom:

```
$ ./pkcs11 decrypt public rsa < sig
PIN: 1234
```

Na zaslonu se izpiše `0123456789abcdef`, torej število, ki smo ga šifrirali. Opazimo pa, da je v izpisu na začetku znak 0, ki ga v začetnem nizu ni bilo. To seveda ni napaka! Ne pozabimo, da šifriramo število 0x123456789abcdef in ne ASCII zapisa števila 123456789abcdef. Razlog zakaj se na začetku izhoda znajde znak 0 je, da je program pkcs11 nastavljen tako, da vsak bajt izpiše z dvema znakoma.

Napravimo sedaj preprost preizkus in se prepričajmo, da program pkcs11 res uporablja pravilen RSA algoritem. Na pomoč nam bo priskočil skriptni jezik Python. Programu pkcs11 je priložen tudi testni program `test`, ki med drugim izpiše atribute javnega RSA ključa (modul in eksponent).

```
$ ./test | grep CKO_PUBLIC_KEY,CKK_RSA
- CKO_PUBLIC_KEY,CKK_RSA[0],id=0x4b4559303030303030303030303030303030303031,
e=0x010001,n=0x9c8f531ae35a8569ee1243d8abf617b2601ccd554917f81000452349b7a
7b1511a480440f7c2929af d3b0a90d6548ddda317ec526f2a5b83e02e754e4705856c07e8
74260f7ac8798a72e9ec6df7923761bb84 d6e94bf65e6e6c287d760c3c06b18d6f5de0eec
03a0bf2de389093f67d3fc05c8fa140882e56d37aaa6766c72d
$ cat sig
41e80d75781de6f12fd829d60780b3b4a1e070e685aba113b0bf61e6aed6a84a916403aa04
cdf678df97a259514a97f667030b8de436e6a52bd5703496a99736d3b08da96f8973a53a41
dd82e0c9251eddf9a11c0b78397e34ad38b947f6b2f248fcf62061a0bf5705a98d806aede7
e7c8699eebde51d045154f4ed48a5f2e64
```

Sedaj v drugem terminalu zaženemo interpreter jezika Python in vtipkamo:

```
$ python
>>> e=0x010001
>>> n=0x9c8f531ae35a8569ee1243d8abf617b2601ccd554917f81000452349b7a7b1511a
480440f7c2929afd3b0a90d6548ddda317ec526f2a5b83e02e754e4705856c07e874260f7a
c8798a72e9ec6df7923761bb84d6e94bf65e6e6c287d760c3c06b18d6f5de0eec03a0bf2de
389093f67d3fc05c8fa140882e56d37aaa6766c72d
>>> c=0x41e80d75781de6f12fd829d60780b3b4a1e070e685aba113b0bf61e6aed6a84a91
6403aa04cdf678df97a259514a97f667030b8de436e6a52bd5703496a99736d3b08da96f89
```

[5]Dolžina števila v bitih mora biti manjša od dolžine ključa minus 3 bajte za podložitev.

```
73a53a41dd82e0c9251eddf9a11c0b78397e34ad38b947f6b2f248fcf62061a0bf5705a98d
806aede7e7c8699eebde51d045154f4ed48a5f2e64
>>> '%x' % pow(c, e, n)
'1ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffff000123456789abcdef'
```

Zadnji ukaz nas je malo presenetil. Pričakovali smo le niz '0123456789abcdef'. Zgodi se naslednje: ko smo poklicali funkcijo C_Encrypt (z ukazom ./pkcs11 encrypt private rsa) iz datoteke p11_crypt.c knjižnice libmusclepkcs11, je ta naš vhod podložila in ga tako spremenjenega posredovala nižjim plastem. Procedura padRSAType1 podloži naše vhodno število tako, da najprej doda bajta 0x00 in 0x01, nato doda nekaj bajtov 0xff, potem še bajt 0x00, nato pa na konec prilepi naše število. Funkcija doda toliko bajtov 0xff, da je skupna dolžina izhoda enaka velikosti ključa.

Seveda lahko števila šifriramo tudi z zasebnim, odšifriramo pa z javnim ključem.

```
$ ./pkcs11 encrypt public rsa > cipher
PIN: 1234
123456789abcdef
$ ./pkcs11 decrypt private rsa < cipher
PIN: 1234
```

Na zaslonu se po pričakovanju izpiše niz '0123456789abcdef'.

Ponovno lahko naredimo kratek preizkus. Tokrat bomo v Pythonu zašifrirali število 123456789abcdef. Izpis se mora ujemati z datoteko cipher.

```
>>> e=0x...
>>> n=0x...
>>> p=0x1ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffff000123456789abcdef
>>> open('cipher_python', 'w').write('%x\n' % pow(p, e, n))
>>> exit()
$ diff cipher cipher_python
```

Vidimo, da sta datoteki cipher in cipher_python identični.

6.3 Podpisovanje

S programom pkcs11 lahko tudi podpisujemo. Podpisovanje je možno tako s ključem RSA kot tudi z EC. Pri podpisovanju z algoritmom RSA se najprej izračuna 16 bajtna MD5 zgostitev vhoda, nato se zgostitev podloži na isti način kot pri šifriranju (vendar s to razliko, da se pri podpisovanju to stori na kartici), rezultat se na koncu zašifrira s privatnim RSA ključem. Pri EC pa se najprej izračuna 20 bajtna SHA zgostitev, nato pa se uporabi algoritem ECDSA.

JavaCard API je očitno zasnovan tako, da se zgostitev vhoda izvede na kartici namesto na uporabnikovem računalniku. To pa je potratno, saj bi bilo hitreje, če bi zgostitev izvedel

kar uporabnik, na kartici pa bi se izvedel samo podpis. Poleg tega, pa je težava tudi v tem, da MUSCLE še ne podpira podpisovanja večjih datotek (možno je le podpisovanje števil, ki so približno tako dolge kot ključ). Zaradi teh dveh razlogov podpisujemo dokumente s programom pkcs11 na sledeč način: najprej dokument zgostimo (npr. s programom md5sum), nato ga pošljemo kartici, kjer se ponovno zgosti, nato pa še podpiše.

Sedaj pa si na primeru oglejmo kako podpisovanje deluje v praksi. Z algoritmom RSA bomo podpisali datoteko s preprostim sporočilom.

```
$ cat << EOF > msg
> podpisi me!
> EOF
$ md5sum msg | cut -d' ' -f1 > msg.md5
$ ./pkcs11 sign rsa < msg.md5 > sig
PIN: 1234
```

Podpis lahko preverimo, če pokličemo program pkcs11 z argumentom verify. Program tokrat pričakuje dve števili v šestnajstiškem zapisu: prvo je sporočilo, drugo pa njen podpis.

```
$ cat msg.md5 sig | ./pkcs11 verify rsa
PIN: 1234
```

Če se na terminal izpiše 'valid' pomeni, da se podpis ujema s sporočilom, sicer se izpiše 'INVALID'.

Tudi tokrat bomo s pomočjo Python-a preverili ali podpis res deluje tako, kot je treba.

```
>>> e=0x...
>>> n=0x...
>>> sig=0x210dcbec090d816313db59bf10d039993ee466507c125893723dcc580218d9ec8
ebd891c0ece26608764542b80f27d16ff49d3085e08e58158d2f3526e58eb7117d610da4f1e
a2c826141f0ba364e5c867e2501d936d0f53db523b3b1851b2848030dd3e36cd62592101e2b
0c98907b826ef6681574a48723264524face35c4c
>>> test = '%x' % pow(sig, e, n)
>>> test
'1ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffff003020300c06082a864886f70d020505000410224
9a36bfc9de3913f11a98a20725f1a'
```

Za trenutek se ustavimo in si oglejmo, kaj smo dobili. Rezultat je tako kot pri šifriranju podložen, zato lahko prvi del ignoriramo. Brez da bi šli preglobokо v podrobnosti, povejmo samo to, da je zadnji del sporočila DER zakodirana struktura in da je zadnjih 32 znakov MD5 zgostitev, ki nas zanima.

Sedaj moramo le še preveriti, da se MD5 zgostitev števila v datoteki msg.md5 ujema z zadnjimi 32 znaki spremenljivke *test* (spomnimo se, da se pri podpisovanju zgostitev opravi dvakrat). Ker pa nas zanima MD5 zgostitev števila v datoteki msg.md5, ne pa njene šestnajstiške ASCII predstavitve, program md5sum ne bo zadostoval - tudi tokrat na pomoč priskoči Python.

```
>>> import md5, binascii
>>> msg_md5 = binascii.unhexlify(open('msg.md5').readline().strip())
```

```
>>> hash = md5.md5()
>>> hash.update(msg_md5)
>>> test[-32:] == hash.hexdigest()
True
```

Rezultata se ujemata! Test je torej uspel.

Podpisovanje z EC je zelo podobno podpisovanju z RSA. Zato podajmo samo kratek primer:

```
$ ./pkcs11 sign ec < msg.md5 > sig
PIN: 1234
$ cat msg.md5 sig | ./pkcs11 verify ec
PIN: 1234
```

6.4 Ogrodje M.U.S.C.L.E. - podrobnosti

Sedaj, ko smo spoznali kako deluje program pkcs11, lahko na primeru natančno opišemo kako so med seboj povezane knjižnice ogrodja MUSCLE. Pogledali si bomo kaj vse se zgodi, ko podpišemo določeno sporočilo. Za podpisovanje bomo uporabili kar program pkcs11, vendar pa se podobno zaporedje akcij zgodi tudi pri podpisovanju s programoma Firefox in Thunderbird. Osredotočili se bomo le na podpisovanje, zato bomo nekatere akcije (PIN, izbor ključa, branje ključa, ...) v primeru izpustili.

Recimo, da želimo podpisati šestnajstiško število `0xcafe`. Ko v terminal vtipkamo ukaz:

```
$ ./pkcs11 encrypt private rsa
PIN:
cafe
```

se pokliče zaporedje procedur, ki ga prikazuje tabela 2. Zaporedje se prične z visokonivijskimi metodami, kot sta encrypt in C_Encrypt, nadaljuje pa z vedno bolj nizkonivojskimi metodami knjižnic MCardPlugin in PCSC-Lite. Ukazi se nato preko žice prenesejo na kartico, kjer se šifriranje tudi izvrši. Tabela ni popolna - manjkajo procedure ki se kličejo znotraj knjižnice CCID.

Poglejmo si tudi, kako izgleda APDU dialog med knjižnico MCardPlugin in apletom MCardA-plet:

```
// Cipher init
=> B0 36 00 01 05 00 03 01 00 00
<= 90 00
// Cipher final
=> B0 36 00 03 83 01 00 80 00 01 FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF 00 CA FE
<= 00 80 3F 37 B4 5D 9F 06 2B F8 4F E2 CD 2F 31 AA B4 D9 70 13 3C 1B 0B 5C
B6 20 1F 5F E0 E5 5E B5 EC F5 0D 66 A4 73 43 CB FD F7 B1 19 5A 3C 42 F4 12
7F A4 4C 80 91 8F E5 F6 4B 4B DF 01 18 A5 05 23 76 01 64 A7 F5 13 E7 0C 36
```

procedura	datoteka	knjižnica/program
encrypt	pkcs11.c	pkcs11
C_Encrypt	p11_crypt.c	libmusclepkcs11
msc_ComputeCrypt	p11x_msc.c	
MSCComputeCrypt	musclecard.c	MuscleCard Library
PL_MSCComputeCrypt	musclecardApplet.c	MCardPlugin
SCardExchangeAPDU		
SCardTransmit	winscard.c	PCSC-Lite
IFDTransmit	ifdwrapper.c	
IFDHTransmitToICC	ifdhandler.c	CCID
prenos po žici		
process	CardEdge.java	MCardApplet
ComputeCrypt		
javacardx.crypto.Cipher.doFinal	?	Java Card

↓

Tabela 2: Zaporedje klicev procedur pri podpisovanju.

```
A6 B7 38 A8 51 8A F6 45 9E 90 51 BD B0 D6 1E 65 0B 18 50 06 C2 CB 4C 47 F9
62 4A 3C 28 69 2F C1 C3 8E 17 C2 5E DB 7A 3B 54 02 57 D4 A2 06 E0 38 F1 14
B8 C4 D2 50 DA 26 90 00
```

Namestitev

Sledijo navodila za namestitev ustreznih programov. Opisan postopek deluje na distribuciji Arch Linux 2008.06, sicer pa so navodila neodvisna od posamezne distribucije, zato bi brez večjih sprememb morala delovati na katerikoli distribuciji. Na ostalih operacijskih sistemih postopka nisem testiral. Vse datoteke, ki jih pri namestitvi potrebujemo, lahko dobite na `ftp://213.250.22.254/pub/pkcs11/`.

7 Java Card Development Kit

Če vaša pametna kartica podpira Java Card 2.2, je dovolj če namestite le verzijo 2.2. Sicer je potrebno namestiti obe verziji.

7.1 JCDK 2.1.1

domača stran: `http://java.sun.com/javacard`
verzija: 2.1.1

JCDK 2.1.1 lahko namestimo kamorkoli. V opisu, ki sledi bomo namestili JCDK 2.1.1 v uporabnikov domač direktorij (globalna spremenljivka HOME).

7.1.1 Namestitev

```
$ tar xzf java_card_kit-2_1_1-unix.tar.Z -C ~
$ echo "export JC21BIN=$HOME/jc211/bin" >> ~/.bashrc
$ echo "export PATH=\$PATH:\$JC21BIN" >> ~/.bashrc
$ . ~/.bashrc
```

7.2 JCDK 2.2.2

domača stran: `http://java.sun.com/javacard`
verzija: 2.2.2

JCDK 2.2.2 potrebujemo, da prevedemo MCardApplet.

7.2.1 Namestitev

```
$ unzip -x java_card_kit-2_2_2-linux.zip -d ~/jc222
$ cd ~/jc222/java_card_kit-2_2_2
$ unzip -x java_card_kit-2_2_2-rr-bin-linux-do.zip -d ..
$ echo "export JC_HOME=$HOME/jc222" >> ~/.bashrc
$ . ~/.bashrc
```

8 ogrodje M.U.S.C.L.E.

Nekaj programov, ki jih pri namestitvi ogrodja MUSCLE potrebujemo [6]:

- gcc (4.3.1)
- java (1.6.0_07)
- pkg-config (0.23)
- GNU make (3.81)
- GNU tar (1.20)
- sudo (1.6.9p17)
- libusb (0.1.12)
- sed (4.1.5)
- perl (5.10.0)
- patch (2.5.9)
- readline (5.2)

Seveda potrebujemo tudi programe ogrodja MUSCLE:

- PCSC-Lite (1.4.102)
- CCID (1.3.8)
- MuscleCard Library (1.3.3)
- Muscle Framework (1.1.6)
- MuscleTool (2.1.0)

[6] V oklepajih so navedene verzije programov, ki sem jih uporabljal sam.

Potrebujemo pa tudi knjižnico Global Platform, s pomočjo katere bomo na pametno kartico naložili javin aplet:

- Global Platform (5.0.0)
- GPShell (1.4.2)

V nadaljevanju sledi opis namestitve programov ogrodja MUSCLE in knjižnice Global Platform. Za vsak program, pa bomo podali tudi kratek opis njegovih glavnih nalog.

8.1 PCSC-Lite

domača stran: `http://pcsclite.alioth.debian.org`
verzija: 1.4.102

PCSC-Lite je implementacija PC/SC standarda [1]. PC/SC definira nizkonivojski API za dostop do pametne kartice, ki je neodvisen od proizvajalca pametne kartice. Standard definira tudi upravitelja virov, ki omogoča večim procesom sočasen dostop do pametnih kartic, priključenih v sistem.

8.1.1 Namestitev

```
$ tar xjf pcsc-lite-1.4.102.tar.bz2
$ cd pcsc-lite-1.4.102
$ ./configure --disable-libhal
...
PC/SC lite has been configured with following options:

Version:             1.4.102
System binaries:     /usr/local/sbin
Configuration files: /usr/local/etc


Host:                i686-pc-linux-gnu
Compiler:            gcc
Preprocessor flags:  -I${top_srcdir}/src
Compiler flags:      -Wall -fno-common -g -O2
Preprocessor flags:  -I${top_srcdir}/src
Linker flags:
Libraries:

PTHREAD_CFLAGS:      -pthread
PTHREAD_LIBS:
PCSC_ARCH:           Linux

libhal support:       no
libusb support:       yes
SCF reader support:   false
USB drop directory:   /usr/local/pcsc/drivers
ATR parsing messages: false
confdir:              /etc
```

```
ipcdir:                /var/run/pcscd
...

$ make
$ sudo make install
```

8.2 CCID

domača stran: `http://pcsclite.alioth.debian.org`
verzija: 1.3.8

CCID je implementacija standardov CCID [2] in ICCD [3], ki definirata protokole med gostiteljevim USB priključkom in napravami. CCID podpira veliko čitalcev pametnih kartic in ga lahko uporabimo, kot gonilnik za PCSC-Lite.

8.2.1 Namestitev

```
$ tar xjf ccid-1.3.8.tar.bz2
$ cd ccid-1.3.8
$ ./configure --enable-udev
...
libccid has been configured with following options:

Version:             1.3.8
User binaries:       NONE/bin
Configuration files: NONE/etc


Host:                i686-pc-linux-gnu
Compiler:            gcc
Preprocessor flags:
Compiler flags:      -g -O2
Preprocessor flags:
Linker flags:
Libraries:

PTHREAD_CFLAGS:      -pthread
PTHREAD_LIBS:
BUNDLE_HOST:         Linux
DYN_LIB_EXT:         so
LIBUSB_CFLAGS:
LIBUSB_LIBS:         -lusb
SYMBOL_VISIBILITY:   -fvisibility=hidden

libusb support:          yes
use USB interrupt:       no
multi threading:         yes
bundle directory name:   ifd-ccid.bundle
USB drop directory:      /usr/local/pcsc/drivers
serial Twin support:     no
serial twin install dir: /usr/local/pcsc/drivers/serial
```

```
compiled for pcsc-lite:  yes
udev support:            yes
...
$ make
$ sudo make install
$ sudo cp src/pcscd_ccid.rules /etc/udev/rules.d/
```

8.2.2 Težave

Če `./configure` vrne

```
configure: error: ifdhandler.h not found, install pcsc-lite 1.3.3
or later, or use ./configure PCSC_CFLAGS=...
```

potem preveri, če je nameščen program `pkg-config`. Ker PCSC-Lite namesti datoteke `.pc` na nestandardno lokacijo, je potrebno nastaviti globalno spremenljivko `PKG_CONFIG_PATH`. To lahko storimo z naslednjima ukazoma:

```
$ echo "export PKG_CONFIG_PATH=/usr/local/lib/pkgconfig" >> ~/.bashrc
$ . ~/.bashrc
```

8.2.3 Test

Preverimo, če je bila namestitev PCSC-Lite in CCID uspešna. Najprej je treba zagnati daemon pcscd.

```
$ sudo pcscd
```

Če operacijski sistem programa ne najde, lahko poskusimo spremeniti globalno spremenljivko PATH:

```
$ echo "export PATH=$PATH:/usr/local/sbin" >> ~/.bashrc
$ . ~/.bashrc
```

Programu PCSC-Lite je priložen testni program:

```
$ pcsc-lite-1.4.102/src/testpcsc
```

Iz izpisa je jasno ali je bila namestitev uspešna.

8.3 MuscleCard Library

domača stran: `http://pcsclite.alioth.debian.org`
verzija: 1.3.3

MuscleCard Library implementira musclecard framework API [6]. Musclecard framwork je višjenivoski API za dostop do pametne kartice. API definira funkcije kot npr. `MSCGenerateKeys` za generiranje ključev, `MSCComputeCrypt` za izračun kriptografskih operacij in `MSCCreateObject` za ustvarjanje novih objektov na kartici.

8.3.1 Patch

V izvorno kodo sem dodal nekaj konstant, ki so potrebne za uporabo eliptičnih krivulj.

8.3.2 Namestitev

```
$ tar xzf libmusclecard-1.3.3.tar.gz
$ cd libmusclecard-1.3.3
$ patch -p0 < ../patches/libmusclecard.patch
$ ./configure
$ make
$ sudo make install
```

8.4 Muscle Framework

domača stran: `http://muscleplugins.alioth.debian.org`
verzija: 1.1.6

V tem paketu najdemo programa MCardPlugin in libmusclepkcs11.

8.5 MCardPlugin

Program MCardPlugin deluje v paru z apletom MCardApplet, ki ga bomo na pametno kartico naložili kasneje. Naloga MCardPlugin je, da klice procedur višjenivojskega MuscleCard framework API-ja spremeni v APDU-je in jih preko knjižnice PCSC-Lite posreduje svojemu paru MCardApplet, ki teče na kartici. Protokol je definiran v [7].

Primer Klic procedure `MSCComputeCrypt`, ki je definirana v MuscleCard frameworku, pokliče proceduro `PL_MSCComputeCrypt` iz MCardPlugina. Procedura nato zgradi dva APDU-ja in jih posreduje proceduri `SCardExchangeAPDU` iz PCSC-Lite, ki jih nato (prek CCID) pošlje pametni kartici.

8.5.1 Patch

Dodal sem kodo, s katero je možno preveriti digitalni podpis.

8.5.2 Namestitev

```
$ tar xzf muscleframework-1.1.6.tar.gz
$ cd muscleframework-1.1.6/MCardPlugin/
$ patch -p0 < ../../patches/MCardPlugin.patch
$ ./configure
$ make
$ sudo make install
```

Po namestitvi je treba v datoteko
`/usr/local/pcsc/services/mscMuscleCard.bundle/Contents/Info.plist`

zapisati še ATR pametne kartice, ki jo imamo namen uporabljati. ATR kartice lahko izvemo na več načiov. Uporabimo lahko program testpcsc:

```
$ ~/build/pcsc-lite-1.4.102/src/testpcsc < /dev/null | grep "ATR Value"
```

8.6 libmusclepkcs11

Knjižnica libmusclepkcs11 implementira standard PKCS #11 [8]. V resnici ni libmusclepkcs11 nič drugega, kot ovojnica knjižnice MuscleCard framework.

8.6.1 Patch

V izvorni kodi knjižnice sem:

- odpravil par hroščev
- implementiral funkcije za preverjanje podpisov
- popravil funkcijo za podpisovanje
- dodal podporo za generiranje EC ključev
- dodal podporo za podpisovanje in preverjanje podpisov z eliptičnimi krivuljami

8.6.2 Namestitev

```
$ cd ../libmusclepkcs11
$ patch -p0 < ../../patches/libmusclepkcs11.patch
$ ./configure
$ make
$ sudo make install
```

8.7 MuscleTool

domača stran: `http://muscleapps.alioth.debian.org`
verzija: 2.1.0

MuscleTool je CLI program, ki ga uporabljamo za komuniciranje s pametno kartico iz ukazne vrstice. Uporabljamo ga tudi za inicializacijo in prikrojitev MCardApplet apleta.

8.7.1 Namestitev

```
$ tar xzf muscletool-2.1.0.tar.gz
$ cd muscletool-2.1.0
$ ./configure --enable-readline
$ make
$ sudo make install
```

8.7.2 Test

Test bo pokazal ali je bila namestitev ogrodja Muscle Framework uspešna. Najprej preverimo če je `/usr/local/bin` v spremenljivki PATH. Če ni, ga je potrebno dodati, sicer vtipkajmo ukaz:

```
$ muscleTool
MuscleCard shell - type "help" for help.
muscleTool > tokens
   1.    MuscleCard Applet

ListTokens Success.
```

Če nam program javi

```
muscleTool: error while loading shared
libraries: libmusclecard.so.1: cannot open shared object file: No such
file or directory
```

pomeni, da nalagalnik ni našel knjižnice libmusclecard.so. Najbrž zato, ker se knjižnica nahaja v `/usr/local/lib`, ne pa v `/usr/lib`. Poskusimo z naslednjima ukazoma:

```
$ sudo sh -c "echo /usr/local/lib >> /etc/ld.so.conf"
$ sudo ldconfig
```

8.8 Global Platform

domača stran: `http://sourceforge.net/projects/globalplatform`
verzija: 5.0.0

Global Platform in GPShell potrebujemo za nalaganje java apletov na pametno kartico.

8.8.1 Namestitev

```
$ ./configure
$ make
$ sudo make install
```

8.9 GPShell

domača stran: `http://sourceforge.net/projects/globalplatform`
verzija: 1.4.2

GPShell je CLI ovojnica za knjižnico Global Platform.

8.9.1 Namestitev

```
$ ./configure
$ make
$ sudo make install
```

8.10 MCardApplet

domača stran: `http://muscleapps.alioth.debian.org`
verzija: 0.9.11

Opisal bom postopek namestitve apleta na pametno kartico Cyberflex Access 32K e-gate. Opisan postopek lahko uporabimo tudi za kartice ostalih proizvajalcev. Ker pa Cyberflex kartica ne podpira Java Card standarda 2.2 je namestitev malo bolj zapletena. Če imamo npr. kartico JCOP je najbrž enostavneje, če za namestitev uporabimo Eclipse in JCOP tools [10].

Kot sem že omenil, kartica Cyberflex ne podpira Java Card 2.2, ki pa je nujno potreben za prevajanje MCardAppleta. Uporabimo naslednji trik[7]: MCardApplet prevedemo v okolju Java Card 2.2, converter pa uporabimo iz JavaCard 2.1.

8.10.1 Patch

V izvorni kodi sem spremenil način podpisovanja in dodal podporo za eliptične krivulje [8].

8.10.2 Namestitev

```
$ tar xzf MCardApplet-0.9.11.tar.gz
$ cd MCardApplet-0.9.11
$ cp ../patches/CardEdge.java src/com/musclecard/CardEdge/

$ cd ~/jc222/samples
$ cp -r ../api_export_files/ classes
$ rm -rf src/com/musclecard
$ cp -rf ~/build/MCardApplet-0.9.11/src/com src

$ rm -rf ~/jc211/samples/classes
$ mkdir -p ~/jc211/samples/classes
$ javac -g -source 1.3 -target 1.1 -classpath classes/:../lib/api.jar \
src/com/musclecard/CardEdge/*.java -d ~/jc211/samples/classes/

$ cd ~/jc211/samples/
$ cat << EOF > CardEdge.opt
-out EXP JCA CAP
-classdir classes
-exportpath ../api21
-applet 0xa0:0x0:0x0:0x0:0x0:0x1:0x1 com.musclecard.CardEdge.CardEdge
com.musclecard.CardEdge
0xa0:0x0:0x0:0x0:0x0:0x1 1.0
EOF
$ converter -config CardEdge.opt
```

Ker Cyberflex kartice ne poznajo CAP zapisa, so potrebni nasledi ukazi, ki datoteko CardEdge.cap spremenijo v out.bin, ki jo lahko neposredno naložimo na kartico.

[7] Ne razumem najbolje, zakaj opisana metoda deluje. Za enkrat zgleda, da je vse v redu, vendar pa apleta še nisem temeljito testiral.

[8] Ker mi je Eclipse, s svojimi zmožnostmi formatiranja kode spremenil prav vsako vrstico datoteke CardEdge.java, je bilo nemogoče narediti spodoben diff. Zato spremembe, ki sem jih naredil, ne prilagam v obliki patcha, ampak je treba datoteko CardEdge.java v celoti prepisati.

```
$ cd classes/com/musclecard/CardEdge/javacard/
$ jar xvf CardEdge.cap
$ cd com/musclecard/CardEdge/javacard/
$ cat Header.cap Directory.cap Import.cap Applet.cap Class.cap Method.cap \
StaticField.cap ConstantPool.cap RefLocation.cap Descriptor.cap > out.bin
```

Naslednji ukaz je odvisen od kartice na katero nalagamo aplet.

- Cyberflex kartice

  ```
  $ gpshell << EOF
  enable_trace
  establish_context
  card_connect
  select -AID a0000000030000
  open_sc -security 1 -keyind 0 -keyver 0 \
  -mac_key 404142434445464748494a4b4c4d4e4f \
  -enc_key 404142434445464748494a4b4c4d4e4f
  delete -AID A0000003230101
  delete -AID A00000032301
  delete -AID A00000000101
  delete -AID A000000001
  install_for_load -pkgAID A000000001 -nvCodeLimit 13000
  load -file out.bin
  install_for_install -instParam 00 -priv 02 -AID A00000000101 \
  -pkgAID A000000001 -instAID A00000000101 -nvDataLimit 13000
  card_disconnect
  release_context
  EOF
  ```

- JCOP kartice

  ```
  $ gpshell << EOF
  mode_201
  enable_trace
  establish_context
  card_connect
  select -AID a000000003000000
  open_sc -security 0 -keyind 0 -keyver 0 \
  -mac_key 404142434445464748494a4b4c4d4e4f \
  -enc_key 404142434445464748494a4b4c4d4e4f
  delete -AID A0000003230101
  delete -AID A00000032301
  delete -AID A00000000101
  delete -AID A000000001
  install -file out.bin -priv 2
  card_disconnect
  release_context
  EOF
  ```

Ukaz najprej izbere aplet Card Manager z AID vrednostjo 0xa000000003000000 in se mu tudi avtenticira. Nato izbriše staro verzijo apleta MCardApplet in naloži novo. MCardApplet je

sedaj naložen na kartico. Vse kar nam še preostane je inicializacija apleta. To opravilo zaupamo programu muscleTool.

```
$ muscleTool << EOF
tokens
format 1
1
1234
1234
100
1234
100
10000
0x0002
0x0002
0x0001
1
exit
EOF
```

Ukaz alocira 10.000B spomina na kartici, nastivi PIN [9] in določi kdo sme uporabljati javne oz. zasebne ključe.

9 Aplikacija pkcs11

Program pkcs11 je treba samo prevesti. To storimo z ukazom:

```
$ make
```

9.1 Test

Recimo, da želimo podpisati izvorno kodo programa pkcs11 - datoteko *pkcs11.c* in nato podpis preveriti. To lahko storimo z naslednjimi ukazi:

```
$ ./pkcs11 genkeypair rsa
PIN: 1234
$ md5sum pkcs11.c | cut -f1 -d' ' > msg
$ ./pkcs11 sign rsa < msg > sig
PIN: 1234
$ cat msg sig | ./pkcs11 verify rsa
PIN: 1234
```

Če je program pkcs11 izpisal *valid* pomeni, da se sporočilo ujema s podpisom. Poskusimo preveriti še namenoma pokvarjen podpis:

[9] PIN je prevzeto nastavljen na 1234 in ga lahko spremenimo tudi po inicializaciji kartice s programom muscleTool

```
$ tr 0-9a-f 1-9a-f0 < sig > sig.corrupt
$ cat msg sig.corrupt | ./pkcs11 verify rsa
PIN: 1234
```

Program nas obvesti, da podpis ne ustreza sporočilu.

10 Zaključek

V tej seminarski nalogi smo se spoznali z java karticami in standardom PKCS #11, ki ga implementira ogrodje MUSCLE. S programom pkcs11 smo se naučili kako standard tudi v praksi uporabimo. Ker je področje, ki sem ga v tej seminarski nalogi raziskoval precej veliko, ostaja še veliko dela, ki ga bo potrebno storiti v prihodnosti. Nekaj takih tem je naštetih v naslednjem odseku.

10.1 Nadaljnje delo

- V tem poročilu smo si podrobneje ogledal le ogrodje MUSCLE. To seveda ni edino ogrodje, ki implementira PKCS #11 standard. Zanimivo ogrodje je tudi OpenSC [4], ki je prav tako kot MUSCLE zastonj in odprtokodno, kar ga naredi primernega za študij. Razlog, zakaj si ogrodja OpenSC nismo podrobneje ogledali je preprost: OpenSC za enkrat podpira le kartice z datotečnim sistemom, kar pa pomeni, da večina java kartic ni podprtih - vključno s karticama, ki sem ju imel na voljo za testiranje.
- Eden izmed ciljev te seminarske naloge je bil izdelati program, ki preko PKCS #11 API-ja podpisuje dokumente s kriptografskimi algoritmi eliptičnih krivulj. Ker pa ogrodje MUSCLE ne implementira ECC, sem moral ustrezno podporo vgraditi sam. To pa prinaša dve posledici:
  - Spremembe, ki sem jih napravil je potrebno temeljito preveriti, saj je možno, da še vedno vsebujejo kakšne napake.
  - Pri implementiranju podpore za ECC, sem zaradi preprostosti nekatere dele standarda PKCS #11 izpustil. Tako, na primer, ni možno prebrati parametre javnega EC ključa. Torej je v prihodnje potrebno implementacijo dopolniti.
- Ogrodje MUSCLE sem testiral samo v Linux okolju. Gotovo bi bilo koristno uporabljati MUSCLE tudi na ostalih operacijskih sistemih, npr. Microsoft Windows, BSD, ... Sicer pa sem se potrudil in svojo aplikacijo pkcs11 napisal v prenosljivem ANSI C jeziku, kar pomeni, da bi se morala prevesti na vseh večjih operacijskih sistemih.

Literatura

[1] *PC/SC standard*, `http://www.pcscworkgroup.com/specifications/overview.php`

[2] *CCID standard*, `http://www.usb.org/developers/devclass_docs/DWG_Smart-Card_CCID_Rev110.pdf`

[3] *ICCD standard*, `http://www.usb.org/developers/devclass_docs/DWG_Smart-Card_USB-ICC_ICCD_rev10.pdf`

[4] *OpenSC*, http://www.opensc-project.org/opensc

[5] *M.U.S.C.L.E* http://www.linuxnet.com/index.html

[6] *MUSCLE Framework API*, http://musclecard.com/musclecard/files/muscle-api-1.3.0.pdf

[7] *MUSCLE applet protocol definition*, http://musclecard.com/musclecard/files/mcardprot-1.2.1.pdf

[8] *PKCS #11*, http://www.rsa.com/rsalabs/node.asp?id=2133

[9] *Java Card Platform Specification* http://java.sun.com/javacard/

[10] *Musclecard with IBM JCOP Smartcard Howto*, http://www.liquid-reality.de/main/projects/musclecard_jcop_howto

[11] *Pluggable Authentication Modules* http://www.opengroup.org/rfc/rfc86.0.html

[12] *An Introduction to Java Card Technology - Part 1* http://java.sun.com/javacard/reference/techart/javacard1/

[13] *An Introduction to Java Card Technology - Part 2* http://java.sun.com/javacard/reference/techart/javacard2/

[14] *An Introduction to Java Card Technology - Part 3* http://java.sun.com/javacard/reference/techart/javacard3/