JavaCard cryptography

General hints:

- **Use existing algorithms/modes rather than write your own** - Algorithms in JavaCard are much slower and most probably less secure against power analysis than the native functions provided by JavaCard library.
- **Store session data in RAM –** operation in RAM are much faster and more secure against power analysis. Moreover, EEPROM has limited number of rewrites before becomes unreliable (10^5 - 10^6 writes).
- **Do NOT store keys and PINs in primitive arrays –** Use specialized objects like OwnerPIN and Key for storage. These are better protected against power ad fault attacks.
- **Erase unused keys and arrays with sensitive values –** Use specialized method if exists (Key::clearKey()) or overwrite with random data.
- **Use transactions to ensure atomic operations –** Short parts of code that must be executed together should be protected by the transaction. Otherwise, power supply can be interrupted inside code and inconsistency may occur. Be aware of attacks based on interrupted transactions so called Rollback attack.
- **Do not use conditional jumps with sensitive data** – Branching after condition is recognizable with power analysis. E.g., branch THEN increase offset for next instruction only by 1, but branch ELSE must compute new offset dependent on length of THEN code. This addition takes much longer time and is recognizable using power analysis.
- **Allocate all necessary resources in constructor –** Applet installation is usually performed in trusted environment. Will prevent attacks based on limiting resources necessary for applet and thus introducing inconsistency into applet execution.

JavaCard applet for PIN verification

The sample applet implements the following logical steps:

- Allocation of PIN object (OwnerPIN())
- Initial setting of the secret value of PIN (OwnerPIN.update())
- Verification of the correctness of the supplied PIN (OwnerPIN.check())
- Get remaining tries of PIN verification attempts (OwnerPIN.getTriesRemaining())
- Set tries counter to maximum value and unblock blocked PIN. (OwnerPIN.resetAndUnblock())

```
// CREATE PIN OBJECT (try limit == 5, max. PIN length == 4)
OwnerPIN m_pin = new OwnerPIN((byte) 5, (byte) 4);
// SET CORRECT PIN VALUE
m_pin.update(INIT_PIN, (short) 0, (byte) INIT_PIN.length);
// VERIFY CORRECTNESS OF SUPPLIED PIN
boolean correct = m_pin.check(array_with_pin, (short) 0, (byte) array_with_pin.length);
// GET REMAING PIN TRIES
byte j = m_pin.getTriesRemaining();
// RESET PIN RETRY COUNTER AND UNBLOCK IF BLOCKED
m_pin.resetAndUnblock();
```

JavaCard applet for encryption of the supplied data

The sample applet implements the following logical steps:

- Allocation and initialization of the key object (KeyBuilder.buildKey())
- Set key value (DESKey.setKey())
- Allocation and initialization of the object of cipher (Cipher. getInstance(), Cipher. init())
- Receive incoming data (APDU.setIncomingAndReceive())
- Encrypt or decrypt data (Cipher.update(), Cipher.doFinal())
- Send outgoing data (APDU. setOutgoingAndSend())

```
// …. INICIALIZATION SOMEWHERE (IN CONSTRUCT)
// CREATE DES KEY OBJECT
DESKey m_desKey = (DESKey) KeyBuilder.buildKey(KeyBuilder.TYPE_DES, KeyBuilder.LENGTH_DES,
false);
// SET KEY VALUE
m_desKey.setKey(array, (short) 0);

// CREATE OBJECTS FOR ECB CIPHERING
m_encryptCipher = Cipher.getInstance(Cipher.ALG_DES_ECB_NOPAD, false);
// INIT CIPHER WITH KEY FOR ENCRYPT DIRECTION
m_encryptCipher.init(m_desKey, Cipher.MODE_ENCRYPT);
//….

// ENCRYPT INCOMING BUFFER
void Encrypt(APDU apdu) {
    byte[]   apdubuf = apdu.getBuffer();
    short    dataLen = apdu.setIncomingAndReceive();

    // CHECK EXPECTED LENGTH (MULTIPLY OF 64 bites)
    if ((dataLen % 8) != 0) ISOException.throwIt(SW_CIPHER_DATA_LENGTH_BAD);

    // ENCRYPT INCOMING BUFFER
    m_encryptCipher.doFinal(apdubuf, ISO7816.OFFSET_CDATA, dataLen, m_ramArray, (short) 0);

    // COPY ENCRYPTED DATA INTO OUTGOING BUFFER
    Util.arrayCopyNonAtomic(m_ramArray, (short) 0, apdubuf, ISO7816.OFFSET_CDATA, dataLen);

    // SEND OUTGOING BUFFER
    apdu.setOutgoingAndSend(ISO7816.OFFSET_CDATA, dataLen);
}
```

JavaCard applet for hashing of the supplied data

JavaCard 2.2.2 standard describes hashing functions MD5, SHA-1, SHA-256 and RIPEMD160. Not all must be implemented by a particular smart card.

The sample applet implements the following logical steps:

- Allocation of the hashing object (MessageDigest.getInstance())
- Reset internal state of hash object (MessageDigest. reset ())

- Update intermediate hash value using incoming array (MessageDigest.update())
- Finalize and read hash value of data (MessageDigest.doFinal())

```
// CREATE SHA-1 OBJECT
MessageDigest m_sha1 = MessageDigest.getInstance(MessageDigest.ALG_SHA, false);

// RESET HASH ENGINE
m_sha1.reset();
// PROCESS ALL PARTS OF DATA
while (next_part_to_hash_available) {
 m_sha1.update(array_to_hash, (short) 0, (short) array_to_hash.length);
}
// FINALIZE HASH VALUE (WHEN LAST PART OF DATA IS AVAILABLE)
// AND OBTAIN RESULTING HASH VALUE
m_sha1.doFinal(array_to_hash, (short) 0, (short) array_to_hash.length, out_hash_array, (short) 0);
```

JavaCard applet for computation of MAC based on symmetric cryptography

Various cryptographic checksum algorithms are implemented by the card (see javacard.security.Signature).

The sample applet implements the following logical steps:

- Allocation of the signature object (Signature.getInstance ())
- Allocation of the key object used for signature (KeyBuilder.buildKey ())
- Set key for usage with signature object in SIGN mode (Signature.init ())
- Generation of MAC over buffer (Signature.sign())

```
private   Signature      m_sessionCBCMAC = null;
private   DESKey         m_session3DesKey = null;

// CREATE SIGNATURE OBJECT
m_sessionCBCMAC = Signature.getInstance(Signature.ALG_DES_MAC4_NOPAD, false);
// CREATE KEY USED IN MAC
m_session3DesKey = (DESKey) KeyBuilder.buildKey(KeyBuilder.TYPE_DES,
KeyBuilder.LENGTH_DES3_3KEY, false);

// INITIALIZE SIGNATURE DES KEY
m_session3DesKey.setKey(m_ram, (short) 0);
// SET KEY INTO SIGNATURE OBJECT
m_sessionCBCMAC.init(m_session3DesKey, Signature.MODE_SIGN);

// GENERATE SIGNATURE OF buff ARRAY, STORE INTO m_ram ARRAY
m_sessionCBCMAC.sign(buff, ISO7816.OFFSET_CDATA, length, m_ram, (short) 0);
```

JavaCard applet for signing of the supplied data

JavaCard 2.2.2 standard describes various combination of signature functions based on asymmetric cryptography (RSA, DSA, ECDSA) and symmetric cryptography (MAC – DES, AES based). Again, not all must be implemented by a particular smart card.

The sample applet implements the following logical steps:

- Allocation of the key and signature objects (KeyBuilder.buildKey, new KeyPair, Signature.getInstance)
- On-card generation of key pair (KeyPair.genKeyPair())
- Obtaining references to private and public key (KeyPair.getPrivate/Public)
- Initialization of signature engine with private key (Signature.init)
- Performing signature operation (Signature.update, Signature.sign)

```
// CREATE RSA KEYS AND PAIR
m_privateKey = KeyBuilder.buildKey(KeyBuilder.TYPE_RSA_PRIVATE,KeyBuilder.LENGTH_RSA_1024,false);
m_publicKey = KeyBuilder.buildKey(KeyBuilder.TYPE_RSA_PUBLIC,KeyBuilder.LENGTH_RSA_1024,true);
m_keyPair = new KeyPair(KeyPair.ALG_RSA, (short) m_publicKey.getSize());

// STARTS ON-CARD KEY GENERATION PROCESS
m_keyPair.genKeyPair();
// OBTAIN KEY REFERENCES
m_publicKey = m_keyPair.getPublic();
m_privateKey = m_keyPair.getPrivate();

// CREATE SIGNATURE OBJECT
Signature m_sign = Signature.getInstance(Signature.ALG_RSA_SHA_PKCS1, false);
// INIT WITH PRIVATE KEY
m_sign.init(m_privateKey, Signature.MODE_SIGN);

// SIGN INCOMING BUFFER
signLen = m_sign.sign(apdubuf, ISO7816.OFFSET_CDATA, (byte) dataLen, m_ramArray, (byte) 0);
```

JavaCard applet for generating random data

JavaCard 2.2.2 standard defines two types of random generators within object RandomData: RandomData.ALG_SECURE_RANDOM and RandomData.ALG_PSEUDO_RANDOM. Sometimes, the ALG_PSEUDO_RANDOM is not implementated by the card.

The sample applet implements the following logical steps:

- Allocation of the random data object (RandomData.getInstance())
- Generation of random block with given length (RandomData.generateData())

```
private RandomData     m_rngRandom = null;

// CREATE RNG OBJECT
m_rngRandom = RandomData.getInstance(RandomData.ALG_SECURE_RANDOM);

// GENERATE RANDOM BLOCK WITH 16 BYTES
m_rngRandom.generateData(m_testArray1, (short) 0, ARRAY_ONE_BLOCK_16B);
```