

Lisa 5. ID-kaardi tarkvara analüüs, 14.12.2011

Toni Koivunen, Sauli Pahlman

Riigi Infosüsteemi Amet

reg kp 20.12.2011

reg nr 1-6/11-6-5-5

2 Bypassing signing and decryption PIN code validation

The implementation of PIN2 verification is inadequate and makes it possible for an attacker to sign documents without entering a PIN2 code. The same weakness affects the document decryption code.

In case a INS 2A (perform security operation) instruction is given to the card, to sign or decrypt documents, the following block of code is reached:

```
case (byte)0x2A: //L_La = perform_security_operation (GB, L_Lc );//- perform
security operation (dig sig, hash, decrypt)
    apdu.setIncomingAndReceive();
    if ( (GB[0] & (byte) 0x0C) != (byte) 0 )
    {
        L_CLA = GB[Offs_CLA];
        unpack_secure_message( GB );
        L _ L a =
compute_digital_signature (GB);//- set security environment
        break;
    }
    apdu.setOutgoingAndSend( (short) 0,
compute_digital_signature (GB) );
    return;
```

as the regular 2A code does not use secure messaging, the if block is not reached. Thus the code calls compute_digital_signature directly.

In compute_digital_signature there is a case where an instance of class Signature is created and updated as new data (from the document which is about to get signed) is received. This operation does not require the user to input any PIN code:

```
case (short)0x90A0: //Set Hash
    if ( GB[5] != (byte) 0x80 ) ISOException.throwIt(ISO7816.SW_WRONG_DATA);
    if ( G_bool[bo_SignatureInputFinished] ) O[o_RunningSignature] = null;

    if ( O[o_RunningSignature] == null )
    {
        byte Sign_Key_Ref = (byte)(G_byte[by_Signing_Key_Reference] - (byte)
1 );
        if ( Sign_Key_Ref < (byte)0 ) Sign_Key_Ref = (byte)(EESF_File.Body[19] -
(byte)0x1);
        switch ( Signature_Key[Sign_Key_Ref].getPrivate().getSize() )
        {
```

```
case (short) 160 :
case (short) 192 : O[o_RunningSignature] = EE_ECDSA_SHA_Signature;
break;
case (short) 224 : O[o_RunningSignature] = EE_ECDSA_SHA224_Signature;
break;
case (short) 256 : O[o_RunningSignature] = EE_ECDSA_SHA256_Signature;
break;
default: O[o_RunningSignature] = EE_RSASignature;
}
( ( S i g n a t u r e )
(O[o_RunningSignature])).init( Signature_Key[Sign_Key_Ref].getPrivate(),
Signature.MODE_SIGN );
}

if ( ( GB[0] & (byte) 0x10 ) != (byte) 0 ) { if ( L_Lc != (short)0x42 && L_Lc !=
(short)0x83 && L_Lc != (short)0xC3 ) ISOException.throwIt(ISO7816.SW_WRONG_LENGTH ); }
else G_bool[bo_SignatureInputFinished] = true;

if ( GB[6] == (byte) 0x81 ) { Size = Util.makeShort( (byte)0, GB[7] ); Offset =
(short) 8; }
else { Size = Util.makeShort( (byte)0, GB[6] ); Offset = (short) 7; }

((Signature)(O[o_RunningSignature])).update( GB, Offset, Size );

return (short)0;
```

Once the hash has been calculated, a command with 9E9A (P1P2) is sent and that is handled by:

```
case (short)0x9E9A: //Compute signature
if ( L_Lc != 0 )
{
byte Sign_Key_Ref = (byte)(G_byte[by_Signing_Key_Reference] -
(byte)1 );
if ( Sign_Key_Ref < (byte)0 ) Sign_Key_Ref = (byte)(EESE_File.Body[19] -
(byte) 1 );
CurrKP_1 = (KeyPair) Signature_Key[Sign_Key_Ref];
if ( CurrKP_1 == null ) ISOException.throwIt(ISO7816.SW_RECORD_NOT_FOUND);
if ( ! G_bool[bo_PIN2] )
{
//if ( G_byte[by_Command_With_Sec_Mess_Key_No] != (byte) 5 )
ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
//if ( G_byte[by_SM_TYPE] != SM_PRO_SEC )
ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
if ( GB[GB_Offs_Command_With_Sec_Mess_Key_No] != (byte) 5 )
ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
if ( GB[GB_Offs_SM_TYPE] != SM_PRO_SEC )
ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
}
```

```
}
G_bool[bo_PIN2] = false;
if ( CurrKP_1.getPrivate().getType() == KeyBuilder.TYPE_RSA_CERT_PRIVATE )
{
    EE_RSA_Cipher.init( CurrKP_1.getPrivate(), Cipher.MODE_ENCRYPT);
    L_La = EE_RSA_Cipher.doFinal( GB, (short)5, L_Lc, G_Scratch, (short) 0);
}
else
{
    if ( L_Lc > 32 )
    ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
    EE_ECDSA_NONE_Signature.init( CurrKP_1.getPrivate(),
    Signature.MODE_SIGN );
    L_La = EE_ECDSA_NONE_Signature.sign( GB, (short)5, L_Lc,
    G_Scratch, (short) 0);
}
    Decrement_Key_Usage_Counter ( (byte) 2 , Sign_Key_Ref, (short)0, GB );
    Util.arrayCopyNonAtomic( G_Scratch, (short) 0, GB, (short)0, L_La );
    return L_La;
}
```

The if (L_Lc != 0) block can be reached since in this command the attacker is safe to set Lc to 0 but still to send more data. The code is supposed to throw an exception and to return a error response code if PIN2 has not been given. However, the exceptions are thrown only if certain checks ("GB[GB_Offs_Command_With_Sec_Mess_Key_No] != (byte) 5" and "GB[GB_Offs_SM_TYPE] != SM_PRO_SEC") are passed. The attacker can bypass these checks by setting the respective bytes in the APDU to the needed values. I.e., the following APDU will end up with the previously set hash being signed:

```
apdu[0x00] = 0x10 // CLA
apdu[0x01] = 0x2A // INS
apdu[0x02] = 0x9e // P1
apdu[0x03] = 0x9a // P2
apdu[0x04-0x386] = 0x00
apdu[387] = 0x05 // GB_Offs_Command_With_Sec_Mess_Key_No != 5 throws
apdu[389] = 0x02 // GB_Offs_SM_TYPE != SM_PRO_SEC (SM_PRO_SEC = 2;) throws
```

The result of the weakness is that an attacker can sign documents without knowing the PIN2 specific to the card.

The same weakness affects document decryption code. The following case is reached in case of decryption command.

```
case (short)0x8086: //Decrypt
    short Chaining_Offset = G_i[i_Chaining_Offset];
    if ( (GB[Offs_CLA] & (byte) 0x10) != (byte) 0 )
    {
```

```
        if ( (short)(Chaining_Offset + L_Lc) > Max_Chaining_Length )
ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
        Util.arrayCopyNonAtomic( GB, (short) 5, G_Chaining_Buffer,
Chaining_Offset, L_Lc );
        G_i[i_Chaining_Offset] += L_Lc;
        return (short)0;
    }
    if ( Chaining_Offset != (short) 0 )
    {
        Util.arrayCopyNonAtomic( GB, (short) 5, GB, (short)(Chaining_Offset+
(short)5), L_Lc );
        Util.arrayCopyNonAtomic( G_Chaining_Buffer, (short) 0, GB, (short)5,
Chaining_Offset );
        L_Lc += Chaining_Offset;
        G_i[i_Chaining_Offset] = (short) 0;
    }
    byte Decr_Key_Ref = (byte)(G_byte[by_Decryption_Key_Reference] - (byte)1 );
    if ( Decr_Key_Ref < (byte)0 ) Decr_Key_Ref = (byte)(EESE_File.Body[9] -
(byte)0x1 );
    if ( Decr_Key_Ref >= 0x10 )
    {
        Decr_Key_Ref -= (byte) 0x10;
        CurrKP_1 = (KeyPair) Authentication_Key[Decr_Key_Ref];
        if ( CurrKP_1 == null ) ISOException.throwIt(ISO7816.SW_RECORD_NOT_FOUND);
        if ( ! G_bool[bo_PIN1] )
        {
            //if ( G_byte[by_Command_With_Sec_Mess_Key_No] != (byte) 4 )
ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
            //if ( G_byte[by_SM_TYPE] != SM_PRO_SEC )
ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
            if ( GB[GB_Offs_Command_With_Sec_Mess_Key_No] != (byte) 4 )
ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
            if ( GB[GB_Offs_SM_TYPE] != SM_PRO_SEC )
ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
        }
        if ( CurrKP_1.getPrivate().getType() != KeyBuilder.TYPE_RSA_CRT_PRIVATE )
ISOException.throwIt(ISO7816.SW_FUNC_NOT_SUPPORTED);
        L_Lc--;
        if ( L_Lc != (short) ( CurrKP_1.getPrivate().getSize() / (short)8 ) )
ISOException.throwIt( (short) 0x6700 );
        EE_RSA_Cipher.init( CurrKP_1.getPrivate(), Cipher.MODE_DECRYPT);
        L_La = EE_RSA_Cipher.doFinal( GB, (short)6, L_Lc, G_Scratch, (short) 0);
        Decrement_Key_Usage_Counter ( (byte) 1, Decr_Key_Ref, (short)0, GB );
        Util.arrayCopyNonAtomic( G_Scratch, (short) 0, GB, (short)0, L_La );
        return L_La;
    }
}
```

The attacker will only need to take care that the Lc (variable L_Lc in the code above) he is feeding to the applet will be set correctly so that it will not cause an exception in "if (L_Lc != (short) (CurrKP_1.getPrivate().getSize() / (short)8)) ISOException.throwIt((short) 0x6700);" However, calling setIncomingAndReceive() in method process() will fill the buffer with all data that was sent to the applet, regardless of which value was given to the length field Lc. That way, the attacker can set the same APDU bytes GB[GB_Offs_Command_With_Sec_Mess_Key_No] and GB[GB_Offs_SM_TYPE] to the values needed to bypass the exceptions.

The weakness can be fixed for example by separating the cases where secure messaging is used from the regular cases and in case of regular operations, always requiring that the PIN code is given, without exceptions. These issues do not seem like design errors but logical or version handling errors in the code. The code which would correct these vulnerabilities is actually commented out in the vulnerable if blocks:

```
        if ( ! G_bool[bo_PIN1] )
        {
            //if ( G_byte[by_Command_With_Sec_Mess_Key_No] != (byte) 4 )
            ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
            //if ( G_byte[by_SM_TYPE] != SM_PRO_SEC )
            ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
            if ( GB[GB_Offs_Command_With_Sec_Mess_Key_No] != (byte) 4 )
            ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
            if ( GB[GB_Offs_SM_TYPE] != SM_PRO_SEC )
            ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
        }
```

3 Using a home-baked PIN functionality scheme instead of JavaCard's builtint OwnerPIN

The author has implemented his own PIN handling code in methodcompute_PIN_reference_data instead of using the OwnerPIN class provided by JavaCard. There is no evident weaknesses in the implementation but the reasoning behind the decision to implement this using own code instead or using a ready made interface should be provided.