UDVIKLINGS OG FORENKLINGS STYRELSEN

The Danish Tax Authority

Requirements and guidelines for implementing digital signatures in Digital Cash Register Systems

Revision released April 2025

Version 1.5.2

Contents

1 Introduction 3

1.1 The main principles 3

1.2 Legal basis for the signature 4

1.3 Cryptography 4

1.4 Digital signature 4

1.5 Acquiring the RSA keypair through OCES CA 4

1.5.1 OCES CA certificate policy 4

1.5.2 Integration of the OCES certificate for signing purposes 5

1.6 Preliminary guidance on eIDAS based certificates 5

2 Technical requirements 5

2.1 General requirements 5

2.2 Example signature trail 6

2.3 Transactions to include in signature trail 6

2.4 2.2 RSA-SHA512-3072 7

2.4.1 General description of RSA and SHA-512 as a paired standard. 7

2.4.2 Using SHA-512 for hashing 7

2.4.3 Implementation for ECR/POS software of RSA: 8

2.4.4 Certificate requisition 8

2.4.5 Technical requirements 9

2.4.6 Practical example 11

2.4.7 Practical Python code example for signing. 13

2.4.8 Practical C code example for signing. 13

2.4.9 Practical PHP code example for signing. 14

3 Key Generation and Management 14

3.1 Responsibility of software vendor 14

3.2 Distribution 14

3.3 Storage 14

3.4 Accountability 15

3.5 Compromising of key to third party 15

4 Classic pitfalls regarding digital signature 15

4.1 Not utilizing the available signature script available on SKAT.dk. 15

4.2 Consistency of the signature: 15

4.3 Wrong format in the certificate 16

4.4 Wrong type of certificate 16

5 Resources 17

1 Introduction

To achieve compliance with the Danish Cash Register Act and Regulations pursuant to the Act all digital cash register systems are required to implement a digital signature.

The signature shall sign specific data from each receipt and be recorded in the electronic journal upon finalization of each transaction. It is also mandatory to export the signature to the SAF-T Cash Register XML.

For the creation of the signature, the system vendors need to use the following standard:

- A digital signature using an RSA 3072 bit key with a SHA512 hash function (RSA-SHA512 3072)

RSA-SHA512-3072 is the only variant accepted by the Danish Tax Authority and is the minimum standard utilized in OCES certificate standard (see section 1.5.1 '*OCES Certificate Policy*'). The public key corresponding to the private key, which is used to digitally sign the transaction data, must be provided as part of the certificate.

This document explains the basic principles and provides a step-by-step guide to implementing RSA-SHA512-3072 via OCES. Alternatively, the use of eIDAS for EU based suppliers is permitted. eIDAS is to be considered a valid alternative to OCES, i.e. in cases when a vendor does not have direct access to acquiring a OCES certificate. In so far you wish to use an eIDAS solution instead of the OCES certificate, please reach out to digital.salg@sktst.dk beforehand, as the supplier of your eIDAS certificate needs to be approved for validation.

1.1 The main principles

The figure presents an overview of the process of signing data with a chaining element:

Data from receipt is defined in section 2. The signing of data is done using RSA-SHA512-3072 which return the signatures.

The use of signatures from the previous receipt ensures a chain of signed data that should not be broken. This must be done in the same ECR or Point of sale (POS) or other logical representation of the point of sale (i.e. registerID etc.).

The signing process **MUST** be done during the completion of a transaction, not by batch processing etc.

1.2 Legal basis for the signature

With reference to:

Regulations «BEK nr 2246 af 30/11/2021» relating to requirements for digital cash register systems (the Cash Register Act) - §63 stk. 1 on digital sales registration systems:

«Alle handlinger via salgsregistreringssystemer skal registreres i systemets elektroniske journal (logges). Transaktionsdata i den elektroniske journal skal signeres digitalt med et dansk OCES-certifikat, udstedt til den erhvervsdrivende eller dennes leverandør af digitalt salgsregistreringssystem, så integriteten af data i den elektroniske journal kan verificeres i forbindelse med kontrol.»

With the signature, any alteration of the signed data without use of the private key of the businessowner/software vendor is made detectable. This adds a strong integrity measure to the electronic journal.

1.3 Cryptography

By requiring the use of RSA-SHA512-3072 digital signature one is utilizing the strength of asymmetric cryptography, also called public key cryptography.

Asymmetric cryptography is a cryptographic system that uses two related keys, a key pair: *private key* and *public key.* Any message that is encrypted by using the private key can only be decrypted using the matching public key. The public key is made freely available to anyone, while a second, private key is kept secret and only known to the vendor.

The advantage to asymmetric cryptography is that it eliminates the issue of exchanging keys over the Internet or large network while preventing them from falling into the wrong hands.

1.4 Digital signature

Digital signatures are based on asymmetric cryptography. Using a public key algorithm such as RSA, two keys that are mathematically linked are generated: one private and one public. The keys should always originate from the OCES/eIDAS certificate issued to the business owner/system supplier. To generate a digital signature, signing software creates a one-way hash (such as SHA 512) of the electronic data to be signed. The private key is then used to encrypt the hash. The encrypted hash is the digital signature. Digital signature provides integrity, authentication, and non-repudiation.

1.5 Acquiring the RSA keypair through OCES CA

1.5.1 OCES CA certificate policy

Generating the keys used in the RSA digital signing is a process carried out through the

acquisition of the OCES certificate. The Danish Agency for Digital Government outlines the regulations that the supplier of such certificates must always adhere to.

1.5.2 Integration of the OCES certificate for signing purposes

The digital signature must be based on the Danish OCES3 standard. It is important to note, that this is a change from the current OCES2 standard, which will be deprecated, and all certificates was made invalid 31st of October 2023, why all solutions must be migrated to OCES3. The certificate shall identify either the company using the cash register or the company providing the cash register service or facility. Therefore, a so called "Organisationscertifikat" eller "virksomhedscertifikat" (Company certificate) is to be used. In OCES3 terms this is the VOCES certificate.

The full OCES3 certificate (without privateKey) is to be provided in a PEM X.509 version within the <certificateData> element. This means that indicators "-----BEGIN CERTIFICATE-----" & "-----END CERTIFICATE-----" are expected and should be included.

Details of the OCES3 certificate can be found at Certifikater - MitID Erhverv (mitid-erhverv.dk)
An example on OCES3 certificate can be located here: https://www.ca1.gov.dk/certifikater/

1.6 Preliminary guidance on eIDAS based certificates

Details of the eIDAS certificate solution can be found at https://eidas.ec.europa.eu/efda/trust-services/browse/eidas/tls

If you are planning to make use of the eIDAS solution, please reach out to digital.salg@sktst.dk beforehand for approval and validation information.

2 Technical requirements

This section addresses the technical requirements for implementing RSA-SHA512-3072 signing of the individual transactions via OCES.

2.1 General requirements

1. The signature must be recorded in the electronic journal with a direct link to the full record of the original receipt.
2. The signature must be created for all transactions that are reported in the XSD element **cashtransaction (6.1 in Technical Description 1.5.0)** in addition to the CVR number (CompanyIdent) of the company. This includes all receipts that are given a transaction number, and normally comprises transactions that influence sales. Please see section 2.1.2 for further clarification of what types of transactions this includes.
3. It must be recorded which version of the private or secret key that was used to generate the signature of the receipt.
4. The format for creating the hash and signature must be identical to the data exported to the Cash Register XML format and is stated in section 2.2.4 (RSA. The data elements must be separated by ";" (semicolon). Further, the currency must be the same as stated on the receipt.
5. The signature must be created and recorded in the electronic journal when finalizing the transaction. Not by batch processing.
6. The signature from previous receipt must be derived from the signature value from the last receipt for the same company in the same cash register. See example in section 2.1.1.
7. When the previous receipt does not have a signature, for example after a fresh install, the

signature value must be set to “0” – number zero.

8. When exporting from the electronic journal to Cash Register XML the signature shall be recorded in the field ‘*signature*’ and the key version in the field ‘*keyVersion*’. The OCES certificate is stored in the ‘*certificateData’* element. All three elements are located at *company/location/cashregister/cashtransaction*.

2.2 Example signature trail

The signature trail must follow the structure of the SAF-T Cash Register XML. This means that the signature for a receipt must have the data from that receipt included, as well as the signature from the *previous* receipt (receipt number) for the same company in the same cash register.

When exporting to the XML datafile the receipts must be in the same order as shown below. This is to make it possible to verify the signature trail.

Company

- Location1
  - CashRegisterX
    - Transaction1X1
      - Signatur1X1
    - Transaction1X2
      - Signature1X2 (from Signature1X1)
    - Transaction1X3
      - Signature1X3 (from Signature1X2)
  - CashRegisterY
    - Transaction 1Y1
      - Signature1Y1
    - Transaction1Y2
      - Signature1Y2 (from Signature1Y1)
    - Transaction1Y3
      - Signature1Y3 (from Signature1Y2)

- Location2
  - CashRegisterX
    - Transaction2X1
      - Signature2X1
    - Transaction2X2
      - Signature2X2 (from Signature2X1)
    - Transaction2X3
      - Signature2X3 (from Signature2X2)

2.3 Transactions to include in signature trail

As the signature trail must follow the structure of the SAF-T Cash Register XML, **all transactions** that are exported to auditfile/company/location/cashregister/**cashtransaction** must be given a signature. They must therefore always be included in the signature trail. Due to this fact, the elements *signature, certificateData* and *keyVersion* are mandatory.

Normally the transactions reported in the **cashtransaction** relates to sales, both through increasing and decreasing the sales amount. However, depending on the preferences of the system vendor, also other additional transactions (transaction types) may be included in the

signature trail.

If for example “Opening of cash drawer” is also treated as a transaction by the system, and used for generation of signature and written as a transaction in the XML export (in <**cashtransaction**>) this must in addition be reported as an “event” in the XML with a reference to the transaction ID <transID>.

All fields required to create the signature are mandatory, and they must all be included in the cashtransactions along with the mandatory elements.

When different types of transactions are used, <transType> is to be filled out to distinguish the different transaction types (For example <TransType>CASHSAL</TransType>). The description of the TransType must be declared in “basics” table (BasicType 11). Please see the Code list for a table of PredefinedBasicIDs for BasicType 11.

The elements <transAmntIn> and <transAmntEx> must be filled with value **“0.00”** when there is no amount. **These elements cannot be left empty or excluded.**

2.4 RSA-SHA512-3072

2.4.1 General description of RSA and SHA-512 as a paired standard.

To begin with, it is important to note that there is a difference between RSA encryption and RSA signature. For all matters in the Digital signature in Digital Cash Registers you will need to use RSA signature. It's important to note that while the underlying RSA algorithm is the same in both cases, the code implementation details differ:

RSA encryption is used to secure data and allows only the intended recipient to decrypt and read the data. This happens when the sender encrypts the data using the recipient's public key, which only the recipient has the private key to decrypt. Only sender and recipient can read the message.

RSA signature, on the other hand, is used to verify the authenticity and integrity of data. This is the intent of the digital signature for in digital cash registers, and it allows anyone to verify that the sender of the data is who they claim to be, and that the data has not been tampered with since it was signed. For RSA signature the sender generates a hash value (a message digest) of the data they want to send and encrypts it using their private key. The recipient can then decrypt and verify the signature (the hash value) using the sender's public key.

In summary, RSA encryption is used for confidentiality purposes, while RSA signature is used to ensure authenticity and integrity of data.

2.4.2 Using SHA-512 for hashing

When RSA and SHA-512 are combined, RSA is used for the digital signing process, while SHA-512 is used for generating a hash of the data being signed. In practice the sender first applies a hash function to the transaction data to create a message digest. The sender then encrypts the message digest with the sender's private key (supplied through OCES) to create the sender's personal signature.

Upon receiving the message and signature, the receiver decrypts the signature using the sender's public key (derived from the XML-element <certificateData>) to recover the message digest and

hashes the message using the same hash algorithm that the sender used.

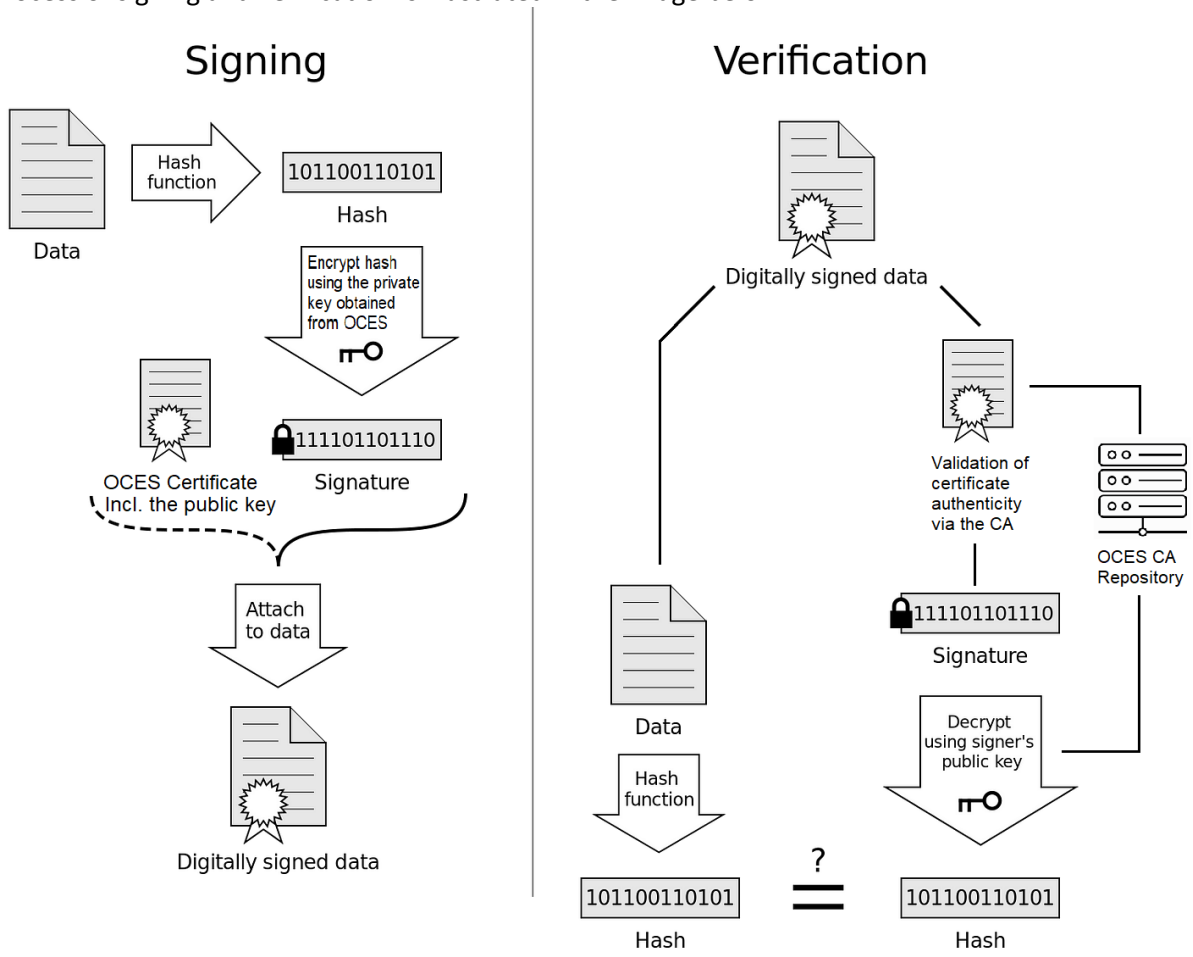
If the message digest that the receiver computes match the message digest received from the sender, the receiver can assume that the message was not altered while in transit. Note that anyone can verify a signature because the sender's public key is common knowledge. This technique does not retain the secrecy of the message; for the message to be secret, it must also be encrypted.

2.4.3 Implementation for ECR/POS software of RSA:

During audits the Tax Authorities will regenerate the hash and compare it with the hash derived from the signature stored in the electronic journal by using the public key obtained through the relevant certificate. The RSA signature must be generated and stored in the electronic journal upon each completion of the sale.

The Tax Authorities generates the hash by using the defined data values stored in the electronic journal. A match indicates that the data has not been altered by a third party without knowledge to the private key.

The process of signing and verification is illustrated in the image below:



If the hashes are equal, the signature is valid.

2.4.4 Certificate requisition

The Digital Cash Register Act demands that the signed transactions are done using the keys validated and delivered in the OCES certificate acquisition process (see 1.5 *Acquiring the RSA keypair through OCES CA*).

The certificate will provide information about the responsible signee and, among other data, contain the public key used in the RSA signature. The OCES CA will host the necessary information to validate the certificate, which can be requisitioned by the Danish Tax Authority on demand.

2.4.5 Technical requirements

IMPORTANT NOTE: *The process of verifying the signature will not be possible if the technical requirements are not met. This is because the recalculation of the SHA hash must be done with the exact same values as done with the signing process.*

When using the RSA algorithm (data encryption algorithm using the asymmetric key system, public and private key), the following guidelines (in addition to general requirements) must be applied:

1. The systems vendor is not allowed to use any other keypair than the ones validated by the OCES CA through the certification installation process (see Certificate Policy in Ressource section below for further information).
2. The public key must result from an extraction from the private key in PEM format (base-64).
3. The systems vendor must ensure that the private key used to create the signature is their unique knowledge and is properly protected in the software environment. See Key Management for further guidelines.

The following table describes what data to be signed and their order (see also section 6.1 *cashtransaction* in *Technical_description_Danish SAF-T format for Cash Register data*):

Table 1: Mandatory transaction data

Element in SAF-T Cash Register	Description of element	Format and requirements	Examples
signature	Signature from previous receipt	Base-64 If no previous receipt the signature value must be set to “0” – number zero	signature_from_previous_receipt
nr	Transaction number. This must be a unique, sequential number within a journal. This will be the same as the number stated on the issued receipt	IdentificationString36	123456789
transID	Transaction ID. Other unique internal, sequential ID used. by the cash register system. This can be the equivalent of the element “nr”.	IdentificationString36	11334455
transType	Transaction type. Description of the code MUST be declared in the ‘Basics’ table (basicType 11). See technical description section 3.20.	IdentificationString36	CASHSAL

transDate	Date at which the transaction was performed which should match the string value that is provided in the SAF-T file.	YYYY-MM-DD If you use time zones, make sure to have these incorporated into your signature in the exact same way as they are displayed in the SAF-T file.	2014-01-01 or 2014-01-01Z or 2014-01-01:+01:00
transTime	Time at which the transaction was performed which should match the string value that is provided in the SAF-T file.	hh:mm:ss If you use time zones, make sure to have these incorporated into your signature as well, the exact same way they are displayed in the SAF-T file.	01:01:01 or 01:01:01Z or 01:01:01+01:00
empID	Unique identification of the employee who has performed, the transaction (refers to the empID of the employee element). See section: 3.15 in technical description.	IdentificationString36	1003
transAmntIn	The amount involved in the transaction, including VAT.	Decimal Data Type: Numerical field with two decimals. Decimal separator "." (dot). No thousand separators. No leading or ending spaces.	1250.00
transAmntEx	The amount involved in the transaction, excluding VAT.	Decimal Data Type: Numerical field with two decimals. Decimal separator "." (dot). No thousand separators. No leading or ending spaces.	1000.00
registerID	Unique id of the register. Identical to *registerID* in section 4.3 in technical description.	String100	123.45678-A
companyIdent	The company's danish 8-digit CVR number. This must be identical to	No leading or ending spaces	12345678

	the number provided in section 3.1 «company\companyIdent»		

2.4.6 Practical example

All code examples are written and tested with OpenSSL and Cryptography in Python. We recommend using the PKCS1v15 padding as this has proven to work more consistently across different code formats but do accept PSS padding as well.

Signing of data:

1. Data to be signed:

Signatur_from_previous_receipt;transaction_data (see table 1)

Example:

apkUomoTd3dBJZ7EShaPAJd5kwPJ+zFGKkip6i8Vr5bp/9l7tQieCr0/Dlfm5sTI0+0b9qbXqcVWP+ts6P+XulTpVCxBsyw
O4ElycZ7XdEWSDYfwoGCXvXwsllsZKHk1a1FxzHb2CPGD44/8ETbYkIh8vJ0INckp3PoiL5N+Ljm6wBuN8qJ6ZSO8DhMJ
CUUUIjrUQnza/oTtdGgMQ1nD/YFLX4oXYkcWbynGQYnvfIUKS57PsMNSTW11XvdITPQbbm+DjP51TsatrRY799t+ozO
icqhLH44Z6s3UQgjWKT05cFtNYkwHmbEovu1o7DXQB1v7/JPHUPicneHKhmtaDreFFTHWFjqwOJSZ/6xT40BBt+UUV
e4RL6fkhxpFNKoHC5urVtpYHopsBNIGSQms+BkSg9Mb2CsCNLvkJkbEWKCTCfigD7kijkSy0d7qzUNiJ/W+XiLftkFdabXe
7mVuSq97XDwl57pyco4YehpVtvv2fgrZGir7qw88eJukTDch;123456789;11334455;CASHSAL;2014-01-
24;23:59:59;1003;1250.00;1000.00;123.45678-A;12345678

2. Hash data with SHA512 (this will return data in bytes):

```
message_sha512 = hashlib.sha512(bytes(message, 'utf-8')).digest()
```

3. Encrypt the byte value with private key and generate "signature_inbyte":

```
Signature_inbyte = private_key.sign(message_to_sign_sha512,
                                    padding.PKCS1v15(),
                                    hashes.SHA512())
```

4. Encode with base64string and generate "signature":

```
signature = base64.b64encode(signature_inbyte).decode('utf-8')
```

Example signatur:

MT+qKDAh2wg1zk2uTtBrIlkTHDgPhG5PvEjcWuo65Uu28bO7WKUUBmbGLzUvhiEAMVGZl1FyR2eIKDwP
Gc99hqvfzHFczQhCrRvlO020WMfO9FltU5qICzay9y40Riqn9ee3Bqq6FCiUwDyqdd0bfz8AQYFxFK0a+sWD
F7FfbEG0aXTbILIT1Olq+JOi613vwTsz69bj4vjtraD5Kl2CTcfbCYYv02FZHjwAvgBBqHS3jU3OWcvDCw46xv7
dY1DPDgPUbb9p/zF3NArcnSP9Cwjs2T/ri6o41s1Z4L9CGd//abaDpBARcxgfHk9VcrKTvNo19roXDxPn7Ham
iQ9fXqNuF3t0x5b7xX1Y3DIXqYz6c8HfaoH22kVHpgJgAJjADqBHTgArOxSV+kUBL2+uZ8baRJ/NM37+QzB8
A7JuFMYYVp/4TrdYZxswCG8Drgfk+rlLwbSfYjt6jahuOLQOZZLosRmN3RLiakvGbCmlVecbp/2RJOSVFgcTY+
Q7xw3e

How to verify the signature:

Mock certificate data for testing:

Certificate format: x.509 PEM

-----BEGIN CERTIFICATE-----
MIIGiTCCBL2gAwIBAgIUKjnFzNdu5RItpr+EpFJIh7k2hEowQQYJKoZIhvcNAQEK
MDSgDzANBglghkgBZQMEAgEFAKEcMBoGCSqGSIb3DQEBCDANBglghkgBZQMEAgEF
AKIDAgEgMGsxLTArBgNVBAMMJERlbiBEYW5za2UgU3RhdCBPQ0VTIHVkc3RlZGVu
ZGUtQ0EgMTETMBEGA1UECwwKVGVzdCAtIGN0aTEYMBYGA1UECgwPRGVuIERhbnNr
ZSBTdGF0MQswCQYDVQQGEwJESzAeFw0yMzA5MjAwOTM0MTRaFw0yNjA5MTkwOTM0
MTNaMIGfMRYwFAYDVQQDDA1TVElMLUlQTC1URVNUMTcwNQYDVQQFEy5VSTpESy1P
Okc6ZTVlYmMzN2EtNWFjNC00MDJjLTk1NzgtNzY3Y2IzZDk3ODU4MSYwJAYDVQQK
DB1UZXN0b3JnYW5pc2F0aW9uIG5yLiA5MzM3NzE2MzEXMBUGA1UEYQwOTlRSREst
OTMzNzcxNjMxCzAJBgNVBAYTAkRLMIIBojANBgkqhkiG9w0BAQEFAAOCAY8AMIIB
igKCAYEAoh/HN0u+6XmFvtcGY5lo4BGefzbPtlbhPrqXTC98R+vHhBJLx4GdKyOu
HMoE5S5bEXEqa8ZYKP/0SF+fIL0ZeprNvMYOil9Xb01Y9V/BbVCcVImCVNdszqMx
Qh0QBoFKrLT100hZUNLf4go46Lzg2mRw2g4jJbTM3uC29vfiD/hzu79WRijyU21Q
aB8Y3QLyBwL6z0uyy5KDgzGtilQPLFpfDbcucGrUotkwiycViUfaUEdd5PQUeAmQ
51Py+wHcQlHEMYPpCmCy9nCUUdHDFwp+yToqtMeX0RUQu3baJ2tqz4Ub6DtRR9rl
2nR8iOb1Gmo6YaVrogbX1uGAEquuDHpAE2TG0xtdUGnSV/x9A6p/AkTZA2tTajTV
6bjGM8K22oGQmAnPgEwph+l9Bl/ICTLovMG9oO1JEo5UUzxyomL9Yb1ANeVpRw/W
NTIxVEj1Wd4DTpg7HHChsmrCzaevnRYT4ThXRXzl05ms6RfRyqVDz06xJVXaV+ed
edjwvvXtAgMBAAGjggGGMIIBgjAMBgNVHRMBAf8EAjAAMB8GA1UdIwQYMBaAFH8o
n9lxmULidefXNXYuTQglbXZeMHsGCCsGAQUFBwEBBG8wbTBDBggrBgEFBQcwAoY3
aHR0cDovL2NhMS5jdGktZ292LmRrL29jZXMvaXNzdWluZy8xL2NhY2VydC9pc3N1
aW5nLmNlcjAmBggrBgEFBQcwAYYaaHR0cDovL2NhMS5jdGktZ292LmRrL29jc3Aw
IQYDVR0gBBowGDAIBgYEAI96AQEwDAYKKoFQgSkBAQEDBzA7BggrBgEFBQcBAwQv
MC0wKwYIKwYBBQUHCwIwHwYHBACL7EkBAjAUhhJodHRwczovL3VpZC5nb3YuZGsw
RQYDVR0fBD4wPDA6oDigNoY0aHR0cDovL2NhMS5jdGktZ292LmRrL29jZXMvaXNz
dWluZy8xL2NybC9pc3N1aW5nLmNybDAdBgNVHQ4EFgQUD6pFevOBC/FSoRzZnyjB
6cV3wikwDgYDVR0PAQH/BAQDAgWgMEEGCSqGSIb3DQEBCjA0oA8wDQYJYIZIAWUD
BAIBBQChHDAaBgkqhkiG9w0BAQgwDQYJYIZIAWUDBAIBBQCiAwIBIAOCAYEACmuR
Xivhc2m3CAWJC01LJAW+CmbNPXbcq3ovTTuqpvp/6/2YyP9xu9VeerUl34Wndl+j
naaXTHxrszWQO+1TEvN/ph6dXu7too0NwVt+wunL8+PCBOULR8Z2L9fSChyExPql
o4LheixLfRFnES67KT2Ny2OvtRpHqVUDF3qwzpra08j2yVEbbhdv1sQ0Sp9pNyez
fpSZzYY216qDj0M5rZMdjeP65rzi8h3wWUQqJcscJuxhvtR+RdHD0VX3X2qV/Zvb
tF7CvaqAJsARNBlRr7kcHrhmdI7MFP6/JQ11zEXYhgiNnuMv4kQKNPknq5NHDn1d
YjittCqAyGKlSIPrLhzJ2hnS+ieVxhKLSaxBHxyJRYkgvySYe/ijr+35XzzW8Jie
AR5IYCt48WiqoJk7AviERie+XwaR0Mc8QRk7kal/6yR34Qaqv8IokeKjzCF84Bh9
F/qx1nePVClxVADF4sLR31V+bfQaq++RCn9BE3/YgWCdYgUusoFft5nZv/Ui
-----END CERTIFICATE-----

5. **Generate hash with SHA512 with same unsigned data as in step 1 “data to be signed” and step 2 “Hash data with SHA512”**

Example (same as step 2):

Message_sha512=

b'5#\xac\x8e\xbe\x11\xfc\x9e\x87\xc0\x1e\xe3V:\xbbI\xe6\x0b\x1d\xd8^T\r\xbc\xf3\xe8_?C_HT\xc6\x88\x92\x9a\xc3\xddB\x96\xfb\xb1\xec\x7f\x8f.\xc7W_\xe2\xd3k[mK\xcff\x80\xf4\x84\x93?\xa4='

6. **Use public key on signed data to verify hash:**

```
    public_key.verify(
        signature,
        message_to_sign_sha512,
        padding.PKCS1v15(),
        hashes.SHA512()
)
```

7. **The hash values are the same and the data integrity is confirmed.**

2.4.7 Practical Python code example for signing.

```
2    """Read certificate, private and public key from a pfx file"""
     with open(pfx_path, 'rb') as f:
         pfx_data = f.read()

     pfx = OpenSSL.crypto.load_pkcs12(pfx_data, pfx_password)

     private_key = serialization.load_pem_private_key(
         OpenSSL.crypto.dump_privatekey(OpenSSL.crypto.FILETYPE_PEM,
     pfx.get_privatekey()), password=None,
         backend=default_backend())

     certificate = x509.load_pem_x509_certificate(
         OpenSSL.crypto.dump_certificate(OpenSSL.crypto.FILETYPE_PEM,
     pfx.get_certificate()), default_backend())

     public_key = certificate.public_key()

     """The message that needs to be signed"""
     message_to_sign = '0;123456789;11334455;CASHSAL;2023-10-
     21;04:22:29;1003;1250.00;1000.00;123.45678-A;123456789'

     """Generate SHA512"""
     message_to_sign_sha512 = hashlib.sha512(bytes(message_to_sign, 'utf-
     8')).digest()

     """Sign it with private key"""
     signature = private_key.sign(message_to_sign_sha512,
                                  padding.PKCS1v15(),
                                  hashes.SHA512())

     """Verify with public key"""
     try:
         public_key.verify(
             signature,
             message_to_sign_sha512,
             padding.PKCS1v15(),
             hashes.SHA512()
         )
         print('valid!')
     except cryptography.exceptions.InvalidSignature:
         print('invalid!')
```

2.4.8 Practical C code example for signing.

```
//Initialize the SHA-512 Context
SHA512_CTX ctx;
SHA512_Init(&ctx);
//Update Has Context with Data
SHA512_Update(&ctx, buffer, bytes); // 'buffer' contains the previous signature or
"0" for the first transaction.
SHA512_Update(&ctx, plainText, strlen(plainText)); // 'plainText' contains the
data to sign.
//Finalize the Hash
```

```
unsigned char digest[SHA512_DIGEST_LENGTH];
SHA512_Final(digest, &ctx);

//Sign the digest with RSA
int result = RSA_sign(NID_sha512, digest, SHA512_DIGEST_LENGTH, buffer, &bytes,
rsa_privkey);

//Encode the Signature in Base64
char* base64EncodeOutput;
int len = Base64Encode(buffer, bytes, &base64EncodeOutput); //
'base64EncodeOutput' will contain the base64-encoded signature.
```

2.4.9 Practical PHP code example for signing.

```
openssl_sign($message, $signature, $privateKeyResource, OPENSSL_ALGO_SHA512);
```

3 Key Generation and Management

The objective of key management is to achieve a situation in which the private key or secret key cannot be revealed or abused. Therefore, great responsibility rests with the software vendors to protect these keys.

This section presents guidelines and best practices for key generation and management.

3.1 Responsibility of software vendor

The software vendor must do a risk assessment based on the circumstances they are facing in the following:

- Protection of private/secret key used by the ECR/POS software available to their customers.
- Protection of private/secret key within the software vendor company premises.

The conclusions and actions taken must be documented and act as the basis for the management of secret key(s).

The consequence of private/secret keys being compromised is that the software vendor must contact the relevant CA responsible for managing the certificate to which the keypair was generated for.

3.2 Distribution

The generated keys shall be transported (when necessary) using secure channels. The distribution of the public key (asymmetric encryption using RSA) to the Danish Tax Authority is done utilizing the OCES CA key distribution solution.

Sharing of secret keys with other parties must not be done, unless stated by an industry agreement with the participation of the Danish Tax Authority. It is however permitted for the business owner to use the system suppliers certified keypair for the signing process. Insofar such a transaction of keys is performed internally within the system supplier or between the system supplier and the customer (business owner), it must be carried out adhering to the OCES CA rules and regulations.

3.3 Storage

The basis of key management is to ensure that the keys are stored in a secure manner. What constitutes a secure manner depends on how the environment for each cash register system is structured.

Regardless of the environment and whether the key is stored internally or externally, the following general protective measures should be considered:

- Developers must understand where cryptographic keys are stored within the application. Understand what memory devices the keys are stored on.
- Limit the amount of time the key is held in plaintext form, for example in volatile memory.
- Keys should never be stored in plaintext format, and humans prevented from viewing it in plaintext.
- Keys should be protected on both volatile and persistent memory, ideally processed within secure cryptographic modules.
- Keys should be stored so that no other than the privileged persons get access to it in plaintext form.

3.4 Accountability

Accountability involves the identification of those that have access to, or control of, cryptographic keys throughout their lifecycles. This can be an effective tool to help prevent key compromises and to reduce the impact of compromises once they are detected. Although it is preferred that no humans are able to view keys, as a minimum, the key management system should account for all individuals who are able to view plaintext cryptographic keys.

In addition, more sophisticated key-management systems may account for all individuals authorized to access or control any cryptographic keys, whether in plaintext or cipher text form.

3.5 Compromising of key to third party

If a key is compromised, the cash register system no longer fulfils the requirements in the Cash Register Systems Regulations. The supplier must, without undue delay, notify the CA of the OCES certificate from which the keypair is generated, of this and rectify the deficiency or withdraw the cash register system from the market, refer the certificate policy of the Danish Agency for Digital Government (see section "*4 Resources*").

4 Classic pitfalls regarding digital signature

In the following section, we introduce the typically identified pitfalls we observe in the implementation and use of digital signatures.

4.1 Not utilizing the available signature script available on SKAT.dk.

Working with digital signatures and cryptography validation in general involves many variables and uncertainties. Therefore, we recommend testing and verifying your signature algorithm with the available script. This ensures that your algorithm corresponds to what is expected in the SAF-T format.

4.2 Consistency of the signature:

It is crucial to check that the data in both the database and the signature are identical. Errors can occur where decimals or dates in the database differ from the signing data (in the SAF-T file) due to the length of the decimals and dates. This can be avoided by ensuring that the signature is consistent throughout.

- We advise to print the data behind the signature in your dataset and compare it directly with the data in your generated SAF-T file to validate that these corresponds.

Practical example:
For the digital signature expect the following format:

previous_signature;nr;transID;transType;transDate;transTime;empID;transAmntIn;transAmntEx;registerID;companyIdent

A typical error we see is discrepancy in the data signed vs. the data present in the SAF-T file, which is used for validation.

Data signed in backend:
0;123456789;11334455;CASHSAL;2014-01-24;23.59.59;Admin;1250,00;100,00;123.45678-A;12345678

Data available in the SAF-T file:
0;123456789;11334455;CASHSAL;2014-01-24;23:59:59;1003;1250.00;100.00;123.45678-A;12345678

Although quite similar, the data signed has different values for empID, transTime and transAmntIn transAmntEx. These need to be identical for us to validate the signature.

Rule of thumb is, that if you have used the signature script and gotten a 'Valid' or 'It worked', your signing algorithm is correct. If you still receive signature errors in actual SAF-T files it typically is caused by mistakes in signed data vs. data in the SAF-T as illustrated above.

4.3 Wrong format in the certificate

For us to validate the digital signatures provided in the SAF-T file, we must be able to read the OCES certificate also provided. As stated in section 1.5.2, the full OCES3 certificate (without privateKey) is to be provided in a PEM X.509 version.

4.4 Wrong type of certificate

To validate the certificate the certificate needs to be the correct type of certificate. If you are in doubt if you have the right certificate, please see the guide below from MitID to get the correct type of certificate.

Link to guide:
https://www.mitid-erhverv.dk/support/vejledning/anvendelse/brugeradministrator/certifikater/bestil-et-organisationscertifikat/

Please be aware that the certificate needs to be valid for the period that the transactions are signed. This means that if you have a certificate valid from the month of May but transactions are signed in April with this certificate we won't be able to validate the signatures.

5 Resources

OCES-standarden – Digitaliseringsstyrelsen (Agency for Digital Government)
https://digst.dk/it-loesninger/nemid/om-loesningen/oces-standarden/

NETS Certificate Policy for OCES-Virksomhedscertifikater (v.5)
https://www.nemid.nu/dk-da/om-nemid/historien_om_nemid/oces-standarden/oces-certifikatpolitikker/VOCES_Certifikatpolitik_v5.pdf

Agency for Digital Government - Certificate Policy
VOCES Certifikatpolitik V4 0 sep 09 Eng.doc (digst.dk)

European Union - eIDAS platform and information center
https://eidas.ec.europa.eu/efda/trust-services/browse/eidas/tls