

[print](#) | [close](#)

Compliance Encryption: A System i Signature

[System iNEWS Magazine](#)

[Carsten Flensburg](#)

Carsten Flensburg

Fri, 06/01/2007 (All day)

[Click here](#) to download the code bundle.

To report code errors, email [SystemiNetwork.com](mailto:info@SystemiNetwork.com)

Signing a document as proof of identity and intention has been done for centuries, and it's still the preferred method of providing

authentication and liability to a legal act (note: not intended as legal advice). More recently, public key infrastructure and the concept of digital signatures have laid the foundation for implementing an electronic method for signing documents. A digital signature adds the following properties to an electronic document or other type of stream file:

- Integrity — The sender and receiver are assured that the message has not been altered during transmission. If a message is digitally signed, any change in the message invalidates the signature.
- Authentication — Signatures are used to authenticate the source of messages. Ownership of the private key binds that key to a specific user, so a valid signature calculated using the related public key confirms that the message was sent by that user.

The Digital Signature Concept

As you may have noticed, a digital signature doesn't ensure confidentiality, which is achieved only through actual encryption of a document. Signing a document or stream file in no way protects the contents from being read or copied; signing merely provides a method of verifying ownership and integrity of a document or stream file. A two-step process creates a digital signature.

1. Using a specific and agreed-upon hash algorithm, a message digest is calculated based on the document or stream file that will be distributed.
2. Using the private key, the message digest is encrypted, resulting in a digital signature string.

After these two steps are completed, the document or stream file, together with the digital signature file, is distributed to one or more recipients. The recipient(s) can now verify the signature by following a similar two-step process.

1. Using the public key, the digital signature is decrypted and the original message digest is reproduced.
2. Using the agreed-upon hash algorithm, the message digest is calculated based on the received document or stream file.

Comparing the message digests should now produce a match. If a match is not produced, either the document or stream file has been changed and the integrity is broken, or the private key used to encrypt the message digest does not correspond to the public key and authentication is not provided.

Regardless of the cause, the signature verification fails if a match is not achieved. However, if the message digest comparison is successful, the match verifies the integrity of the document or stream file and authenticates the message's origin — the document or stream file has not been altered following the message digest calculation. The private key counterpart of the public key was used to encrypt the message digest, thereby confirming the link between the private key owner and the document or stream file.

Cryptographic Services Signature APIs

It should come as no surprise that the System i and i5/OS are fully equipped to participate in a digital signature scheme. Over the last three releases, the introduction of (and subsequent comprehensive enhancements to) the Cryptographic Services APIs has provided a feature-rich and useful set of cryptographic functions that supports all types of cryptographic needs, including signatures.

The group of APIs includes an API that generates a Public Key Algorithm (PKA) Key Pair (i.e., a private key and its corresponding public key), an API that calculates a signature based on an input data stream and a private key, and an API that verifies a signature based on an input data stream and a public key. For more information about these APIs, go to IBM's Cryptographic Services APIs page (publib.boulder.ibm.com/infocenter/iserics/v5r4/index.jsp?topic=/apis/catcrypt.htm). Specific APIs of interest include

- Generate PKA Key Pair (OPM, QC3GENPK; ILE, Qc3GenPKAKeyPair)
- Calculate Signature (OPM, QC3CALSG; ILE, Qc3CalculateSignature)
- Verify Signature (OPM, QC3VFYSG; ILE, Qc3VerifySignature)

Calling these APIs and implementing a digital signature calculation and verification scenario takes quite a bit of programming effort. To make things a little easier, I have created a set of Signature CL commands. You can use these commands as is or as a starting point for your own digital signature implementation and setup.

The Signature CL Command Set

The Signature CL command set includes the following key commands:

- Generate PKA Key Pair (GENPKAKEYS) generates a random PKA key pair for use with the PKA cipher algorithm RSA.
- Calculate Stream File Signature (CLCSTMFSIG) produces a digital signature by hashing the input data and encrypting the hash value using a PKA.
- Verify Stream File Signature (VFYSTMFISIG) verifies that a digital signature is correctly related to the input data.
- Check Cryptographic Support (CHKCRPSPT) checks whether a specified cryptographic algorithm is currently supported on the system, or it displays a list of informational messages of all currently supported cryptographic algorithms. This command lets you verify that your system actually does support the RSA algorithm.

To ease access to the key signature commands, I've collected all of them on a Signature Commands Menu (CMDSIG) that is included with the code for this article. I have also added a CL program that

creates all of the command and menu objects for you if you follow the instructions in the source header. Read "Source Specification" below for more information.

The Command Processing Programs

The Generate PKA Key Pair command processing program (CPP) is primarily a wrapper for the Qc3GenPKAKeyPair API. It lets you specify a key size and a public key exponent parameter as input to the key generation process. It also lets you set the path-qualified names of the two IFS stream files to hold the resulting private and public keys.

If any of the key files already exist, an error is returned. The two files inherit the data authority of the parent directory, so be careful to set the parent directory appropriately before running the GENPKAKEYS command.

The Calculate Stream File Signature CPP takes as input parameters the path names of the stream file to sign, the private key stream file to use in the signature calculation process, and the signature stream file to create and use to store the calculated signature files. The two remaining parameters deliver the PKA block format and the Signing hash algorithm.

Initially, the CPP registers a Termination Exit program that runs in the event of an abnormal termination of the program — for example, one caused by the CPP receiving or resending an escape message. The Termination Exit program ensures that all resources allocated by the CPP are correctly cleaned and released.

Next, the CPP reads the private key file and saves the value to be used in the Calculate Signature API call. Following an initialization of all of the Calculate Signature API parameters, the stream file that will be signed is opened and read in appropriate block sizes (for more information on how to optimize read feeds, read "[Tech Tips: Optimizing IFS Read Speeds](#)," February 2007, article ID 20807 at [SystemiNetwork.com](#)).

For each block read, the Calculate Signature API is called in a continuous process. As long as the API's Final Operation Flag input parameter is not set, the API saves the result of the signature calculation and uses it as the offset for the continued signature calculation on the subsequent call to the API.

When the last block of the input file is read, the Final Operation Flag input parameter is set accordingly before the Calculate Signature API is called. As a result, the API returns the calculated signature in the API's output parameter. Subsequently, the CPP stores the signature in the specified signature file.

The Verify Stream File Signature CPP has a set of parameters that are almost identical to those in the Calculate Stream File Signature CPP. Those parameters are the stream file to verify the signature against, the public key file to use in the signature verification process, and the signature file to use in the signature verification process.

The only difference between the two CPPs is that to verify a signature, a public key must be specified instead of the private key needed to calculate the signature. The two remaining parameters are also the same as previous, as they provide the PKA block format and Signing hash algorithm used earlier to calculate the signature. To produce a match, these parameter values must be the same for both the signature calculation and the signature verification processes.

As indicated by the matching parameter structure, the work of the CPP itself is also very similar to the Calculate Stream File Signature CPP, except for the obvious difference that it returns either a

completion message that signals a successful verification or an escape message that signals a verification failure.

Running the Signature Commands

To get an idea of how the signature commands participate in a digital signature setup, follow this example that runs a simple test. First, a PKA key pair is created and is subsequently used in the signature calculation and verification process against the document test.doc in the QOpenSys directory. To generate the PKA key pair, run the following command:

```
GENPKAKEYS PVTKEY ('/QOpenSys/PKA/
  Keys/cbx_key_1024.pvk')
PUBKEY ('/QOpenSys/PKA/Keys/cbx_key_1024.pub')
KEYSIZE (1024)
PUBKEYEXP (3)
```

Next, calculate the signature using the private key:

```
CLCSTMF SIG STMF ('/QOpenSys/test.doc')
  PVTKEY ('/QOpenSys/PKA/Keys/
    cbx_key_1024.pvk')
SIGFILE ('/QOpenSys/PKA/Signatures/
  cbx_sig_md5.sig')
PKABLK FMT (*PKCS1_00)
  SIGHASHALG (*MD5)
```

The signature is now calculated and stored in the signature file. Finally, verify the signature using the public key:

```
VFYSTMF SIG STMF ('/QOpenSys/test.doc')
PUBKEY ('/QOpenSys/PKA/Keys/cbx_key_1024.pub')
SIGFILE ('/QOpenSys/PKA/Signatures/
  cbx_sig_md5.sig')
PKABLK FMT (*PKCS1_00)
  SIGHASHALG (*MD5)
```

Now if you change just a single character in the document data and repeat the third step, you get a different result. You can also specify a different public key than the one created here or a different hash algorithm, and then check the outcome. Either way, the signature is no longer verified, in turn, verifying the System i's signing skills!

Don't let the fun stop here. Read "You Can Sign i5/OS Objects Too!," below, to learn how to sign objects. For additional background on RSAs and PKIs, go to Bruce Schneier's website and read the articles in his Communications of the ACM Columns section (schneier.com/essays-comp.html).

Thanks to Patrick Botz of IBM for his insight and contribution to the research of this article.

Carsten Flensburg is a **System iNEWS** technical editor.

Source Specification

The Signature commands are created from the following sources:

CBX6031 — RPGLE — Generate PKA Key Pair - CPP

CBX6031H — PNLGRP — Generate PKA Key Pair - Help

CBX6031V — RPGLE — Generate PKA Key Pair - VCP

CBX6031X — CMD — Generate PKA Key Pair

CBX6032 — RPGLE — Calculate Stream File Signature - CPP

CBX6032H — PNLGRP — Calculate Stream File Signature - Help

CBX6032V — RPGLE — Calculate Stream File Signature - VCP

CBX6032X — CMD — Calculate Stream File Signature

CBX6033 — RPGLE — Verify Stream File Signature - CPP

CBX6033H — PNLGRP — Verify Stream File Signature - Help

CBX6033V — RPGLE — Verify Stream File Signature - VCP

CBX6033X — CMD — Verify Stream File Signature

CBX6034 — RPGLE — Check Cryptographic Support - CPP

CBX6034H — PNLGRP — Check Cryptographic Support - Help

CBX6034X — CMD — Check Cryptographic Support

CBX603U — UIM — Signature Commands Menu

Compile and run the following source to have it build the commands and menu for you; please note the instructions in the source header:

CBX603M — CLP — Signature Commands - Build commands

—C.F.

You Can Sign i5/OS Objects Too!

Similar to the stream file signatures already explained and demonstrated in the main article, the System i also supports signing i5/OS objects. Signing objects is very useful to ensure the integrity of a software package or its components. This capability lets the object creator certify that the object being signed is trustworthy (as of the time the object is signed) and lets object users verify this property.

The QSYS file system supports the *PGM, *SRVPGM, *MODULE, *SQLPKG, *FILE (save file), and *CMD object types. For more information on object signing and signature verification, including object signing scenarios, go to publib.boulder.ibm.com/infocenter/iseres/v5r4/index.jsp?topic=/rzal/rzalzosintro.htm.

An entire set of object signing APIs and facilities for Digital Certificate Manager (DCM) and iSeries Navigator have been developed to support signing executable objects. To check signed i5/OS objects, use the following commands:

- Check Object Integrity (CHKOBJITG)
- Check Product Option (CHKPRDOPT)
- Save Licensed Program (SAVLICPGM)

For more information on object signing APIs, go to the IBM Information Center API Finder (publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp?topic=/apifinder/name_finder.htm) and look up the following APIs:

- Add Verifier (OPM, QYDOADDV; ILE, QydoAddVerifier)
- Retrieve Object Signatures (OPM, QYDORTVO; ILE, QydoRetrieveDigitalSignatures)
- Sign Buffer (OPM, QYDOSGNB; ILE, QydoSignBuffer)
- Verify Buffer (OPM, QYDOVFYB; ILE, QydoVerifyBuffer)
- Verify Object (OPM, QYDOVFYO; ILE, QydoVerifyObject)

—C.F.

Source URL: <http://iprodeveloper.com/rpg-programming/compliance-encryption-system-i-signature>