QDL's Crypto Module

Introduction

This is QDL's module for a bit of basic cryptography. This is not intended to be a full suite of cryptographic tools, but tried to capture a Pareto's Law selection (20% of the calls that do 80% of the work). It lets you create **RSA** (Rivest–Shamir–Adleman), **EC** (Elliptic Curve) and **AES** (Advanced Encryption Standard, used for symmetric) keys and encrypt or decrypt them. The basic structure is that of a JSON webkey as a stem. It also allows for importing and exporting JSON webkeys to files, as well as certain **PKCS** (Public Key Cryptography Standards group) formats. It can do basic reading of **X 509** certificates.

Variable reference

$$KEY_TYPE

There is a single variable that is global, **`$$KEY_TYPE`** and this contains the following key types which are used in the import/export as well as key creation functions.

Key	Value	Use	Description
jwks	jwks	I/E	JSON webkey format
pkcs1	pkcs1	I/E	PKCS 1 private key format
pkcs8	pkcs8	I/E	PKCS 8 private key format
public	public	I/E	PKCS 8 public key format (“ “ “)
x509	x509	I/E	Same as PKCS 8 public and used as the public key format in an X 509 certificate, so many just refer to it as “x 509 format”.
rsa	rsa	create	RSA key
ec	elliptic	create	Elliptic curve key
aes	aes	create	AES (symmetric) key

Key:

I/E = used for **`import/export`**
create = key creation with **`create_key`**

Example

```
crypto#import(file_path, $$KEY_TYPE.'public')
```

would try to import a PKCS 8 public key from the given file_path.

QDL's Crypto Module

Function reference

b64_to_int

Description

Convert a base 64 encoded string to an integer. In many cryptographic applications it is required to change a base 64 string to an immense integer and possibly the hexadecimal representation of the integer. This is just another way of sending byte arrays.

Usage

`b64_to_int(arg)`

Arguments

arg is a string, a set of strings or a stem of strings. Non-strings are ignored.

Output

An integer, set of integers or stem of integers conformable to the argument. This is the integer representation of the string. Note that the output is exact, *however*, depending on how you have set numeric_digits algobraic operations on returned values may lose their exactness. It is best to process returned valus minimally.

Example

```
   crypto#b64_to_int('eHuCI7jCg4fwfXWf_TVTbvH-4pw0x2h9nf1RSEdqraI')
5449576220904040400113633670447028557288419873251412634989090468466896893891
```

code_challenge

Description

Create an RFC 7636 code challenge with the SHA-256 hash of a random verifier string.

Usage

`code_challenge(verifier)`

QDL's Crypto Module

Arguments

`verifier` is a string or set or stem of strings. These are usually base 64 encoded random strings that are a multiple of 8 bytes and must be less than 128 bytes total.

Output

A base 64 encoded SHA-256 hash of the verifier., or set or stem of them, conformable to the argument.

Example

```
   verifier := random_string(64); // 64 chars = 48 bytes
   verifier;
AJqLTlmTqqAW4N-iv7c20ETPP2PYZehi8mL7G-5KiDaPZ9o5vknXe38lauI3TCgY
   code_challenge := crypto#code_challenge(verifier);
   code_challenge;
oSpP8an566XxzrzVQOtMRWZfCGwAdezSbUkZVNC1F4I
```

create_key

Description

Create an RSA, EC or AES key subject to given parameters. AES keys are used for symmetric encryption.

Usage

`create_key()` - create an RSA key of 1024 bits, using the RS256 algorithm

`create_key(length)` create an RSA key of the given number of bits (must be a multiple of 256) using the RS256 algorithm

`create_key(parameters.)` - Create an AES, EC or RSA key

Arguments

`length` – (Only for and RSA key) the number of bits. Must be a multiple of 256. See below for more.

`parameters.` – a stem. There are three sets of parameters available for AES RSA and Elliptic Curve (EC) keys.

QDL's Crypto Module

AES key parameters

Name	Req?	Value
alg	N	The algorithm. Supported algorithms are none (default) A128GCM A192GCM A256GCM
length	N	Number of bits in the key. 112 <= length and length must be divisible by 8. Default is 256.
type	Y	$$KEY_TYPE.'aes'

RSA key parameters

Name	Req?	Value
alg	N	The algorithm. Supported algorithms are RS256 (default) RS384 RS512
length	N	Number of bits in the key. Must be a multiple of 256. Defaults is 1024
type	Y	$$KEY_TYPE.'rsa'

EC key parameters

Name	Req?	Valie
alg	N	Supported algorithms are ES256 (default) ES384 ES512
curve	N	The curve to use. Supported curves are Ed25519 Ed448 P-256 (default) P-256K P-384 P-521 X25519 X448 secp256k1
type	Y	$$KEY_TYPE.'elliptic'

QDL's Crypto Module

Output

A stem with the resulting key.

Examples

An AES key at 256 bits.

```
print(crypto#create_key({'type':'aes','alg':'A256GCM':'length':256}))
   alg : A256GCM
     k : nNgJT77SxxN0l57nCw2WJq0Q3IFaKt5T-
         Lz4p7GfnHocIjIwEMZqRlr9qBnNoK7g9mmvF33kx4JVCyaXaGHSkQ
   kid : 6RI6s6fAX50
   kty : oct
```

An RSA key. The default is 1024 bits. Display width is set to 72, since some are quite long

```
  print(crypto#create_key(), 72)
alg : RS256
  d : hDzkT_3B3GKajhSDBPA_koRQOwO-KCux_VDEG58yH4wdFjEx1xLefM14xS_w9K2BbC
      MSY33TCQ5XWPBexGuM-RbfQtZ5ZSYalVWw_QGyPzY2dnMAFAiwCKAxblDElkkKWo5o
      QD9yD71HrkEd1G7CAGby_k8jbpx7x6TSQolX-J0
 dp : wsnT1lf5wjXkcMon-lpXbYD-TUdZ1EwOU398zGU85GQ37hnQ1BHfsxUNscKHcLjsQF
      fIcgf0dEw-SjCZJG0SsQ
 dq : JSwRjLA6aFqYA5gQ95ie86nyzHdR5BZYYduyTP2tK07AjaGhQyY9MLyH5hUdWuoP71
      cpntL1Y_zhOMEGBTioVw
  e : AQAB
iat : 1735045281
kid : CCB6D75A73F1F2F8
kty : RSA
  n : mngD5D-5cxK-d3FQmcP2y61QpU274DuL1XflX3KTv80sKASGydsJ6eNciJREQ26nmh
      FStGO82P-PKn4k03RBNA7V5Ukpm5wjH6-3yaj12kAAOxCfBC12nnJZX50xLHhN2qq5
      0fk6DzwQoyc7gCxSCo5lPbwJvSLP00jGS96ydb0
  p : 28Rx824bi5SRAXB5BXmb1aiLZeHH7CQdVOyTJNo12ComptfWdUa047WuljH22PjmaL
      sKg7irMNV01ZTDamW2Tw
  q : s--QOmwtKgOs2Dy-g5ggwS9CvEDwH-v3Tla3djISgs5wSPOrdOaczm-b6S2KID8h0N
      NWd7NjMS-KHlVnmnmcMw
 qi : dIOAlQiVjV5fYI1mPI7S6rcTNvjpYJqsZeVJvHovUEdUdPOf16jyaNel_1xttSTSRb
      5uvVVaCU2i19ojUtjZgQ
use : sig
```

An EC key with specified algorithm and curve.

```
  print(crypto#create_key({'type':'elliptic','alg':'ES256','curve':'P-256'}))
   alg : ES256
     d : DVrqf9-yXI37aWVnUmkaRQcH-KQlrCUHxtlltdXoBb4
   kid : 7F50DDCC6B757372
   kty : EC
   use : sig
     x : fdc_cSgAiTKYm2byYD67GnF8VoE82g2n6jXU_JzFi9I
     y : XkzpiKF1LFmWzWnDW85dt0YUUq-VaKK4ovPmBQ_2hgY
```

QDL's Crypto Module

decrypt

Description

Decrypt (i.e., reverse the encryption) for string or stems/sets of them.

Usage

`decrypt(arg | arg., key.)`

`ecrypt(arg | arg., key.,  use_private_key)`

Arguments

`key.` - a key to be used. Only RSA or AES keys are supported

`arg` – a simple string, or a set of strings. Any non-string entries are simply returned unchanged.

`arg.` **–** a stem of strings. Each entry will be decrypted

`use_private_key` **–** Use the RSA private key to decrypt (default is false).

Output

If a simple string, the string is decrypted. If a stem or set, each element is decrypted. If the entry is not a string, it is not touched.

Example

Create a symmetric key and encrypt a string, then decrypt it. Note there are no length restrictions on the string for AES keys.

```
   aes. := crypto#create_key({'type':'aes', 'length' : 1024})
   crypto#encrypt('The quick brown fox', aes.)
NS5Uov65heE66PVYdg9fkM7vjw
   crypto#decrypt('NS5Uov65heE66PVYdg9fkM7vjw', aes.)
The quick brown fox
```

Another example using an RSA key. This uses the default key length of 1024 bits.

```
   rsa. := crypto#create_key()
   x := crypto#encrypt('The quick brown fox', rsa.)
   x
aQFQOZoCW6i8M4pmDdOCB9KgxPwvmUSZcbmwbOyWG6RwzP…
   crypto#decrypt(x, rsa.)
The quick brown fox
```

QDL's Crypto Module

encrypt

Description

Encrypt a string or stem/set of them using a key. Only RSA or AES keys are used.

Usage

```
encrypt(arg | arg., key.)
encrypt(arg | arg., key.,  use_private_key)
```

Arguments

`key.` - the stem for the key

`arg` – the target of the encyption. May be a string or set of them. Note that in complex arguments, non-strings are ignored.

`arg.` - a stem of strings. Each will be separately encrypted.

`use_private_key` - if true, encrypt with the RSA private key. Default is true.

Output

The object with its entries encrypted. Note that and RSA key will use public/private key encryption (so the default is to encrypt using the private key.) RSA keys strict the size of the input to being less than the key size. An AES key is a symmetric binary key. The encryption used here is quite basic and its inverse function is **`decrypt`**.

Examples

Nota Bene: The standard requires that the string to be encrypted is shorter (in bits) than the key length. For 1024 bits this means there are at most 1024/8 = 128 characters allowed, less a bit for overhead. In the next example, we create a 200 character long string and try to encrypt it using the 1024 bit key. It gives an error message and tells you the maximum size of the string you can process:

```
   rsa. := crypto#create_key()
   crypto#encrypt( random_string(200), rsa.)
illegal argument:encrypt could not process argument for key='0' with value
='kJS5_Q8oC1... (Data must not be longer than 117 bytes) At (1, 7)
```

Compare this with and AES key that has no restriction. We encrypt a 1000 character (8000 bit) string.

QDL's Crypto Module

```
   aes. := crypto#create_key({'type':'AES', 'length' : 1024})
   y :=crypto#encrypt(random_string(1000), aes.)
   size(y)
1779
   y
KXVLz-SGqvg0vKJQVgBW6MflwNZvzdPrPqkuPFI6LQ7eFNP…
```

Example of encrypting a stem.

In this example we can get around the length restriction by having a stem of strings. Each entry is encrypted:

```
   Jabberwocky. :=[''Twas brillig, and the slithy toves',
                  '   Did gyre and gimble in the wabe;',
                 'All mimsy were the borogoves,',
                  '   And the mome raths outgrabe.']
   j. :=crypto#encrypt( jabberwocky., rsa.)
   print(crypto#encrypt(rsa., jabberwocky.), 72)
0 : UosA6cdoAHJFOl7f9WhRI6ZgJ7CqHlbxRyhf8t4SqRb9fNGZhZJuTqszo8dP2HLbOTCm
    YmQr3CprCseQOKhW84fPx3wvqcB12hSu2PNqhfiYYEqeyBj5XdTlMAVvsZeco2GW3tng
    YSbGCgqczy_Dqud_-7KswUyOrz4QPL4e3hE
1 : K1c2DsHRJN2_-ArqkbTzrOwQxdhMhBlg9NyuTcKj5rgp1WQpgpKlyRwwWktAMT9KNmO6
    FqSsbO-IXrJxHxbPgQswzTzgTDO89PziM5ajp-EJFcMTZAP-rWwTbh3Y3OxaUBqcU2qX
    MSjc53higmmkuvkzyVdpDakPPUhwyu1bur0
2 : J4wVKF44Sj-STcvHMXojR82Mht362DCtPh5gLQ0DP_SgX62ZHLq8Uyp7UoIP92KTVZjT
    LTl6G7zzCUQHt5cTPgajIXlbOeBD81kV4-akYT8nzW2i4LzyGQ6D4JjGhqDiO0YLZzm1
    4gth-TMKrzGz_Rz7lmLWhlvLMPYMR3FY3Ao
3 : aoe42_bmni1iQT6djFO4ZFAANR4G0OyAkMLzcexwUz9oWgCVs7-r8IPJmS135-495l0J
    LpTjlBV9orrjmkYIQl4P9sfz3qMeyUEWPZNAGYIIcbHvDTKHGCMk1aXKNgGs5neBi8fx
    W2-t-D4Ch3qKfMt1vVbHPRyrayyOWUQUj7M
   crypto#decrypt(j., rsa.)
['Twas brillig, and the slithy toves,
   Did gyre and gimble in the wabe;,
  All mimsy were the borogoves,,
   And the mome raths outgrabe.]
```

Example of encrypting a set with a symmetric (AES) key.

Complex data structures like sets are processed in full. A simple example is the following

```
   crypto#encrypt({'a',{'b'}}, aes.)
{AA,{Aw}}
   crypto#decrypt({'AA',{'Aw'}}, aes.)
{a,{b}}
```

Note that the nesting of sets is preserved.

QDL's Crypto Module

export

Description

Export a key or set of keys to RFC 7517 format (JSON Web Key) or various PKCS formats and write to storage.

Usage

```
export_jwks(keys., file_path)
export_jwks(keys., file_path ,type)
```

Arguments

`keys.` - either a single key or a set of keys.

`file_path` - the fully qualified path for the output.

`type` – If omitted, the default is JWKS. Supported file types are

Type	Req?	Description
jwks	N	Use RFC 7517 format. This is the default
pkcs8	Y	Use PKCS 8 PEM format
x509	Y	Use the PEM format in X 509 certificates (which is really just a subset of PKCS 1).

Nota Bene: We do not support writing PKCS 1 files at this time, just PKCS 8. the reason is that PKCS 1 is mostly deprecated and hard to get to interoperate. PKCS 8 is now the universal standard. We do, however, read PKCS 1 files fine.

Output

This returns true if the operation successfully wrote the file. Otherwise, an error will be raised. Multiple keys are supported in JWKS, but only single keys in PKCS formats. To write multiple keys in PKCS would require either a PKCS 12 or Java keystore (JKS) file which would also require PKCS 5 (password protection) support, hence is currently not supported in QDL.

Examples

```
kk. := create_key(1024, 3)
export(kk., '/tmp/keys.jwk')
```

QDL's Crypto Module

```
true
```

This means that the set of keys was written in the correct format to the given file.

from_jwt

Description

Take a JWT or stem of them and convert to their stem payload. No verification is done, call **`verify`** for that. Note that non-JWT strings and other values (such as integers) will simply be returned unaltered.

Usage

`from_jwt(jwt | jwt.)`

Arguments

`jwt` – a single string that is a JWT.

`jwt.`– a stem or set of JWTs

Output

Returns a stem (for a single jwt) that is the payload of the JWT. If you supply a stem of JWTs, each will be converted to its payload.

Example

The keys and payload are exactly as in the example from the section on to_jwt, so look there.

```
   rr ≔crypto#to_jwt(p., rsa.);
   crypto#from_jwt(rr)
{a:q, b:{s:t}}
```

In this case, a simple stem, **`p.`** is created along with an **`rsa.`** key. It (**`p.`**) is turned into a JWT, **`rr`**, then back to show this works. To show how this operates on a more general stem,

```
   crypto#from_jwt({'A':rr,'B':'foo'})
{A:{a:q, b:{s:t}}, B:foo}
```

A stem that consists of the JWT and a random string is used. The JWT as expected is converted back to its payload, the non-JWT is unaltered.

To check verification,

QDL's Crypto Module

```
   crypto#verify({'A':rr,'B':'foo'}, rsa.)
{A:true, B:false}
```

which shows that the entry for **A** is a valid JWT, the entry for **B** is not.

hex_to_int

Description

Convert the (string) hexadecimal representation of a number to an integer.

Usage

```
hex_to_int(arg)
```

Arguments

arg is a string, set of strings or stem of strings. Non-string scalars are ignored.

Output

An exact integer created from the hexadecimal value.

Example

```
   numeric_digits(100)
15
   c#hex_to_int('37fce3336733c5921635aafe956522b92efe3da73087e93e2ff024a637ba0266')
2.532402021465026940301358379119819916955149716252466682016577051091381950935E76
```

import

Description

Read JSON webkeys (as per RFC 7517) or a PKCS format key

Usage

```
import_jwks(file_path)
import_jwks(file_path, type)
```

QDL's Crypto Module

Arguments

`file_path` - the fully qualified path to the file.

`type` – The type of the key. Supported types are

Type	Req?	Description
jwks	N	Use RFC 7517 format. This is the default
pkcs1	Y	Use PKCS 1 (deprecated RSA format).
pkcs8	Y	Use PKCS 8 PEM format
x509	Y	Use the PEM format in X 509 certificates (which is really just a subset of PKCS 1).

.

Output

This returns a stem of of keys for JWKS if there are multiples, or a single key for JWKS if there is one and a single key for PKCS format.

Examples

```
   keys. := import_jwks('/tmp/keys.jwk');
```

Since there were no errors, the set of keys in RFC 7517 format was successfully imported and converted to a stem. Note that if there was one single key in the file, a single key would result.

int_to_b64

Description

Convert a (possibly very large) integer to its base 64 encoding. Some cryptographic applications use large integers as another way of sending byte arrays. This lets you operate on those. Note that the built-in QDL **`encode`** function does not alter integers.

Usage

`int_to_b64(arg)`

Arguments

arg – an integer, set or stem of them. Any other scalar is left unaltered

QDL's Crypto Module

Output

A string, or set or stem of them encoded to base 64 and conformable to the original argument

Example

```
   code_verifier := size(random_string(48));
   code_verifier;
qBdfP8Wmpomgkq6aJwcvZQMHx553RK4P7LAYxmzMAkmo8cM7MlE8ViJSOx38nlHr
   crypto#int_to_b64(crypto#hex_to_int(hash(code_verifier, 'sha-256')));
N_zjM2czxZIWNar-lWUiuS7-Pacwh-k-L_Akpje6AmY
```

this is to create a code challenge for a PKCE (RFC 7636) exchange.

int_to_hex

Description

Convert a possibly large integer to its hexadecimal representation. The standard QDL **`encode(arg, 16)`** function is not the same, since it convert its input, a string, to its RFC 4648 compliant base 16 encoding. **`encode`** does not alter integers.

Usage

`int_to_hex(arg)`

Arguments

arg is a string, set of them or stem of them.

Output

A string or a set or stem of them, conformable to the argument.

Example

```
   2^45
35184372088832
   crypto#int_to_hex(2^45)
200000000000
```

QDL's Crypto Module

The first value is the decimal representation, the second is the hexadecimal value *as a string*. Note that these fit nicely into the default 15 digit numerical precision, but it should be set higher for other values. Compare

```
   numeric_digits(100)
15
   3^45
2.954312706550833698643E21
   crypto#int_to_hex(3^45)
a0275329fd09495753
```

If you had gotten the value from one of the other functions, the value would be precise:

```
   numeric_digits(15);
100
    a := crypto#hex_to_int('a0275329fd09495753'); // get the value from a function
   // This will be exact:
   crypto#int_to_hex(a)
a0275329fd09495753
   // This will incur rounding since percision is 15 places.
   crypto#int_to_hex(3^45)
a0275329fd0910e780
   crypto#hex_to_int('a0275329fd0910e780')
2.95431270655083E21
```

This is because computing the value puts it in the current precision. When dealing with possibly large integers in cryptography, process them the minimum possible. Do not just treat them like numbers, but as very specific byte arrays. On the other hand, QDL will allow you to set the numeric precision to effectively infinity (limit is your hardware, not QDL), so if you ***really*** need to do something with one, it is possible. And it is always safest to keep them in string representations, specifically hex if you need to send them to someone else.

read_oid

Description

Read an entry from an X 509 certificate using its OID (Object Identifier). This is a low-level operation but is often about the only way to get certain values.

Usage

```
read_oid(cert., oid | oids.)
```

QDL's Crypto Module

Arguments

`cert.` - stem that represents the X 509 certificate.

`oid` - A single OID (of the form x.y.z….) You must know this

`oids.` - A stem of oids.

Output

If a single oid is requested, the response is the base64 encoding of the ASN 1.1 octet stream. Since each OID specifies how to interpret this binary array, this is about the best we can do in general.

If a stem of OIDs is sent, each entry of the will be the base64 encoded octet stream.

Note that when you read a certificate, the criticial and non-criticial OIDs are returned.

Example

The GitHub cert that is read in the **`read_x509`** section is used here.

```
   crypto#read_oid(cert., '2.5.29.14')
BBYEFDtoPzQ69Uc0yu-mTj2avV5uesyf
```

This reads a single OID. Again, the octet stream is encoded faithfully, but there is no canonical way to interpret a general ASN 1.1 entry.

read_x509

Description

Read an X 509 certificate or certificate chain. We say *read* instead of import since you cannot alter a certificate without invalidating it, hence there is no way to write any changes.

Usage

`read_x509(file_path)`

Arguments

`file_path` - the path (VFS paths are of course supported) to the file holding the cert(s).

QDL's Crypto Module

Output

If there is a single certificate, then a stem representing that. If there is a certificate chain, then the result is a list of the certificates in the order found.

Example

in this example, I downloaded the certificate from the GitHub main site and am going to read it. A bit of truncation and formatting is done to make it display nicer.

```
        cert.:=crypto#read_x509('/home/ncsa/Downloads/github-com.pem')
print(cert.)
      algorithm : {name:SHA256withECDSA, oid:1.2.840.10045.4.3.2}
          email : www.github.com
        encoded : -----BEGIN CERTIFICATE-----MIIEozCCBEmgAwIBAgIQT...
         issuer : {alt_names:{dNSName:www.github.com},
                  dn:CN=Sectigo ECC Domain Validation Secure Server CA, O=Sectigo
                     Limited, L=Salford, ST=Greater Manchester, C=GB,
                  x500:CN=Sectigo ECC Domain Validation Secure Server CA,O=Sectigo
                       Limited,L=Salford,ST=Greater Manchester,C=GB}
     not_after : 1741391999000
    not_before : 1709769600000
          oids : {critical:[2.5.29.15,2.5.29.19],
                  noncritical:[1.3.6.1.4.1.11129.2.4.2,
                  1.3.6.1.5.5.7.1.1,
                  2.5.29.14,
                  2.5.29.17,
                  2.5.29.32,
                  2.5.29.35,
                  2.5.29.37]}
  serial_number : 103892495973767669722220901035501109925
      signature : MEUCIQCu7Yxw-vR43BxY24MRjRr-sbNdF9Gub7pd9l5LOFhl...
        subject : {alt_names:{dNSName:www.github.com},
                  dn:CN=github.com, x500:CN=github.com}
        version : v3
```

Note that the OIDs are listed. Several of these are interpreted and returned as standard values.

Getting a stem of OIDs

```
   crypto#read_oid(cert., {'a':'2.5.29.14','b':'2.5.29.17', 'c':'2.5.29.32'})
{a:BBYEFDtoPzQ69Uc0yu-mTj2avV5uesyf,
 b:BB4wHIIKZ2l0aHViLmNvbYIOd3d3LmdpdGh1Yi5jb20,
 c:BEIwQDA0BgsrBgEEAbIxAQICBzAlMCMGCCsGAQUFBwIBFhdodHRwczovL3NlY3RpZ28uY2
   9tL0NQUzAIBgZngQwBAgE
}
```

QDL's Crypto Module

Each entry is returned with its value.

to_public

Description

Return the public part of a key

Usage

```
to_public(key. | keys.)
```

Arguments

`key.` - a single key stem

`keys.` - a stem of of key stems.

Output

Each key has its public parts returned. In the case of an symmetric key, the key itself is returned.

Example

This takes a single RSA key and returns the public part:

```
print(crypto#to_public(rsa.), 72)
alg : RS256
  e : AQAB
kid : CCB6D75A73F1F2F8
kty : RSA
  n : AJp4A-Q_uXMSvndxUJnD9sutUKVNu-A7i9V35V9yk7_NLCgEhsnbCenjXIiURENup5
      oRUrRjvNj_jyp-JNN0QTQO1eVJKZucIx-vt8mo9dpAADsQnwQtdp5yWV-dMSx4Tdqq
      udH5Og88EKMnO4AsUgqOZT28Cb0iz9NIxkvesnW9
use : sig
```

to_jwt

Description

Sign a stem as a JWT using an RSA or EC key. This turns a given stem into an RFC 7517 compliant JWT (JSON web token).

QDL's Crypto Module

Usage

`to_jwt(payload.) -` create an unsigned JWT

`to_jwt(payload., key.)` – create the JWT, also creating the header

`to_jwt(header., payload., key.)` – use the supplied header, adding only required information.

Arguments

`header.` – a stem of information about the way the payload is signed.

`payload.` – the stem. It will be turned into a JSON object then processed, so not every stem can be signed this was (e.g., JSON has no concept of a set, so set entries are converted to a list).

`key.` – the RSA or EC key to use.

Output

A JWT is of the form `header.payload.signature` unless it is unsigned, in which case it is of the form `header.payload.` (and the trailing period is required!). You can create an unsigned JWT also by supplying a header with the "alg" set to "none".

Example. An unsigned JWT

Create an unsigned JWT

```
   c := j_load('crypto');
   jwt := c#to_jwt{'a':{'b':[;3]}});
   jwt
eyJ0eXAiOiJKV1QiLCJhbGciOiJub25lIn0.eyJhIjp7ImIiOlswLDEsMl19fQ.
 decode(tokenize(jwt)\[0,1])
[{"typ":"JWT","alg":"none"},{"a":{"b":[0,1,2]}}]
```

Note that the output is JSON here. You would need to convert it to a stem or just use the **`from_jwt`** method

Example. A signed JWT

```
   p. :={'a':'q','b':{'s':'t'}};
   crypto :=j_load('crypto');
   rsa. :=crypto#create_key(2048);
   rr :=crypto#to_jwt(p., rsa.);
   rr
eyJraWQiOiI3REQ1RDJDMkJCMUE2MzdBIiwidHlwIjoiSldUIiwiYWxnIjoiUlMyNTYifQ.
EyJhIjoicSIsImIiOnsicyI6InQifX0.
Qqd9QotTC0...
```

QDL's Crypto Module

This uses an RSA key (which has to be at least 2048 bytes long).

If you prefer an example using an elliptic key

```
   p. ≔{a':'q','b':{'s':'t'}};
   crypto ≔_load('crypto');
   ec.≔ crypto#create_key({'type':'EC'});
   ss ≔ crypto#to_jwt(p., ec.);
   ss;
eyJraWQiOiI5RDRCOTM0QjEyMkY4QkFBIiwidHlwIjoiSldUIiwiYWxnIjoiRVMyNTYifQ.
EyJhIjoicSIsImIiOnsicyI6InQifX0.
4g7p54TXhWkxctB...
```

Note that the headers are different. In the case of the elliptic curve that is

```
   decode('eyJraWQiOiI5RDRCOTM0QjEyMkY4QkFBIiwidHlwIjoiSldUIiwiYWxnIjoiRVMyNTYifQ')
{"kid":"9D4B934B122F8BAA","typ":"JWT","alg":"ES256"}
```

Example. Sending a custom header

In this example, we'll send along a custom header that includes the time issued at. Just because, we'll how how to print the raw token, by chopping it up at each "." and decoding the result. The header and payload are in JSON, not stems! As expected, the header shows the **`iat`** (issued at time) claim and the payload is as expected. Note that the signature, which is simply an array of bytes, decodes as gibberish, so is not terribly interesting by itself. This does show all the parts of a JWT nicely though.

```
   ec_jwt := crypto#to_jwt({'iat':date_ms()%1000},p., ec.);
print(decode(tokenize(ec_jwt,'.')))
   0 : {"typ":"JWT","iat":1735561786.341000000000000,"alg":"ES256","kid":"9D4B934B122F8BAA"}
   1 : {"a":"q","b":{"s":"t"}}
   2 : n#]_W���;#0##0/0K#0=00r0"���D���P0mT*.00&0'[#0Fm@R?#0N=00#0
```

Note To get the issued at time in seconds, we used **%** and not division, since division would have given us a decimal in the header, not an integer.

verify

Description

Verify a JWT or aggregate (stem or set) of them against a key.

Usage

`verify(jwt | jwt., key.)`

QDL's Crypto Module

Arguments

`jwt` – A JWT

`jwt.` – a stem of JWTs

`key.` – the key that was used to sign them.

Output

A left conformable object that has **`true`** for each valid signature and **`false`** otherwise. Unrecognized arguments (such as a non-string or non-JWT string) return **`false`**.

Example

Bare bones to show that verify checks what signing does. The values for the key and payload are from the **`to_jwt`** function:

```
   crypto#verify(crypto#to_jwt(p., ec.), ec.);
true
```

Next, we make a list of JWTs and verify them:

```
   jwt. ≔[
          crypto#to_jwt(p.), // unsigned
          crypto#to_jwt(p., ec.), // basic
          crypto#to_jwt({'iat':date_ms()%1000}, p., ec.) // custom header
         ];
   crypto#verify(jwt., ec.);
[true,true,true]
```

Which shows that each entry of the list is checked. If we submitted non-JWTs each would be flagged as false. In this next example,we append a string and an integer. The contract for the function is to check if the arguments can be verified with the given key, hence the result

```
   crypto#verify(jwt.~'foo'~42, ec.);
[true,true,true,false,false]
```