Python OpenSSL Wrappers v0.5

Introduction

This is an early release for POW. The intention was to quickly cover the breadth of the OpenSSL library, then the depth of particular areas. Future releases will focus on bug fixes and filling missing gaps in the API. The digests and ciphers should be adequate for most purposes as should the SSL wrappers. Although certificates and CRLs can be generated using this library no extension support has been included in the current release.

The issue of how extensions should be supported is a high priority but is not clear how best to go about supporting this functionality. The ASN1 support in OpenSSL is due to be overhauled in 0.9.7 which should improve matters. Other possibilities would include using a tool like SNACC to augment OpenSSLs capabilities or handle these problems in with a standalone module. If you have any thoughts on this or other issues please contact the author.

Module Functions

Function Prototypes

```
def pemRead(string, type):
def seed(data):
def readRandomFile(filename):
def writeRandomFile(filename):
def getError():
def clearError():
def __doclist__():
```

Function Descriptions

The `pemRead` Function

This function attempts to parse the `string` according to the PEM type passed. `type` should be one of the following:

```
RSA_PUBLIC_KEY
RSA_PRIVATE_KEY
X509_CERTIFICATE
X509_CRL
```

The object returned will be and instance of `Asymmetric`, `X509` or `X509Crl`.

The `seed` Function

The `seed` function adds data to OpenSSLs PRNG state. It is often said the hardest part of cryptography is getting good random data, after all if you don't have good random data, a 1024 bit key is no better than a 512 bit key and neither would provide good protection from a targeted brute force attack.

The `readRandomFile` Function

This function reads a previously saved random state. It can be very useful to improve the quality of random data used by an application. The random data should be added to, using the seed function, with data from other suitable random sources.

The `writeRandomFile` Function

This function writes the current random state to a file. Clearly this function should be used in conjunction with `readRandomFile`.

The `getError` Function

Pops an error off the global error stack and returns it as a string.

The `clearError` Function

Removes all errors from the global error stack.

The `__doclist__` Function

This function returns a list of all the doc strings in this module. The doc strings contain a mixture of DocBook markup and custom tags which formally describe the class or function. The list of strings was used to generate this documentation, it was processed by a simple and pretty raw script which produced a valid DobBook article. Finally Openjade was used process the DocBook article to produce this document.

Module Classes

The `Ssl` Class

This class provides access to the Secure Socket Layer functionality of OpenSSL. It is designed to be a simple as possible to use and is not designed for high performance applications which handle many simultaneous connections. The original motivation for writing this library was to provide a security layer for network

agents written in Python, for this application, good performace with multiple concurrent connections is not an issue.

Performance issues will be addressed in later releases. Currently threaded applications are not supported, nor are SSL sessions.

Class Prototypes

```
class Ssl :
    def __init__(protocol):
    def setFd(descriptor):
    def accept():
    def connect():
    def write(string):
    def read(amount=1024):
    def peerCertificate():
    def useCertificate(cert):
    def useKey(key):
    def checkKey():
```

The `__init__` Method

This constructor creates a new `Ssl` object which will behave as a client or server, depending on the `protocol` value passed. The `protocol` also determines the protocol type and version and should be one of the following:

```
SSLV2_SERVER_METHOD
SSLV2_CLIENT_METHOD
SSLV2_METHOD
SSLV3_SERVER_METHOD
SSLV3_CLIENT_METHOD
SSLV3_METHOD
TLSV1_SERVER_METHOD
TLSV1_CLIENT_METHOD
TLSV1_METHOD
SSLV23_SERVER_METHOD
SSLV23_CLIENT_METHOD
SSLV23_METHOD
```

The `setFd` Method

This function is used to associate a file descriptor with a `Ssl` object. The file descriptor should belong to an open TCP connection. Once this function has been called, calling `useKey` or `useCertificate` will, fail rasing exceptions.

The `accept` Method

This function will attempt the SSL level accept with a client. The `Ssl` object must have been created using

a `XXXXX_SERVER_METHOD` or a `XXXXX_METHOD` and this function should only be called after `useKey`, `useCertificate` and `setFd` functions have been called.

Example 1. `accept` function usage

```
keyFile = open( 'test/private.key', 'r' )
certFile = open( 'test/cacert.pem', 'r' )

rsa = pow.pemRead( pow.RSA_PRIVATE_KEY, keyFile.read(), 'pass' )
x509 = pow.pemRead( pow.X509_CERTIFICATE, certFile.read() )

keyFile.close()
certFile.close()

sl = pow.Ssl( pow.SSLV23_SERVER_METHOD )
sl.useCertificate( x509 )
sl.useKey( rsa )

s = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
s.bind( ('localhost', 1111) )
s.listen(5)
s2, addr = s.accept()

s.close()

sl.setFd( s2.fileno() )
sl.accept()
print sl.read(1024)
sl.write('Message from server to client...')

s2.close()
```

The `connect` Method

This function will attempt the SSL level connection with a server. The `Ssl` object must have been created using a `XXXXX_CLIENT_METHOD` or a `XXXXX_METHOD` and this function should only be called after `setFd` has already been called.

Example 2. `connect` function usage

```
s = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
s.connect(('localhost', 1111))

sl = pow.Ssl( pow.SSLV23_CLIENT_METHOD )
sl.setFd( s.fileno() )
sl.connect()
sl.write('
```

```
Message from client to server...')
      print sl.read(1024)
```

The `write` Method

This method writes the `string` to the `Ssl` object, to be read by it's peer. This function is analogous to the `socket` classes `write` function.

The `read` Method

This method reads up to `amount` characters from the `Ssl` object. This function is analogous to the `socket` classes `read` function.

The `peerCertificate` Method

This method returns any peer certificate presented in the initial SSL negotiation or `None`. If a certificate is returned, it will be an instance of `X509`.

The `useCertificate` Method

The parameter `cert` must be an instance of the `X590` class. A second restriction, this function cannot be called after the file descriptor has been set.

The `useKey` Method

The parameter `key` must be an instance of the `Asymmetric` class and must contain the private key. A second restriction, this function cannot be called after the file descriptor has been set.

The `checkKey` Method

This simple method will return 1 if the public key, contained in the X509 certificate this `Ssl` instance is using, matches the private key this `Ssl` instance is using. Otherwise it will return 0.

The `X509` Class

This class provides access to a significant proportion of X509 functionality of OpenSSL.

Example 3. `x509` class usage

```
privateFile = open('test/private.key', 'r')
publicFile = open('test/public.key', 'r')
certFile = open('test/cacert.pem', 'w')
```

```
publicKey = pow.pemRead(pow.RSA_PUBLIC_KEY, publicFile.read())
privateKey = pow.pemRead(pow.RSA_PRIVATE_KEY, privateFile.read(), 'pass')

c = pow.X509()

name = [  ['C', 'GB'], ['ST', 'Hertfordshire'],
          ['O','The House'], ['CN', 'Peter Shannon'] ]

c.setIssuer( name )
c.setSubject( name )
c.setSerial(0)
c.setNotBefore( time.time() )
c.setNotAfter( time.time() + 60*60*24*365)
c.setPublicKey(publicKey)
c.sign(privateKey)

certFile.write( c.pemWrite() )

privateFile.close()
publicFile.close()
certFile.close()
```

Class Prototypes

```
class X509 :
    def __init__():
    def pemWrite():
    def sign(key):
    def setPublicKey(key):
    def getVersion():
    def setVersion(version):
    def getSerial():
    def setSerial(serial):
    def getIssuer(format=SHORTNAME_FORMAT):
    def setIssuer(name):
    def getSubject(format=SHORTNAME_FORMAT):
    def setSubject(name):
    def getNotBefore():
    def setNotBefore(time):
    def getNotAfter():
    def setNotAfter(time):
    def pprint():
```

The `__init__` Method

This constructor crates a skeletal X509 certificate object. It won't be any use at all until several stuctures have been created using it's member functions.

The `pemWrite` Method

This method returns a PEM encoded certificate as a string.

The `sign` Method

This method signs a certificate with a private key. See the example for the methods which should be invoked before signing a certificate. `key` should be an instance of `Asymmetric` contianing a private key.

The `setPublicKey` Method

This method sets the public key for this certificate object. The parameter `key` should be an instance of `Asymmetric` containing a public key.

The `getVersion` Method

This method returns the version number from the version field of this certificate.

The `setVersion` Method

This method sets the version number in the version field of this certificate. `version` should be an integer.

The `getSerial` Method

This method get the serial number in the serial field of this certificate.

The `setSerial` Method

This method sets the serial number in the serial field of this certificate. `serial` should ba an intger.

The `getIssuer` Method

This method returns a tuple containg the issuers name. Each element of the tuple is a tuple with 2 elements. The first tuple is an object name and the second is it's value. Both issuer and sybject names a distinguished names normally composed of a small number of objects:

```
c  or  countryName
st  or  stateOrProvinceName
o  or  organizationName
```

```
l  or  localityName
ou  or  organizationalUnitName
cn  or  commonName
```

The data type varies from one object to another, however, all the common objects are strings. It would be possible to specify any kind of object but that would certainly adversly effect portability and is not recomended.

The `setIssuer` Method

This method is used to set the issuers name. `name` can be comprised of lists or tuples in the format described in the `getissuer` method.

The `getSubject` Method

This method returns a tuple containg the subjects name. See `getIssuer` for a description of the returned object's format.

The `setSubject` Method

This method is used to set the subjects name. `name` can be comprised of lists or tuples in the format described in the `getIssuer` method.

The `getNotBefore` Method

This method returns a tuple containing two integers. The first number represents the time in seconds and is the same as the C `time_t` typedef and the second represents the time zone offset in seconds.

The `setNotBefore` Method

This method sets part of the `Validity` sequence of the certificate, the `notBefore` time. `time` should be a time in seconds, as generated by the `time` function in the Python Standard Library.

The `getNotAfter` Method

This method returns a tuple containing two integers. The first number represents the time in seconds and is the same as the C `time_t` typedef and the second represents the time zone offset in seconds.

The `setNotAfter` Method

This method sets part of the `Validity` sequence of the certificate, the `notAfter` time. `time` should be a time in seconds, as generated by the `time` function in the Python Standard Library.

The `pprint` Method

This method returns a formatted string showing the information held in the certificate.

The `X509Crl` Class

This class provides access to OpenSSL X509 CRL management facilities.

Class Prototypes

```
class X509Crl :
    def pemWrite():
    def getVersion():
    def setVersion(version):
    def getIssuer(format=SHORTNAME_FORMAT):
    def setIssuer(name):
    def getThisUpdate():
    def setThisUpdate(time):
    def getNextUpdate():
    def setNextUpdate(time):
    def getRevoked():
    def setRevoked(revoked):
    def verify(key):
    def sign(key):
    def pprint():
```

The `pemWrite` Method

This method returns a PEM encoded CRL as a string.

The `getVersion` Method

This method returns the version number from the version field of this CRL.

The `setVersion` Method

This method sets the version number in the version field of this CRL. `version` should be an integer.

The `getIssuer` Method

This method returns a tuple containg the issuers name. See the `getIssuer` method of `X509` for more details.

The `setIssuer` Method

This method is used to set the issuers name. `name` can be comprised of lists or tuples in the format described in the `getIssuer` method of `X509`.

The `getThisUpdate` Method

This method returns a tuple containing two integers. The first number represents the time in seconds and is the same as the C `time_t` typedef and the second represents the time zone offset in seconds.

The `setThisUpdate` Method

This method sets the `thisUpdate` field of this CRL. `time` should be a time in seconds, as generated by the `time` function in the Python Standard Library.

The `getNextUpdate` Method

This method returns a tuple containing two integers. The first number represents the time in seconds and is the same as the C `time_t` typedef and the second represents the time zone offset in seconds.

The `setNextUpdate` Method

This method sets the `thisUpdate` field of this CRL. `time` should be a time in seconds, as generated by the `time` function in the Python Standard Library.

The `getRevoked` Method

This method returns a tuple of `X509Revoked` objects described in the CRL.

Example 4. `getRevoked` function usage

```
publicFile = open('test/public.key', 'r')
crlFile = open('test/crl.pem', 'r')

publicKey = pow.pemRead(pow.RSA_PUBLIC_KEY, publicFile.read())

crl = pow.pemRead( pow.X509_CRL, crlFile.read() )

print crl.pprint()
if crl.verify( publicKey ):
   print 'signature ok!'
else:
   print 'signature not ok!'

revocations = crl.getRevoked()
for revoked in revocations:
```

```
    print 'serial number:', revoked.getSerial()
    print 'date:', time.ctime( revoked.getDate()[0] )

publicFile.close()
crlFile.close()
```

The `setRevoked` Method

This method sets the sequence of revoked certificates in this CRL. `revoked` should be a list or tuple of `X509Revoked`.

Example 5. `setRevoked` function usage

```
privateFile = open('test/private.key', 'r')
publicFile = open('test/public.key', 'r')
crlFile = open('test/crl.pem', 'w')

publicKey = pow.pemRead(pow.RSA_PUBLIC_KEY, publicFile.read())
privateKey = pow.pemRead(pow.RSA_PRIVATE_KEY, privateFile.read(), 'pass')

crl = pow.X509Crl()

name = [  ['C', 'GB'], ['ST', 'Hertfordshire'],
          ['O','The House'], ['CN', 'Peter Shannon'] ]

crl.setIssuer( name )
rev = [  pow.X509Revoked(3, int( time.time() ) - 24*60*60 ),
         pow.X509Revoked(4, int( time.time() ) - 24*60*60 ),
         pow.X509Revoked(5, int( time.time() ) - 24*60*60 )    ]

crl.setRevoked( rev )
crl.setThisUpdate( time.time() )
crl.setNextUpdate( time.time() + 2*60*60*24*365)
crl.sign(privateKey)

crlFile.write( crl.pemWrite() )

privateFile.close()
publicFile.close()
crlFile.close()
```

The `verify` Method

The `X509Crl` method `verify` is based on the `X509_CRL_verify` function. Unlike the `X509` function of the same name, this function simply checks the CRL was signed with the private key which corresponds the parameter `key`. `key` should be an instance of `Asymmetric` and contain a public key.

The `sign` Method

This method signs a CRL with a private key. `key` should be an instance of `Asymmetric` and contain a private key. See the `setRevoked` example.

The `pprint` Method

This method returns a formatted string showing the information held in the CRL.

The `X509Revoked` Class

This class provides a container for details of a revoked certificate. It normally would only be used in association with a CRL, its not much use by itself. Indeed the only reason this class exists is because in the future POW is likely to be extended to support extensions for certificates, CRLs and revocations. `X509Revoked` existing as an object in its own right will make adding this support easier, while avoiding backwards compatibility issues.

Class Prototypes

```
class X509Revoked :
    def __init__(serial, date):
    def getDate():
    def setDate(time):
    def getSerial():
    def setSerial(serial):
```

The `__init__` Method

This constructor builds a X509 Revoked structure. Both arguments are integers, `date` is the same as the C `time_t` typedef and can be generated with the Python Standard Library function `time`.

The `getDate` Method

This method returns a tuple containing two integers representing `revocationDate`. The first number represents the time in seconds and is the same as the C `time_t` typedef and the second represents the time zone offset in seconds.

The `setDate` Method

This method sets the `revocationDate` field of this object. `time` should be a time in seconds, as generated by the `time` function in the Python Standard Library.

The `getSerial` Method

This method get the serial number in the serial field of this object.

The `setSerial` Method

This method sets the serial number in the serial field of this object. `serial` should ba an intger.

The `X509Store` Class

This class provides preliminary access to OpenSSL X509 verfication facilities.

Example 6. `x509_store` class usage

```
store = pow.X509Store()

caFile = open( 'test/cacert.pem', 'r' )
ca = pow.pemRead( pow.X509_CERTIFICATE, caFile.read() )
caFile.close()

store.addTrust( ca )

certFile = open( 'test/foocom.cert', 'r' )
x509 = pow.pemRead( pow.X509_CERTIFICATE, certFile.read() )
certFile.close()

print x509.pprint()

if store.verify( x509 ):
   print 'Verified certificate!.'
else:
   print 'Failed to verify certificate!.'
```

Class Prototypes

```
class X509Store :
    def __init__():
    def verify(cert):
    def addTrust(cert):
    def addCrl(crl):
```

The `__init__` Method

This constructor takes no arguments. The `X509Store` returned cannot be used for verifying certificates until at least one trusted certificate has been added.

The `verify` Method

The `X509Store` method `verify` is based on the `X509_verify_cert`. It handles certain aspects of verification but not others. The certificate will be verified against `notBefore`, `notAfter` and trusted certificates. It crucially will not handle checking the certificate against CRLs. This functionality will probably make it into OpenSSL 0.9.7.

The `addTrust` Method

This method adds a new certificate to the store to be used in the verification process. `cert` should be an instance of `X509`. Using trusted certificates to manage verification is relativly primative, more sophisticated systems can be constructed at an application level by by constructing certifate chains to verify. This support will be added in the near future.

The `addCrl` Method

This method adds a CRL to a store to be used for verification. `crl` should be an instance of `X509Crl`. Unfortunately, the current stable release of OpenSSL does not support CRL checking for certificate verification. This functionality will probably make it into OpenSSL 0.9.7, until it does this function is useless and CRL verification must be implemented by the application.

The `Digest` Class

This class provides access to the digest functionality of OpenSSL. It emulates the digest modules in the Python Standard Library but does not currently support the `hexdigest` function.

Example 7. `digest` class usage

```
plain_text = 'Hello World!'
sha1 = pow.Digest( pow.SHA1_DIGEST )
sha1.update( plain_text )
print ' Plain text: Hello World! =>', sha1.digest()
```

Class Prototypes

```
class Digest :
    def __init__(type):
    def update(data):
    def copy():
```

```
def digest():
```

The `__init__` Method

This constructor creates a new `Digest` object. The parameter `type` specfies what kind of digest to create and should be one of the following:

```
MD2_DIGEST
MD5_DIGEST
SHA_DIGEST
SHA1_DIGEST
RIPEMD160_DIGEST
```

The `update` Method

This method updates the internal structures of the `Digest` object with `data`. `data` should be a string.

The `copy` Method

This method returns a copy of the `Digest` object.

The `digest` Method

This method returns the digest of all the data which has been processed. This function can be called at any time and will not effect the internal structure of the `digest` object.

The `Symmetric` Class

This class provides access to all the symmetric ciphers in OpenSSL. Initialisation of the cipher structures is performed late, only when `encryptInit` or `decryptInit` is called, the constructor only records the cipher type. It is possible to reuse the `Symmetric` objects by calling `encryptInit` or `decryptInit` again.

Example 8. `Symmetric` class usage

```
passphrase = 'my silly passphrase'
md5 = pow.Digest( pow.MD5_DIGEST )
md5.update( passphrase )
password = md5.digest()[:8]

plaintext = 'cast test message'
cast = pow.Symmetric( pow.CAST5_CFB )
cast.encryptInit( password )
ciphertext = cast.update(plaintext) + cast.final()
print 'Cipher text:', ciphertext

cast.decryptInit( password )
```

```
    out = cast.update( ciphertext ) + cast.final()
    print 'Deciphered text:', out
```

Class Prototypes

```
class Symmetric :
    def __init__(type):
    def encryptInit(key, initialvalue=''):
    def decryptInit(key, initialvalue=''):
    def update(data):
    def final(size=1024):
```

The `__init__` Method

This constructor creates a new `Symmetric` object. The parameter `type` specfies which kind of cipher to create. `type` should be one of the following:

```
DES_ECB         IDEA_CBC
DES_EDE         RC2_ECB
ES_EDE3         RC2_CBC
DES_CFB         RC2_40_CBC
DES_EDE_CFB     RC2_CFB
DES_EDE3_CFB    RC2_OFB
DES_OFB         BF_ECB
DES_EDE_OFB     BF_CBC
DES_EDE3_OFB    BF_CFB
DES_CBC         BF_OFB
DES_EDE_CBC     CAST5_ECB
DES_EDE3_CBC    CAST5_CBC
DESX_CBC        CAST5_CFB
RC4             CAST5_OFB
RC4_40          RC5_32_12_16_CBC
IDEA_ECB        RC5_32_12_16_CFB
IDEA_CFB        RC5_32_12_16_ECB
IDEA_OFB        RC5_32_12_16_OFB
```

Please note your version of OpenSSL might not have been compiled with all the ciphers listed above. If that is the case, which is very likely if you are using a stock binary, the unsorported ciphers will not even be in the module namespace.

The `encryptInit` Method

This method sets up the cipher object to start encrypting a stream of data. The first parameter is the key used to encrypt the data. The second, the `initialvalue` serves a similar purpose the the salt supplied to the Unix `crypt` function. The `initialvalue` is normally chosen at random and often transmitted with the encrypted data, its purpose is to prevent two identical plain texts resulting in two identical cipher texts.

The `decryptInit` Method

This method sets up the cipher object to start decrypting a stream of data. The first value must be the key used to encrypt the data. The second parameter is the `initialvalue` used to encrypt the data.

The `update` Method

This method is used to process the bulk of data being encrypted or decrypted by the cipher object. `data` should be a string.

The `final` Method

Most ciphers are block ciphers, that is they encrypt or decrypt a block of data at a time. Often the data being processed will not fill an entire block, this method processes these half-empty blocks. A string is returned of a maximum length `size`.

The `Asymmetric` Class

This class provides access to RSA asymmetric ciphers in OpenSSL. Other ciphers will probably be supported in the future but this is not considered a priority.

Class Prototypes

```
class Asymmetric :
    def __init__(keytype, keylength):
    def pemWrite(keytype, ciphertype=None, passphrase=None):
    def publicEncrypt(plaintext):
    def publicDecrypt(ciphertext):
    def privateEncrypt(plaintext):
    def privateDecrypt(ciphertext):
    def sign(digesttext, digesttype):
    def verify(signedtext, digesttext, digesttype):
```

The `__init__` Method

This constructor builds a new cipher object. Only RSA ciphers are currently support, so the first argument should always be `RSA_CIPHER`. The second argument, `keylength`, is normally 512, 768, 1024 or 2048. Key lengths as short as 512 bits are generally considered weak, and can be cracked by determined attackers without tremendous expense.

Example 9. `asymmetric` class usage

```
privateFile = open('test/private.key', 'w')
publicFile = open('test/public.key', 'w')
```

```
passphrase = 'my silly passphrase'
md5 = pow.Digest( pow.MD5_DIGEST )
md5.update( passphrase )
password = md5.digest()

rsa = pow.Asymmetric( pow.RSA_CIPHER, 1024 )
privateFile.write( rsa.pemWrite(
         pow.RSA_PRIVATE_KEY, pow.DES_EDE3_CFB, password ) )
publicFile.write( rsa.pemWrite( pow.RSA_PUBLIC_KEY ) )

privateFile.close()
publicFile.close()
```

The `pemWrite` Method

This method is used to write `Asymmetric` objects out to strings. The first argument should be either `RSA_PUBLIC_KEY` or `RSA_PRIVATE_KEY`. Private keys are often saved in encrypted files to offer extra security above access control mechanisms. If the `keytype` is `RSA_PRIVATE_KEY` a `ciphertype` and `passphrase` can also be specified. The `ciphertype` should be one of those listed in the `Symmetric` class section.

The `publicEncrypt` Method

This method is used to encrypt the `plaintext` using a public key. It should be noted; in practice this function would be used almost exclusivly to encrypt symmetric cipher keys and not data since asymmetric cipher operations are very slow.

The `publicDecrypt` Method

This method is used to decrypt the `ciphertext` which has been encrypted using the corresponding private key and the `privateEncrypt` function.

The `privateEncrypt` Method

This method is used to encrypt the `plaintext` using a private key. It should be noted; in practice this function would be used almost exclusivly to encrypt symmetric cipher keys and not data since asymmetric cipher operations are very slow.

The `privateDecrypt` Method

This method is used to decrypt ciphertext which has been encrypted using the corresponding public key and the `publicEncrypt` function.

The `sign` Method

This method is used to produce a signed digest text. This instance of `Asymmetric` should be a private key used for signing. The parameter `digesttext` should be a digest of the data to protect against alteration and finally `digesttype` should be one of the following:

```
MD2_DIGEST
MD5_DIGEST
SHA_DIGEST
SHA1_DIGEST
RIPEMD160_DIGEST
```

If the procedure was succesful, a string containing the signed digest is retuned.

The `verify` Method

This method is used to verify a signed digest text.

Example 10. `verify` method usage

```
plain_text = 'Hello World!'
print ' Plain text:', plain_text
digest = pow.Digest( pow.RIPEMD160_DIGEST )
digest.update( plain_text )
print ' Digest text:', digest.digest()

privateFile = open('test/private.key', 'r')
privateKey = pow.pemRead( pow.RSA_PRIVATE_KEY, privateFile.read(), 'pass' )
privateFile.close()
signed_text =  privateKey.sign(digest.digest(), pow.RIPEMD160_DIGEST)
print ' Signed text:', signed_text

digest2 = pow.Digest( pow.RIPEMD160_DIGEST )
digest2.update( plain_text )
publicFile = open('test/public.key', 'r')
publicKey = pow.pemRead( pow.RSA_PUBLIC_KEY, publicFile.read() )
publicFile.close()
if publicKey.verify( signed_text, digest2.digest(), pow.RIPEMD160_DIGEST ):
   print 'Signing verified!'
else:
   print 'Signing gone wrong!'
```

The parameter `signedtext` should be a signed digest text. This instance of `Asymmetric` should correspond to the private key used to sign the digest. The parameter `digesttext` should be a digest of the same data used to produce the `signedtext` and finally `digesttype` should be one of the following:

```
MD2_DIGEST
MD5_DIGEST
SHA_DIGEST
```

```
SHA1_DIGEST
RIPEMD160_DIGEST
```

If the procedure was succesful, 1 is returned, otherwise 0 is returned.