Pisces User Manual

Release 1.0 alpha 1

Jeremy Hylton

April 27, 2000

Corporation for National Research Initiatives
Reston, VA
E-mail: jeremy@alum.mit.edu

Abstract

Pisces is a Python implementation of the SPKI certificate standard. This document provides a brief introduction to SPKI (the Simple Public Key Infrastructure) and describes the tools available in the Pisces distribution. Pisces contains a library and command-line utility for creating and using SPKI certificates in Python. It also contains a toy secure socket library that demonstrates the use of the library.

Contents

Acknowledgements. Roger Masse and Guido van Rossum made helpful comments on library design. The HMAC implementation is based on an earlier version by Barry Warsaw. This work was supported in part by the Advanced Research Projects Agency of the United States Department of Defense under grant MDA972-95-1-0003.

1 Introduction

SPKI (Simple Public Key Infrastructure) is an experimental protocol developed by an IETF working group [Ell99]. This protocol defines public key certificate and signature formats, along with many associated objects, to support security for a wide range of applications. The SPKI protocol is simple to understand, use, and implement.

The CNRI SPKI library provides an implementation of the SPKI protocol for the Python language and a command-line utility, spkitool, for creating and using keys and certificates. This document primarily describes the utility.

Before going on to a detailed look at spkitool, we offer a brief introduction to the SPKI protocol based on the language in RFC 2693 [2]. For a more thorough discussion of SPKI, RFC 2693 itself, which lays out the basic theory underlying the protocol, and Carl Ellison's SPKI Web page, http://world.std.com/ cme/html/spki.html.

A certificate is a signed object that transfers some authority for the certificate's issuer to its subject. The main purpose of a SPKI certificate is to authorize the subject to perform some action. The subject and issuer are principals, i.e. a cryptographic key capable of generating a signature. The principals are represented by the key itself, a hash of the key, or a name that is bound to the key. A certificate that gives some authority to a key is, in effect, transferring some authority to the keyholder (anyone with access to the private key). The principal speaks for the keyholder by creating signed objects.

Unlike other public key infrastructures, SPKI principals are not named users, they are just keys. As it turns out, most authorization decisions are based not on the name of the keyholder, but on some attribute of the keyholder, e.g. whether she is a member of a particular organization, has paid some access fee, etc. A SPKI certificate can carry any authorizations or attributes the creator wishes; the specific interpretation is left to the applications that uses them.

A typical use of SPKI certificates is to implement a protected subsystem [3]. The root of each certificate's authority is an access control list (ACL).

The basic SPKI theory and protocol describe formats and uses for keys, certificates, and many associated objects. They do not describe protocols or APIs for transferring certificates between programs or storing them on disk. The implementation of Pisces SPKI library makes a number of concrete decisions about issues like these.

2 An Example

The script example.py in the scripts directory shows how you could use Pisces to perform access control on an object. The example is a bank account object, with methods deposit, withdraw, checkBalance, and setBalance.

The example scenario involves several different keyholders. This scenario isn't meant to be an exact model

of the real world or a suggestion for how to model a bank's trust relationships; rather, it demonstrates the various ways in which keys and certificates can be used. The Pisces distribution contains keys and certificates in the test/example directory. The text below explains the commands used to create them.

The bank has a key that is on the access control list for bank accounts; it is also used to delegate permission to bank employees and account holders. The bank grant the following permissions to keyholders:

The auditor has permission to check the balance of the account. The adjuster has permission to set the balance of the account. The account holder has permission to check the balance of the account and deposit and withdraw money.

The account holder delegates her permissions to these keyholders:

The spouse has all the permissions that the account holder has. The employer has permissions to deposit money.

The permissions are represent as a SPKI tag set.

The following commands show how to create each of the keys described above. We start by creating a default key. The other keys will each be issued with names relative to the default key. Also note that the **--unsafe** argument is used.

```
spkitool.py -d ../test/example create -b 512 --unsafe --default
spkitool.py -d ../test/example create -b 512 --unsafe bank
spkitool.py -d ../test/example create -b 512 --unsafe auditor
spkitool.py -d ../test/example create -b 512 --unsafe adjuster
spkitool.py -d ../test/example create -b 512 --unsafe account_holder
spkitool.py -d ../test/example create -b 512 --unsafe spouse
spkitool.py -d ../test/example create -b 512 --unsafe employer
```

Now that all the keys are created, the list command will show the hashes for each of them.

```
spkitool.py -d ../test/example list
PRIVATE KEYS
(hash md5 |kcMoJXXqOptLRRZfy4iWqA==|) default
(hash md5 |j5r4iybdTiCY5W/OEMVykQ==|)
(hash md5 |C7ZoGtjRIpwpiMj3CrwFBw==|)
(hash md5 |+39QQ2EKpFCcJvM3X3W2tA==|)
(hash md5 |1NECGruVdAzXN5cpsHOBdw==|)
(hash md5 |I5kaokdnKniSiXLoUh2y/A==|)
(hash md5 |UUWuzINnGXAxIl4H1OLFpw==|)

NAMES
Names issued by key (hash md5 |kcMoJXXqOptLRRZfy4iWqA==|)
"auditor": (hash md5 |j5r4iybdTiCY5W/OEMVykQ==|)
"employer": (hash md5 |+39QQ2EKpFCcJvM3X3W2tA==|)
"adjuster": (hash md5 |1NECGruVdAzXN5cpsHOBdw==|)
"bank": (hash md5 |I5kaokdnKniSiXLoUh2y/A==|)
"spouse": (hash md5 |UUWuzINnGXAxIl4H1OLFpw==|)
"account_holder": (hash md5 |C7ZoGtjRIpwpiMj3CrwFBw==|)

PUBLIC KEYS
(hash md5 |j5r4iybdTiCY5W/OEMVykQ==|)
(hash md5 |kcMoJXXqOptLRRZfy4iWqA==|)
(hash md5 |C7ZoGtjRIpwpiMj3CrwFBw==|)
(hash md5 |+39QQ2EKpFCcJvM3X3W2tA==|)
(hash md5 |1NECGruVdAzXN5cpsHOBdw==|)
(hash md5 |I5kaokdnKniSiXLoUh2y/A==|)
(hash md5 |UUWuzINnGXAxIl4H1OLFpw==|)
```

Now we need to create a collection of certificates that authorize the keyholders to act on the bank account. We'll start with the access control list for the account:

```
spkitool.py -d ../test/example acl --subject bank -p '(*)' --db \
    ../test/example/acl
```

The access control list, which is newly created, looks like this:

```
debugdb.py ../test/example/acl
../test/example/acl
(entry
   (hash md5 |I5kaokdnKniSiXLoUh2y/A==|)
   (tag
      (*)))
```

```
spkitool.py -d ../test/example cert --issuer bank --subject auditor \
    --after now --permission '(* set checkBalance)'
spkitool.py -d ../test/example cert --issuer bank --subject adjuster \
    --after now --permission '(* set setBalance)'
spkitool.py -d ../test/example cert --issuer bank --subject account_holder \
    --after now --before 2002-04-01_00:00:00 \
    --permission '(* set checkBalance deposit withdraw)'
spkitool.py -d ../test/example cert --issuer account_holder \
    --subject spouse --after now --permission '(*)'
spkitool.py -d ../test/example cert --issuer account_holder \
    --subject employer --after now --permission '(* set deposit)'
```

Now all the permissions should be in place. The test/example directory contains a pickled bank account object in account.pyp. The example.py script has rather complicated calling conventions. There is a lot of state necessary for checking permissions – the ACL, the certs and keys, and the key of the caller; each of these items is passed on the command line. The principal making the caller is specified with the **-p name** option and the method being invoked is specified with the **-m method_name** option.

```
python example.py -d ../test/example -o ../test/example/account.pyp \
    -a ../test/example/acl -k ../test/example/keys \
    -p account_holder -m checkBalance
python example.py -d ../test/example -o ../test/example/account.pyp \
    -a ../test/example/acl -k ../test/example/keys \
    -p adjuster -m checkBalance
python example.py -d ../test/example -o ../test/example/account.pyp \
    -a ../test/example/acl -k ../test/example/keys \
    -p employer -m deposit 100
```

3 Command-Line Utility: spkitool.py

The spkitool.py utility supports several different commands. The specific command to run is passed as an argument to spkitool.py. A typical usage would look like this:

```
% spkitool.py [generic options] command [command options]
```

The options and commands are described in the next section.

It may be helpful to think of spkitool.py as an analog to the GPG or PGP command-line tools. It manages

a collection of keys and certificates stored on your local system.

You need to create a directory that spkitool.py will use to store keys and certificates. The default directory is /.spki, but you can change this by setting the $SPKIHOME environment variable or using the generic -d option to spkitool.py.

Many of the spkitool commands require the use of a public-private key pair. You should create a default key pair the first time you use spkitool. The default key will be used for all commands that require a key, unless you specify a different key. The easiest way to specify a key is with a SPKI name, which can be created with the name command described below.

3.1 spkitool.py command reference

`spkitool.py` [`-v`] [`-h` [***cmd***]] [`-d` ***dir***] ***command*** [`command-options`]

This script can be invoked to run any of the commands listed below. It supports several generic options that must be listed before the command. Each command also supports command-specific options, which must be listed after the command name.

The **-v** option causes verbose output while the command runs.

The **-h** option prints help. If **-h** is specified by itself, a list of all commands is printed. If **-h** is followed by a command name, detailed help for that command is printed. The name 'all' is used as an argument, detailed help is printed for each command.

The **-d** ***dir*** specifies the location of the user's spkitool configuration directory.

The spkitool script supports the following commands:

`acl -s/--subject` ***key*** `-p` ***permissions*** [`-o output`] [`-b/--before` ***time***] [`-a/--after` ***time***] [`--test URI`] [`-d/--delegate`] [`--db` ***acl***]

Creates a certificate for an access control list (Entry). The subject can be a hash of a key or a name. If the subject is a name, the name is interpreted relative to the user's default key.

The permissions should be a text representation of a SPKI sexp. This is a little clunky, but it's hard to come up with a general interface for something that is essentially application-specific.

If the **--db** option is used, the Entry is added to the database.ACL file at the specified path. If this option is used, it overrides the **-o** option. Will create a new ACL if one does not exist.

The **-d**/**--delegate** option allows permissions to be delegated. By default, delegation is disabled.

The before/after/test modifiers are the same as for the cert command. They affect the validity constraints.

`cert -s/--subject -p/--permission permission` [`-i/--issuer`] [`-b/--before time`] [`-a/--after time`] [`-t/--test URI`] [`-d/--delegate`] [`-k/--key`]

Create a new certificate. The certificate has the following parts: subject, issuer, validity, and permissions. It may also have a delegation tag. Each of these parts can be specified using a different option; each option has a short name and a long name.

Note: To create a name certificate, use the name command.

The options for the cert command are list below. Each take a single argument following the option name.

optional: **--issuer** (**-i**)

The hash or name of the key to use as issuer. Will use the default key as issuer otherwise.

required: **--subject** (**-s**)

The hash or name of the key that is the subject of the certificate.

optional: **--before** (**-b**)
optional: **--after** (**-a**)

These options limit the period of time for which the certificate is valid. The time format is YYYY-MM-DD_HH:MM:SS. You can also use the string "now" to indicate the current time.

optional: **--test** (**-t**)

Specify an online validity test for the certificate. The argument should be the URL for the test. This optional currently has no associated implementation; although it can be included in the certificate, the test will not be performed.

required: **--permission** (**-p**)

Specify the permissions that are being granted to the subject key. The argument must be an S-expression in human-readable form for the permissions. The permissions will be wrapped in a (tag ...) S-expression.

optional: **--delegate** (**-d**)

Allow subject to delegate permission. Default is to disallow delegation.

optional: **--key** (**-k**)

Include the key of the issuer with the certificate. By default, the certificate does not include the key itself, only the hash of the key.

`create [-b bits] [--unsafe] [--replace] [--dup] [--default|name]`

Create a key pair with label as specified by user. Add the key pair to the user's key file and add the public key to the user's ACL. The user will be prompted for a pass phrase to encrypt the private key. Currently on the rsa-pkcs1-md5 algorithm is supported.

Options:

--unsafe: use the user's PID as pass phrase

--replace: replace an existing default key

-b NNNN: number of bits for key pair (default 1024)

--dup: create a name cert even if name is already used

Note: The SPKI protocol allows multiple keys to be bound to the same name, which allows the creation of a group. But multiple keys with the same name presents a problem for naming private keys, because the name no longer unambiguously refers to a single key.

`export [-o output] [--canonical] <hash-or-name>`

Export a public key from the store. The key may be specified by its hash or its name. The **-o** flag can be used to specify a file to place the key; if no file is specified, the key will be printed on stdout.

Keys are output in base64 encoding by default. Use the **--canonical** flag to specify the canonical (binary) encoding instead.

`import [path]`

Load a new public key or certificate into the keystore. The object will be loaded from a file containing either the canonical or base64 encoding of the S-expression. If no path is specified, the object will be read from stdin.

For a certificate to be useful in verifying a certificate chain, it must be signed. A public key, however, needs no signature.

`list [--public] [--private] [--name]`

List all the keys stored in the KeyStore and all the name certificates issued by the private keys. For private keys, the hash of the corresponding public key is listed. As a result, the same hash should appear in the private and public lists.

Specifying the options, limits the listing to only the specified sections.

`name -n/--name name -h/--hash hash [-i/--issuer principal] [-o output]`

Create a name certificate. Can be used to associate a name with a public key on the key server.

The hash designates the public key that the name will be bound to. The hash can either by the advanced form of a SPKI hash object, or just the base64 encoded digest. Thus, either of the following is allowed: '(hash md5 —hTK6mv8Nbspy9jsljfb2DQ==—)' or 'hTK6mv8Nbspy9jsljfb2DQ=='.

By default, the issuer is the user's default key. To use a different key, the user should specify the name or hash of the key with the **--issuer** argument.

The **-o** option can be used to make a local copy of the name cert.

This command needs to be extended with validity handling.

`show` [`-i input`] [`-o output`]
Read in an arbitrary SPKI object and display it in human-readable form. The show command will read the object from stdin and display it on stdout. The source and destination can be changed with the **-i** and **-o** arguments.

`sign` [`-s/--signer name`] [`-o output`] `file`
Create a digital signature for the contents of the specified file. By default, the signature is placed stored as file.sig, but you can use the **-o** option to specify a different output location.

The signature is created using the default key, unless the **--signer** option is used to specify a different key. The **--signer** option will accept the name or hash of a key.

`verify` [`-i signaturefile`] `file`
Verify a digital signature of a file. The default behavior is to look for the signature in file.sig. The **-i** option can be used to specify a different path for the signature.

4 Other command-line scripts

4.1 pyarrowd.py command reference

`pyarrowd.py` [`-h host`] [`-p port`]
A Yarrow daemon, only supported on Solaris and Linux so far.

Simple protocol for requesting random bytes over a socket. The client requests random data by sending a 32-bit int in network byte-order. The server will return that many bytes of random data. The return format is a 32-bit in in network byte order, specifying length of return value, followed by that many bytes of data. The client can issue multiple requests on a single socket.

The server runs on port 12000 by default. The specific host and port can be set with the **-h host** and **-p port** options, respectively.

The Yarrow daemon uses a few system utilities that should produce data that is somewhat hard to predict. There has been little effort put into choosing the utilities or estimating the entropy they produce. These utilities are determined by the `fast_sources` and `slow_sources` global variables, which are only defined for Linux and Solaris.

5 Pisces Library Reference

5.1 `pisces.algid` – Constants for PKCS #1 AlgorithmIdentifiers

This module implements the `AlgorithmIdentifer` objects required to implement PKCS #1. It defines one class, `AlgorithmIdentifier`, and several instances that are used as constants in other modules.

`AlgorithmIdentifier(`[*obj,* [*params*]]`)`
`AlgorithmIdentifier` is a subclass of `pisces.asn1.ASN1Object` that represents the AlgorithmIdentifier structure defined by PKCS #1 and #7. In addition to the methods defined by `ASN1Object`,

it has three public attributes: `oid`, the `pisces.asn1.OID` of the algorithm, `params`, the parameters optionally defined for the algorithm, and `name`, the name of algorithm. `params` and `name` may be None.

The constructor can be called two ways. It can be called with a single sequence that matches the following ASN.1 defintion:

```
AlgorithmIdentifier  ::=  SEQUENCE  {
    algorithm               OBJECT IDENTIFIER,
    parameters              ANY DEFINED BY algorithm OPTIONAL}
```

It can also be called with the *algorithm* and *parameters* components as arguments. *algorithm* must an OID. If *parameters* is omitted, it is treated as `None`.

The module defines the attributes listed in the table below, each of which is an instance of `pisces.asn1.OID`.

attribute	OID	name
oid_dsa	1.2.840.10040.4.1	dsa
oid_dsa_sha1	1.2.840.10040.4.3	dsaWithSha1
oid_rsa	1.2.840.113549.1.1.1	rsa
oid_rsa_md2	1.2.840.113549.1.1.2	md2withRSAEncryption
oid_rsa_md5	1.2.840.113549.1.1.4	md5withRSAEncryption
oid_md2	1.2.840.113549.2.2	md2
oid_md5	1.2.840.113549.2.5	md5
oid_sha	1.3.14.3.2.26	sha

5.2 `pisces.asn1` – Encode and decode ASN.1 BER format

This module provides a parser for ASN.1 objects encoded using BER. The parser produces `ASN1Object` instances that can be converted back to BER encoding using the `encode` method.

`parse(`*buf*`)`
: Parse string *buf* and return an `ASN1Object` instance that it encodes. Raises `ValueError` if invalid data is passed to it. *Warning:* This code could be more robust; other exceptions may be raised for particular invalid inputs.

`ASN1Object(`*val*`)`
: The `ASN1Object` class is the abstract base class of all the objects generated by `parse`. The constructor takes a single argument *val*, a parsed ASN.1 object.

  Subclasses of `ASN1Object` define the following method and attribute:

  `atomic`: True if the object is atomic; false if it is a container.

  `encode(`$\left[io\right]$`)`
  : Returns a string containing the BER encoding of the ASN.1 object. If a file-like object is passed as *io*, the encoding will be written to the file instead.

The following subclasses of `ASN1Object` are defined in this module. Methods defined only for single subclasses are also described here.

`Sequence()`
: `Sequence` implements the ASN.1 type SEQUENCE, an ordered collection of one or more types. Instances can are also Python sequence objects.

`Set()`
: `Set` implements the ASN.1 type SET, an unorderd collection of one or more types. Instances can are also Python sequence objects, where the order depends on the specific order of elements in the original encoding.

`UTCTime()`

`UTCTime` implements the standard ASN.1 type for time expressed in GMT. The X.509 standards note that UTCTime values shall be expressed Greenwich Mean Time (Zulu) and shall include seconds (i.e., times are YYMMDDHHMMSSZ), even where the number of seconds is zero. Conforming systems shall interpret the year field (YY) as follows:

Where YY is greater than or equal to 50, the year shall be inter- preted as 19YY; and

Where YY is less than 50, the year shall be interpreted as 20YY.

`UTCTime` instances can be compared to each other and support ordering in the natural way.

`Contextual()`

`Contextual` is a wrapper for ASN.1 types defined using CHOICE. For contextual encoding, it isn't possible to tell what the type of the contained object is without looking at the ASN.1 type declaration. This module is designed for parsing independently of the type declaration, which works for every case exception this one.

In the case of contextual encoding, this object is returned. When the decoded structure is actually used, it should be clear whether this is, e.g., an OPTIONAL integer type, some other tagged, known type, or an encoded CHOICE. Clients should call the decode method when the encoding includes the full DER encoding. Clients should call choose when the value doesn't have the appropriate tag/len info.

`Contextual` has two extra methods:

`decode()`

Return the decoded object. This method should be called when the encoding includes the tag and length.

`choose(`*tag*`)`

Return the decoded object, using *tag* as the ASN.1 tag. This method should be called when the containing type declaration uses CHOICE and the tag is not included in the encoding.

`Boolean()`

`Boolean` is the ASN.1 Boolean type. Instances of this class implement `__nonzero__`, so they can be used in Python conditionals.

`OID()`

An ASN.1 Object Identifier. Can be compared to other `OID` objects and used as a dictionary key.

Other ASN.1 types are represented using Python's builtin datatypes. This table summarizes the ASN.1 types supported.

ASN.1 type	**Python type**
INTEGER	int
BIT_STRING	string
OCTET_STRING	string
NULL	None
PrintableString	string
T61String	string
IA5String	string

`display(`*obj*`)`

Pretty-print an `ASN1Object` instance.

`Displayer(`$\left[$*oids*$\right]$`)`

The `Displayer` class creates pretty-printers for ASN.1 objects that will print labels for certain well-known oids. The constructor argument *oids* is a mapping from `OID` object to string names.

`parseCfg(`*io*`)`

Reads a configuration file from *io*, which must be a file-like object ready for reading. Returns a

dictionary mapping `OIDs` to strings that describe the `OIDs`. This function is provided for compatibility with Peter Gutmann's **dumpasn1** program. An example of the configuration file is available at http://www.cs.auckland.ac.nz/ pgut001/dumpasn1.cfg.

5.3 `pisces.cryptrand` – Interface to cryptographic random number generators

This module provides a common interface to the various cryptographic random numbers that may be available on a particular system. The module defines one function `random` and one constant `implementation`, which is a string that describes the implementation used for `random`.

`random(`*num*`)`
: Return *num* bytes of random data.

The implementation depends on the random number generators available on a particular system. On Linux systems, /dev/random is used. This module can also use the Yarrow daemon provided with Pisces or the Entropy Gathering Daemon from gpg if it is installed in ' /.gnupg/entropy.

WARNING: If no cryptographic random number generator is available, Python's `whrandom` implementation will be used. The is *not* a cryptographic random number generator.

5.4 `pisces.egdlib` – Interface to the Entropy Gathering Daemon

EGD (Entropy Gathering Daemon) is a tool designed to be used by GPG (GNU Privacy Guard) on systems that do not have a /dev/random. It is a user-space program that culls entropy from system statistics reported by various commands like **w** or **vmstat**. EGD is available from http://www.lothar.com/tech/crypto/.

This module implements an interface for request random data from EGD. For most purposes, it is better to use the interface provided by `pisces.cryptrand` which provides an abstraction layer on top of this and other modules.

`EGD(`*path*`)`
: The `EGD` class provides a high-level interface for communicating with an EGD daemon. The constructor takes that path of the Unix-domain socket used by the daemon.

  `EGD` defines the following methods:

  `getAvailableEntropy()`
  : Returns the number of bits of entropy currently available.

  `getRandomBytes(`*num*`)`
  : Returns up to *num* bytes of random data. It will return less than *num* bytes if sufficient entropy is not available.

  `getRandomBytesSync(`*num*`)`
  : Returns exactly *num* bytes of random data. *WARNING:* This method does not appear to behave correctly, perhaps because of a bug in EGD.

  `getPID()`
  : Returns the daemon's process id.

5.5 `pisces.hmac` – Keyed-hashing for message authentication

This module implements HMAC, a method for message authentication using cryptographic hash functions described in RFC 2104. It is a slight improvement to an earlier version written by Barry Warsaw.

`HMAC(`*hashmodule*`)`
: Instances of the `HMAC` class implement HMAC for a specific hash function. The constructor takes the

hashmodule as an argument. It must be a module that follows conforms to the interface of the hashes in `Crypto.Hash`. This requires that the module have a `digestsize` attribute and a `new` function.

`HMAC` provides the following method:

`hash`(*key, block*)
: Produce the HMAC hash for the given string, *block*. *Key* is the shared secret authentication key, as a string. For best results RFC 2104 recommends that the length of key should be at least as large as the underlying hash's output block size, but this is not enforced.

  If the key length is greater than the hash algorithm's basic compression function's block size (typically 64 bytes), then it is hashed to get the used key value. If it is less than this block size, it is padded by appending enough zero bytes to the key.

`HMACSpecializer`(*hashmodule, key*)
: Instances of the `HMACSpecializer` class implement an HMAC for a particular hash module and key. If many HMACs will be computed with the same key, it is more efficient to use this class than `HMAC`. There is no other reason to use this class.

  `HMACSpecializer` provides the following method:

  `hash`(*block*)
  : Produce the HMAC hash for the given string, *block*. See `HMAC.hash` for details.

5.6 `pisces.pkcs1` – A PKCS #1 wrapper for Crypto.PublicKey.RSA keys

This module implements that PKCS #1 RSA encryption standard. It must be used in conjunction with `Crypto.PublicKey.RSA`, which provides the cryptographic primitives.

The PKCS #1 standard is available from RSA Labs. As of April 10, 2000 the url is http://www.rsasecurity.com/rsalabs/pkcs/pkcs-1/.

This module defines several classes. The primary interfaces are the `RSA_pkcs1` class and `getSignatureImpl()` function, which returns an appropriate subclass of `DigestWithRSA`.

`RSA_pkcs1`(*key*)
: The `RSA_pkcs1` class is a wrapper for `Crypto.PublicKey.RSA` key objects that implements the PKCS #1 standard. Its encryption and decryption methods handle objects that are properly padded and encoded for interchange with other PKCS #1 implementations.

  The constructor accepts either a key object generated by `Crypto.PublicKey.RSA` or a tuple of key components that can be used to construct one.

  `RSA_pkcs1` defines the following methods:

  `getPublicComponents()`
  : Returns the public components of the key, e and n.

  `getPrivateComponents()`
  : Returns the public components of the key, d, p and q.

  `encryptPublic`(*plain*)
  : Returns the plaintext *plain* encrypted with the public key. Raises `ValueError` if the plaintext is too long for the key.

  `decryptPublic`(*cipher*)
  : Returns the plaintext obtained by decrypting *cipher* with the public key. Raises `ValueError` if the ciphertext is too long for the key.

  `encryptPrivate`(*plain*)
  : Returns the plaintext *plain* encrypted with the private key. Raises `ValueError` if the plaintext is too long for the key.

`decryptPrivate(`*cipher*`)`
: Returns the plaintext obtained by decrypting *cipher* with the private key. Raises `ValueError` if the ciphertext is too long for the key.

`DigestWithRSA(`*key*`)`
: The `DigestWithRSA` is an abstract base class that defines `sign` and `verify` methods that perform digital signature operations as defined by PKCS #1. Subclasses of `DigestWithRSA` implement a `digest` method that is used to generate the approriate message digest of the signed object.

  Subclasses must also define two attributes that identify the hash algorithm: `_digAlgId`, a `pisces.algid.AlgorithmIdentifier`, and `oid`, a `pisces.asn1.OID`.

  The constructor takes an `RSA_pkcs1` instance.

  `DigestWithRSA` defines the following methods:

  `sign(`*data*`)`
  : Returns a string representing the signature of *data*. Internally, encrypts a digest of *data* with the private key.

  `verify(`*data, sig*`)`
  : Verify that the signature *sig* matches the original string *data*. Returns 1 if the signature is correct and 0 if it is not. Raises a `ValueError` if the hash algorithm used with the signature does not match the hash algorithm the instance supports.

  `digest(`*data*`)`
  : Returns a digest of *data* using the hash function defined for the instance. Note that this method is defined in subclasses of `DigestWithRSA`.

`MD5withRSA(`*key*`)`
: A subclass of `DigestWithRSA` that supports the MD5 hash algorithm.

`MD2withRSA(`*key*`)`
: A subclass of `DigestWithRSA` that supports the MD2 hash algorithm.

`getSignatureImpl(`*algorithmId*`)`
: Returns a subclass of `DigestWithRSA` that supports the hash algorithm described by *algorithmId*, which must be an instance of `pisces.asn1.OID`. Currently, MD2 and MD5 are the only supported hash algorithms.

5.7 `pisces.pwcrypt` – Support for password-based cryptography (PKCS #5)

This module supports the use of password-based cryptography for encryption and message authentication using key derivation functions. This module is based on recommendations in PKCS #5 v2.0: Password-Based Crypotgraphy, RSA Laboratories, March 25, 1999. The recommendations are available from http://www.rsasecurity.com/rsalabs/pkcs/pkcs-5/index.html.

WARNING: It is not very practical to use Python for generating keys from passphrases; it is merely convenient. The key derivation process should take a long time, to thwart an attacker who attempts a dictionary attack on the password. But it can't take so long that the user grows impatient waiting for the key to be generated. The attacker could implement her brute force search in optimized C, which would be much faster than this Python implementation. Thus, this module provides much less security-for-the-wait that an optimized C version would.

`KeyDerivationFactory(`*keylen, saltlen,* $\left[\right.$*iterations*$\left.\right]$, $\left[\right.$*hash*$\left.\right]$, $\left[\right.$*labels*$\left.\right]$`)`
: A `KeyDerivationFactory` instance will generate keys that are *keylen* bytes long with *saltlen* bytes of salt. The optional arguments specify: the number of *iterations* of the the F function, the default value is 1000; the *hash* function to use, the default is SHA. The *hash* argument must support the interface implemented by `Crypto.Hash` hashes.

Labels are an optional feature. The *labels* argument accepts a sequence of strings. If several keys with the same generation parameters are going to be created, the salt should contain some text that identifies the particular use of the key. These are the labels. When `createKey` is called, it will check to see if the label used is valid.

The design of this class is explained carefully in the PKCS #5 document. The implementation uses HMAC plus a hash function as its pseudorandom function. The default hash is SHA.

`createKey`(*password* [, *label*])
: Create a new key generated from the string *password* and optionally the string *label*. Returns the salt, the number of iterations of the F function, the name of the hash function, and the key itself. Raises `ValueError` if *label* is specified and does not match one of the labels specified in the constructor.

`recreateKey`(*password, salt*)
: Creates a key generated from the string *password* and the explicitly supplied *salt* string. Returns only the key. This method should only be used to recreate a key previously generated by `createKey`.

5.8 `pisces.utils` – Internal pisces utility functions

`loadModule`(*name*)
: Import the module *name* and return it. If *name* is a module within a package, return the final module object and not the package.

`xor`(*s1, s2*)
: Return the bytewise-exclusive-or of two strings.

5.9 `pisces.yarrow` – Yarrow random number generator

This module implements a Yarrow-160 cryptographic pseudorandom number generator for Unix. It is based on a design described in the following paper: John Kelsey, Bruce Schneier, and Niels Ferguson: "Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator," http://www.counterpane.com/yarrow.html. The documentation here assumes you are familiar with the design described there.

Counterpane also provides an implementation, Yarrow 0.8.71, for Windows developed by Ari Benbasat. The implementation appears to diverge from the design document in numerous respects, e.g. entropy estimation and generating pseudorandom ouputs from the pool values.

A note on the implementation: The Yarrow design paper uses 3DES in counter mode. Counter mode is fairly unusual. It is not implemented in the Python Crypto Toolkit or OpenSSL; it only merits a one-paragraph mention in Schneier's book Applied Cryptography. The implementation here uses 3DES in ECB mode with a counter. The counter is encrypted with the key, then XORed with the plaintext.

`EntroypSource`()
: `EntropySource` instances track the amount of entropy available from a single source. It is used internally by `EntropyPool`.

  The design document proposes three different methods for Entropy Estimation: It estimates the entropy using three different methods and returns the lowest result.

  This implementation, following Benbasat, uses two estimates: one provided by the programming who supplies the input and another generator from zlib.

  `addInput`(*buf, estbits*)
  : Update the estimate to account for the entropy in *buf* with *estbits* bits of entropy. Both the actual data and the user-supplied estimare are necessary to update the internal entropy estimate.

`getEntropy()`
: Return the minimum estimated entropy currently available.

`reset()`
: Reset the estimates to zero.

`EntroypPool(`*threshold, count*`)`
: The `EntropyPool` collects samples from entropy sources. The design document described an Entropy Accumulator, which collects samples from entropy sources and puts them into two pools. This class implements the pools.

  A pool contains the running hash of all inputs fed into it since the `accumulate` method was called. The `accumulate` method is called by the `Yarrow` class during reseed operations.

  The pool keeps estimates about the entropy of each individual source, although the digest is over all sources. Each souce must be initialized by calling `addSouce` and passing the source's name. The instance variable sources maps from names to `EntropySource` instances.

  The constructor takes two arguments, the *threshold* and a *count*. A pool is ready to be used when at least *count* of its sources have an entropy greater than or equal to *threshold*. The `isReady` method returns true when this condition is met.

  `EntropyPool` instances are thread safe, because a typical use is to have multiple threads adding entropy to the pool.

  `addSource(`*name*`)`
  : Prepare pool for accepting input from source named *source*.

  `addInput(`*source, buf, estbits*`)`
  : Update hash of pool *source* with *buf* containing *estbits* estimated entropy.

    If the source was not initialized via `addSource`, this method will raise a `KeyError`.

  `isReady()`
  : Returns true if there is enough entropy in the pool. Enough is defined by the threshold and count arguments to the constructor.

  `accumulate()`
  : Return the current pool digest and reset the pool.

`Yarrow()`
: The `Yarrow` class generates random data and managed the fast and slow entropy pools for seeding the PRNG. These functions are described as three seperate entities in the design document: "generating pseudorandom outputs," "reseed mechanism," and "reseed control."

  The main API for this class is three methods: `getOutput`, `addSource`, and `addInput`.

  A client may also call `forceReseed` and `allowReseed` to cause a reseed to occur. However, reseed control is implemented internally and should occur regularly even if the client does not call these methods.

  The reseed methods take an optional *ticks* argument that affects how long the reseed will take. The class implements a default number, which should be sufficient, but the user can override it.

  The `Yarrow` class is not threadsafe.

  `getOuput(`*num*`)`
  : Return *num* bytes of random data.

  `addSource(`*sourceName*`)`
  : Intialize a new entropy source named *sourceName*.

  `addInput(`*source, input, estbits*`)`
  : Add *input* string to *source* pool, estimating *estbits* of entropy.

  `forceReseed(`$\left[$*ticks*$\right]$`)`
  : Force a reseed.

`allowReseed(`[*ticks*]`)`
Perform a reseed in enough entropy is available.

`reseed(`[*ticks*]`)`
Use current entropy to generate new seed.

`ThreadedYarrow()`
This subclass of `Yarrow` adds locking on `addInput`, `getOutput` and `allowReseed` calls. This locking is sufficient to make the class thread-safe.

`EntropyGatherer(`*jobs, yarrow*`)`
An `EntropyGatherer` runs a collection of system utilities periodically. Each instance is a `threading.Thread` designed to be run via a `start` method. One instance should be created for each collection of utilities that are run at the same period.

The arguments are: *jobs*, a description of the system utilities to run, and *yarrow*, a `Yarrow` instance to feed the data to.

`pad64(`s`)`
Returns a copy s padded to 64 bits.

`hash(`x`)`
Returns the SHA digest of x.

`hash_ex(`m, k`)`
Extend input m to k bytes using SHA digests.

5.10 `pisces.ttls` – Trivial transport-layer security

This package, which contains submodules `config` and `protocol`, implement a protocol loosely based on the TLS protocol (RFC 2246). There is no reason to believe that this new protocol offers any of the security guarantees of the real TLS protocol. It does use some of the same basic ideas, and it might be possible to show that it is secure.

The protocol is intended as an example of how to define new SPKI-based applications. It will be documented in a future release.

5.11 `pisces.spkilib` – SPKI package

The SPKI implementation is a contained in the `pisces.spkilib` sub-package. The core of the implementation, contained in `pisces.spkilib.spki`, defines Python objects for each of the SPKI/SDSI 2.0 objects defined in the last draft of the structure document [1].

5.12 `pisces.spkilib.config` – Configuration and command-line argument handling

This module defines an `Options` class and several helper functions that are used by **spkitool.py** to process command-line arguments and establish the location of configuration files and the `pisces.spkilib.keystore.KeyStore`. It can be used by other programs based on `pisces.spkilib` to perform the same functions.

Further documentation is not yet available.

5.13 `pisces.spkilib.database` – Store keys and certificates in text file

This module defines a format for storing S-expressions in a text file. The file contains one or more base64-encoded S-expressions. Comment lines begin with a #. The classes defined all include a human-readable

description of the S-expression in a comment before the actual encoded object. Each type of database defines some specific semantics for the object.

`AbstractDatabase(`*path*`)`

The `AbstractDatabase` class is the parent of all the specific database types. It implements two general methods and requires that subclasses implement three other that describe the specific kind of S-expressions that are supported.

The constructor argument *path* specifies the file that contains the database.

The two general methods are:

`reload(`$\left[$*create*$\right]$`)`

Load the contents of the database file into memory. If the optional *create* argument is non-zero, succeed if the file does not exist; otherwise, an `IOError` will be raised if the file does not exist.

`rewrite()`

Save the current contents of the database into the database file.

The methods that must be implemented by subclasses are:

`loadObject(`*obj*`)`

Called for each S-expression in the file when `reload` is executed.

`getObjects()`

Called by `rewrite`. This method should return a list of all objects to be written out.

`writeStorageHint(`*obj, io*`)`

Called for each S-expression to be written out by `rewrite`. A hint for S-expression *obj* should be written to the file-like object *io*.

`DebugDatabase(`*path*`)`

This method loads in an arbitrary database, but is not capable of rewriting it. It is useful for debugging a database.

`ACL(`*path*$\left[$*, create*$\right]$`)`

This class defines a file containing `pisces.spkilib.spki.Entry` objects. Each `Entry` is written with two hints: the subject and the tag.

This class defines the following additional methods:

`add(`*entry*`)`

Add a new `Entry` object to the database.

`lookup(`*subject*`)`

Return all `Entry` objects that match *subject*.

`CertificateDatabase(`*path*$\left[$*, create*$\right]$`)`

This class defines a file containing `pisces.spkilib.spki.Cert` objects, including name certs. Each cert is written with two hints: the subject and issuer.

This class defines the following additional methods:

`lookupBySubject(`*subject*`)`

Return all certs that match *subject*.

`lookupByIssuer(`*issuer*`)`

Return all certs that match *issuer*.

`add(`*cert*`)`

Add a new cert *cert* to the database.

`delete(`*obj*`)`

Delete all certificates with issuer and subject fields that match *obj*.

`PrincipalDatabase(`*path*$\left[$*, create*$\right]$`)`

A `PrincipalDatabase` stores public keys.

This class defines the following additional methods:

`add(`*key*`)`
Add the public key *key*.

`lookup(`*p*`)`
Return the key corresponding to hash *p*.

`delete(`*p*`)`
Delete a public key. Accepts a single argument *p* that can be either a key or its hash.

`PrivateKeyDatabase(`*path*[, *create*]`)`
This class stores private keys. Unlike other database classes, it depends on the order of the objects in the database file being preserved. It stores a collection of private keys and their associated public keys. One of the keys is marked as the default key.

This class defines the following additional methods:

`lookup(`*hash*`)`
Return a private key for the principal *hash*. The *hash* is of the public part of the key pair.

`setDefault(`*hash*`)`
Make the private key for the principal *hash* the default key.

`getDefault()`
Return the hash of the public part of the default key.

`add(`*pub, priv*`)`
Add the key pair with public part *pub* and private part *priv* to the database.

`listPublicKeys()`
Return a list of all the public keys.

`listPrivateKeys()`
Return a list of all the private keys.

5.14 `pisces.spkilib.keystore` – Abstract storage interface for keys and certifications

`getPrincipal(`*obj*`)`
Return the principal associated with a SPKI object. The implementation either returns the object directly, if it is a hash, or calls the object's `getPrincipal` method.

`KeyStore(`*path*`)`
A `KeyStore` provides a high-level interface for a collection of keys and certificates stored in files. The constructor takes a *path* argument that specifies the directory where the files are located.

A `KeyStore` uses three files: 'keys', 'private', and 'certs'. Each file uses is read and written using `pisces.spkilib.database`.

Each `KeyStore` has a default key that is used to create and resolve name certs.

`close()`
Calls the `save` method if changes have been made since the last save.

`save()`
Writes the contents of the `KeyStore` to files.

`setDefaultKey(`*hash*`)`
Make key with has *hash* be the default key. The `KeyStore` must already contain the private key.

`getDefaultKey()`
Return the hash of the default key.

`addPrivateKey(`*key, pub, pword,* [*bogus*]`)`
Add a private key *key* with corresponding public key *pub*. The key is encrypted using password *pword* and marked as bogus if the optional *bogus* argument is non-zero. The key is encrypted using `pisces.spkilib.spki.encryptWithPassword`. The public key is not added to the database.

`addPublicKey(`*key*`)`
Add the public key *key*.

`addCert(`*cert*`)`
Add the certificate *cert* to the database. A name cert should be added using `addName`.

`addName(`*cert*`)`
Add the name certificate *cert* to the database.

`lookupKey(`*hash*`)`
Return the public key corresponding to *hash*.

`lookupPrivateKey(`*hash*`)`
Return the encrypted private key corresponding to *hash*. The hash is of the public key.

`lookupName(`*name*[, *namespace*]`)`
Return a list of certificates issued for *name*, which can be either an instance of `pisces.spkilib.spki.Name` or a simple string. If *name* is a string, a public key or hash must be supplied as the optional *namespace* argument.

`lookupCertBySubject(`*subj*`)`
Return all certificates with a subject matching *subj*.

`lookupCertByIssuer(`*iss*`)`
Return all certificates with an issuer matching *iss*. A certificate with a name in the issuer slot matches when the name is identical to *iss* or *iss* is the principal at the root of a fully qualified name.

`listPublicKeys()`
Return the hashes of all public keys in the `KeyStore`.

`listPrivateKeys()`
Return the hashes of all public keys corresponding to private keys in the `KeyStore`.

`listCerts()`
Return a list of all certificates in the `KeyStore`.

`MultiKeyStore(`[*readers, writers, both, private*]`)`
A `MultiKeyStore` instance provides a `KeyStore`-interface on top of several underlying `KeyStore` implementations. It can be used to share a `KeyStore` among several users.

The constructor accepts objects implementing the `KeyStore` interface as arguments. It uses keyword arguments to indicate whether a particular object should be read-only or read-write and whether it should be used to store private keys. The *readers* argument accepts a list of objects that will be used for lookups only. The *writers* argument accepts a list of objects that will be used for adds only. The *both* argument accepts a list of objects that are used for lookups and adds. The *private* argument accepts a list of objects that can be used to store private keys; a *writer* object will not be used for private keys unless it is also in the *private* list.

5.15 `pisces.spkilib.sexp` – Support routines from S-expressions

This module defines public functions for manipulating S-expressions, and a collection of helper functions that are used by by other modules in `pisces.spkilib` to create S-expressions for standard SPKI objects.

The SPKI structure draft [1] defines a canonical S-expression as follows:

"All SPKI structures communicated from one machine to another must be in canonical form. If canonical S-expressions need to be transmitted over a 7-bit channel, there is a form defined for base64 encoding them.

"A canonical S-expression is formed from binary byte strings, each prefixed by its length, plus the punctuation characters `()[]`. The length of a byte string is a non-negative ASCII decimal number, with no unnecessary leading `0` digits, terminated by `:`. The canonical form is a unique representation of an S-expression and is used as the input to all hash and signature functions."

`parse(`*buf*`)`
: Parse the S-expression *buf* encoded in canonical form or in base64. Returns a `SExp` instance or raises `ParseError`. Can raise `ParseError`.

`parseText(`*buf*`)`
: Parse a human-readable version of an S-expression *buf*. The human-readable form is roughly equivalent to the advanced form described in the struction document [1], but has some significant shortcomings. It is not possible to use `parseText` to parse an S-expression that includes multi-line base64 output. It is most useful for parsing simple S-expressions entered by users on a command line.

`construct(`*elts*`)`
: Construct an S-expression from the Python sequence *elts*. Each element of *elts* must be a string or another sequence.

`SExp(`$\left[$*canon*$\right]$, $\left[$*repr*$\right]$`)`
: The `SExp` class represents S-expressions. Instances behave like sequences, allowing you to access element n of the S-expression using `sx[n]`.

  The constructor takes takes a single keyword argument. It must be either *canon*, a string containing the canonical representation of the S-expression, or *repr*, a Python sequence representing the S-expression. These two constructor forms are equivalent to calling the functions *parse* and *construct* below. User code will be clearer if these functions are called.

  The `__str__` method, called by `str` and `print`, converts the S-expression to a human-readable form. The pretty-printer is not ideal, but it is better than nothing.

  `encode_base64()`
  : Return a string containing the base64 encoding of the S-expression.

  `encode_canonical()`
  : Return a string containing the canonical encoding of the S-expression.

`ParseError`
: A `ParseError` is raised when `parse` is called with invalid data. The exception object contains three attributes: `exp`, the data expected, `got`, the data actually found, and `ref`, the encoded data that contained the error. The `ref` attribute may be None.

5.16 `pisces.spkilib.sexpsocket` – Send and receive s-exps over sockets

This module defines a socket wrapper `SexpSocket` that supports sending and receiving S-expressions over a Python socket object. It supports simple buffering.

Further documentation is not yet available.

5.17 `pisces.spkilib.spki` – Core SPKI implementation

This module implements Python classes that correspond to SPKI object. The objects defined by SPKI are S-expressions consisting of an object name and zero or more "parts." This module defines a Python class for each SPKI object, where the object name corresponds to the class name and the parts correspond to instance attributes.

Each SPKI object is represented in Pisces by a subclass of `SPKIObject`. These classes have attributes for each of the parts of the S-expression. Some classes also have methods for performing operations using the data contained in the SPKI object; e.g. `public-key` objects have `encrypt` and `decrypt` methods.

`SPKIObject()`
: This abstract base class defines the `sexp` method and several operators that rely on the S-expression of an object.

The constructors for `SPKIObject` instances take a number of argument equal to the number of parts in the S-expression, i.e. the number of parts following the object name.

`sexp()`
: Returns a `pisces.spkilib.sexp.SExp` for the object.

The following methods use Python operator overloading. They should not be called directly but provide support for Python builtin operations like `repr`, comparison, and use as a dictionary key.

`__repr__()`
: Returns the advanced form of the S-expression.

`__cmp__(`*other*`)`
: `SPKIObject` instances are compared using the canonical S-expression encoding.

`__hash__()`
: The hash of an object is that hash of the canonical encoding of its S-expression.

The object names used by SPKI can not be used directly as Python class names because they contain hyphens. They also clash with typical Python style which uses capital letters. SPKI object names are converted to class names using the `name_to_impl` function.

`name_to_impl(`*name*`)`
: Returns the class name corresponding to a SPKI object. Names are translated as follows: The SPKI name is broken up into multiple components at each hyphen. The first letter of each component is capitalized. If the first character of the first component is a number, an underscore is prepended. The SPKI name `*` is translated as `TagExpr`.

  For example, the SPKI name `rsa-pkcs1-md5` is translated as `RsaPkcs1Md5`.

In some cases, the name returned by `name_to_impl` refers to a factory function that will produce instances of the desired class. The factory functions are necessary when the class returned depends on some part of the S-expression other than the object name.

There are several helper functions that convert from S-expressions to Python instances.

`parse(`*buf*`)`
: Parse the canonical S-expression *buf* and return a `SPKIObject` instance. Uses `pisces.spkilib.sexp.parse`.

`parseText(`*s*`)`
: Parse the human-readable S-expression *s* and return a `SPKIObject` instance. Uses `pisces.spkilib.sexp.parseText`.

`Evaluator(`**namespaces*`)`
: An `Evaluator` instance is used to translate from S-expressions to Python objects. It contains one or more namespaces that map from SPKI object names to factory functions and classes. The optional *namespaces* arguments must support the mapping protocol.

`eval(`*s*, $\left[$*tag*$\right]$`)`
: Returns a `SPKIObject` corresponding to the `pisces.spkilib.sexp.SExp`. It uses an `Evaluator` intialized with the current module's namespace.

The module defines the following utility functions:

`isPrincipal(`*obj*`)`
: Return true if *obj* is a public key or hash.

`encryptWithPassword(`*object, pw*$\left[$*, bogus*$\right]$`)`
: Return a `PasswordEncrypted` instance containing an encrypted copy of *object*. The encryption is performed using the PBES2 scheme defined by PKCS #5 using *pw* as the password, a triple DES in CBC mode as the cipher.

  If the optional argument *bogus* is specified, the object is marked as using a bogus password. This is useful only for testing purposes.

`getTime(`[*t*]`)`
: Return a time in SPKI format. If *t* is specified, it must be a numer representing the current Unix time, e.g. as returned by `time.time`. If *t* is not specified, the current time is returned.

`checkTime(`*t*`)`
: Returns true if *t* is a syntactically valid SPKI time.

`getIssuerAndSubject(`*obj*[*warning*]`)`
: Retrieve issuer and subject from *obj*, which may be a Sequence containing a cert. This function will search through a sequence looking for a cert object or use a cert object passed directly.

`extractSignedCert(`*seq*`)`
: Extract a cert and its signature from a sequence.

`makePublicKey(`*impl*`)`
: Return a `PublicKey` SPKI object from *impl*, an instance of `pisces.pkcs1.RSA_pkcs1`. By default, produces an `rsa-pkcs1-md5` key.

`makePrivateKey(`*impl*`)`
: Return a `PrivateKey` SPKI object from *impl*, an instance of `pisces.pkcs1.RSA_pkcs1`. By default, produces an `rsa-pkcs1-md5` key.

`setHashAlgorithm(`*alg*`)`
: Change the hash function used by `makePublicKey` and `makePrivateKey` to the algorithm identified by *alg*, a `pisces.asn1.OID`.

`makeRSAKeyPair(`*bits*`)`
: Create a new *bits*-bit RSA key pair. Return two values, the public and private components. Uses `makePublicKey` and `makePrivateKey`.

`RSAKeyMaker(`[*algid*]`)`
: The functions `makePublicKey` and `makePrivateKey` are defined by a default instance of `RSAKeyMaker`. This class, which is instantiated with the OID of a hash function, generates keys that use that hash function for signing.

  `makePublicKey(`*impl*`)`
  : See the function `makePublicKey`.

  `makePrivateKey(`*impl*`)`
  : See the function `makePrivateKey`.

`makeCert(`*issuer, subject, tag*[*, propagate* [*, valid*]]`)`
: Create a SPKI `cert`. The *issuer* and *subject* must be principals; the function will wrap them in `issuer` and `subject` S-expressions. If the *propagate* keyword arg is true, a propagate entry will be added to the cert. If the *valid* keyword argument is supplied, it should contain a sequence of validity constraints, e.g. `not-before`, `not-after`, and `online`.

`makeNameCert(`*issuer, subject, name*[*, valid*]`)`
: Create a SPKI name `cert` binding the string *name* to *subject* in *issuer*'s namespace. The *valid* keyword argument should contain one or more validity constraints.

`makeAclEntry(`*subject, valid, propagate, permissions*`)`
: Return a `Entry` object for an access control list. An `Entry` is like a `Cert`, but without an issuer.

The rest of this section describes the Python classes that correspond to SPKI objects.

Many of the SPKI objects defined in this module do not have methods associated with them. They do define attributes, which are themselves SPKI objects or strings. These objects include `Hash`, `Signature`, `Do`, and `Name`. Other objects are always contained by another object; e.g. `RSAPublicKey` is always used as a component of `PublicKey`. The contained classes are not documented.

`PublicKey(`*key*`)`

The implementation delegates all calls to the contained `RSAPublicKey` object, which in turn delegates to a `pisces.pkcs1.RSA_pkcs1` object.

`verify(`*obj, sig*`)`
: Verify that the `Signature` *sig* matches *obj*.

`getPrincipal()`
: Return the principal (hash) of the key.

`encrypt(`*plain*`)`
: Return a string containing *plain* encrypted with the key.

`decrypt(`*cipher*`)`
: Return a string containing the plain text recover from decrypting the string *cipher* with the key.

`PrivateKey(`*key*`)`
: The implementation delegates all calls to the contained `RSAPrivateKey` object, which in turn delegates to a `pisces.pkcs1.RSA_pkcs1` object.

`sign(`*obj*`)`
: Return a `Signature` object for the SPKI object or string *obj*.

`getPrincipal()`
: Return the principal (hash) of the public key.

`getPublicKey()`
: Return the public key that corresponds to this private key.

`encrypt(`*plain*`)`
: Return a string containing *plain* encrypted with the key.

`decrypt(`*cipher*`)`
: Return a string containing the plain text recover from decrypting the string *cipher* with the key.

`Sequence()`
: Behaves like a Python sequence.

`Valid()`
: subclasses `NotBefore`, `NotAfter`, `Online`

`isValid()`

`Cert()` `getTag()`

`isValid()`

`isNameCert()`

`getSubject()`

`getIssuer()`

`SignedCert(`*cert, sig*`)` `getSequence()`

`verifySignature(`*keydb*`)`

`Issuer()` `isName()`

`getPrincipal()`

`Subject()` `isName()`

`getPrincipal()`

`Entry()` `getTag()`

`isValid()`

`Keyholder()` `getPrincipal()`

`Tag()`

supports __nonzero__, __cmp__, __and__

`intersect(`*atag*`)`

`TagExpr()`

note: realy _TagExpr

subclasses `TagStar`, `TagSet`, `TagRange`, `TagPrefix`

`copy()`

`contains()`

`intersect()`

`TagSet()` `add(`*arg*`)`

`AppTag()` `copy()`

XXX need to define SPKI-style bnf for password-encrypted pbes2-hmac 3des-cipher

`PasswordEncrypted()` `getKey(`$\left[pw\right]$`)`

`decrypt(`$\left[pw\right]$`)`

`isBogus()`

`Pbes2Hmac()` `getKey(`*pw*`)`

`_3desCipher()` `decrypt(`*key*`)`

5.18 `pisces.spkilib.verify` – Certificate chain verification

This module processes a collection of certificates to yield an authorization result. The SPKI theory RFC [2] describes a tuple-reduction process as an example. The Pisces implementation uses a search mechanism that has the same semantics.

This module is a little out of date. It uses the `pisces.spkilib.database` interface instead of the higher-level `pisces.spkilib.keystore` interface. It will be revised before the final version 1.0 release.

`Verifier(`*acl, certs, keys*`)`

A `Verifier` uses an access control list *acl*, a collection of certificates *cert*, and a collection of public keys *keys* to make access control decisions. Each argument should be a database from `pisces.spkilib.database`: *acl* is a `ACL`, *cert* is a `CertificateDatabase`, and *keys* is a `PrincipalDatabase`.

`verify(`*prin, perm*$\left[$*, delegate*$\right]$`)`

Find a valid certificate chain from an ACL entry to the prinicipal *prin* that grants permission *perm*.

This method searches through the collection of certificates to find a valid chain from an access control list entry to the principal making the request. The return value is a sequence of certificates forming a valid chain. The first entry in the sequence is a `pisces.spkilib.spki.Entry` object.

Each subsequent element will be a certificate delegating some permissions from the previous element to the next element. The last element will delegate permissions to the principal.

There is a *delegate* argument, because there can not be more than one non-delegate-able certificate between a valid delegate-able certificate and the principal requesting permission. That one certificate is the one that grants permissions to the principal, but doesn't allow the principal to delegate further. The delegate flag should always be true when called recursively.

`ReferenceMonitor(`*acl, certs, keys*`)`
: A `ReferenceMonitor` provides a simpler interface to a `Verifier`. The constructor arguments for `ReferenceMonitor` are the same as for `Verifier`.

  `checkPermission(`*caller, perm*`)`
  : Check to see if principal *caller* has permission *perm* by calling `Verifier.verify`. Raises `SecurityError` if the permission does not exist. Returns None otherwise.

`SecurityError()`
: Raised by `ReferenceMonitor` when `checkPermission` fails.

A Installation

You need to have the following software installed:

- Python 1.5 or higher.

  You can download Python from http://www.python.org/download.

- distutils 0.8 or higher

  You can download distuils from http://www.python.org/sigs/distutils-sig/download.html.

- OpenSSL 0.9.5 or higher

  You can download OpenSSL from http://www.openssl.org/source/. This package is not used directly by Pisces. Rather Pisces uses amkCrypto, which requires OpenSSL.

- amkCrypto-0.1.2 or higher

  You can download it from ftp://starship.python.net/pub/crew/amk/crypto/.

The top-level Pisces directory contains two scripts. The first **test.py** will run some tests to make sure that the Pisces libraries are working and that all the other software packages that are required have been installed. (The test framework will probably improve in future releases.)

The other script is **setup.py**. This is an install script written using distutils. To install Pisces so that it is available for use from Python, run `./setup.py install`.

References

[1] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian M. Thomas, and Tatu Ylonen. Simple public key certificate. Distributed as Internet Draft draft-ietf-spki-cert-structure-06.txt., July 1999.

[2] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian M. Thomas, and Tatu Ylonen. SPKI certificate theory. RFC 2693, September 1999.

[3] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.