Security and Cryptography All Versions

Python, being one of the most popular languages in computer and network security, has great potential in security and cryptography. This topic deals with the cryptographic features and implementations in Python from its uses in computer and network security to hashing and encryption/decryption algorithms.

Examples

Secure Password Hashing

The PBKDF2 algorithm exposed by `hashlib` module can be used to perform secure password hashing. While this algorithm cannot prevent brute-force attacks in order to recover the original password from the stored hash, it makes such attacks very expensive.

```
import hashlib
import os

salt = os.urandom(16)
hash = hashlib.pbkdf2_hmac('sha256', b'password', salt, 100000)
```

PBKDF2 can work with any digest algorithm, the above example uses SHA256 which is usually recommended. The random salt should be stored along with the hashed password, you will need it again in order to compare an entered password to the stored hash. It is essential that each password is hashed with a different salt. As to the number of rounds, it is recommended to set it as high as possible for your application .

If you want the result in hexadecimal, you can use the `binascii` module:

```
import binascii
hexhash = binascii.hexlify(hash)
```

Note : While PBKDF2 isn't bad, bcrypt and especially scrypt are considered stronger against brute-force attacks. Neither is part of the Python standard library at the moment.

Calculating a Message Digest

The `hashlib` module allows creating message digest generators via the `new` method. These generators will turn an arbitrary string into a fixed-length digest:

```
import hashlib

h = hashlib.new('sha256')
h.update(b'Nobody expects the Spanish Inquisition.')
h.digest()
# ==> b'.\xdf\xda\xdaVR[\x12\x90\xff\x16\xfb\x17D\xcf\xb4\x82\xdd)\x14\xff\xbc\xb6Iy\x0c\x0eX\x9
```

Note that you can call `update` an arbitrary number of times before calling `digest` which is useful to hash a large file chunk by chunk. You can also get the digest in hexadecimal format by using `hexdigest` :

```
h.hexdigest()
# ==> '2edfdada56525b1290ff16fb1744cfb482dd2914ffbcb649790c0e589e462d3d'
```

Available Hashing Algorithms

`hashlib.new` requires the name of an algorithm when you call it to produce a generator. To find out what algorithms are available in the current Python interpreter, use `hashlib.algorithms_available` :

```
import hashlib
hashlib.algorithms_available
# ==> {'sha256', 'DSA-SHA', 'SHA512', 'SHA224', 'dsaWithSHA', 'SHA', 'RIPEMD160', 'ecdsa-with-SH
```

The returned list will vary according to platform and interpreter; make sure you check your algorithm is available.

There are also some algorithms that are *guaranteed* to be available on all platforms and interpreters, which are available using `hashlib.algorithms_guaranteed` :

```
hashlib.algorithms_guaranteed
```

```
# ==> {'sha256', 'sha384', 'sha1', 'sha224', 'md5', 'sha512'}
```

File Hashing

A hash is a function that converts a variable length sequence of bytes to a fixed length sequence. Hashing files can be advantageous for many reasons. Hashes can be used to check if two files are identical or verify that the contents of a file haven't been corrupted or changed.

You can use `hashlib` to generate a hash for a file:

```
import hashlib

hasher = hashlib.new('sha256')
with open('myfile', 'r') as f:
    contents = f.read()
    hasher.update(contents)

print hasher.hexdigest()
```

For larger files, a buffer of fixed length can be used:

```
import hashlib
SIZE = 65536
hasher = hashlib.new('sha256')
with open('myfile', 'r') as f:
    buffer = f.read(SIZE)
    while len(buffer) > 0:
        hasher.update(buffer)
        buffer = f.read(SIZE)
print(hasher.hexdigest())
```

Asymmetric RSA encryption using pycrypto

Asymmetric encryption has the advantage that a message can be encrypted without exchanging a secret key with the recipient of the message. The sender merely needs to know the recipients public key, this allows encrypting the message in such a way that only the designated recipient (who has the corresponding private key) can decrypt it. Currently, a third-party module like pycrypto is required for this functionality.

```
from Crypto.Cipher import PKCS1_OAEP
from Crypto.PublicKey import RSA

message = b'This is a very secret message.'

with open('pubkey.pem', 'rb') as f:
    key = RSA.importKey(f.read())
cipher = PKCS1_OAEP.new(key)
encrypted = cipher.encrypt(message)
```

The recipient can decrypt the message then if they have the right private key:

```
with open('privkey.pem', 'rb') as f:
    key = RSA.importKey(f.read())
cipher = PKCS1_OAEP.new(key)
decrypted = cipher.decrypt(encrypted)
```

Note : The above examples use PKCS#1 OAEP encryption scheme. pycrypto also implements PKCS#1 v1.5 encryption scheme, this one is not recommended for new protocols however due to known caveats .

Generating RSA signatures using pycrypto

RSA can be used to create a message signature. A valid signature can only be generated with access to the private RSA key, validating on the other hand is possible with merely the corresponding public key. So as long as the other side knows your public key they can verify the message to be signed by you and unchanged - an approach used for email for example. Currently, a third-party module like pycrypto is required for this functionality.

```
import errno

from Crypto.Hash import SHA256
from Crypto.PublicKey import RSA
from Crypto.Signature import PKCS1_v1_5
```

```
message = b'This message is from me, I promise.'

try:
    with open('privkey.pem', 'r') as f:
        key = RSA.importKey(f.read())
except IOError as e:
    if e.errno != errno.ENOENT:
        raise
    # No private key, generate a new one. This can take a few seconds.
    key = RSA.generate(4096)
    with open('privkey.pem', 'wb') as f:
        f.write(key.exportKey('PEM'))
    with open('pubkey.pem', 'wb') as f:
        f.write(key.publickey().exportKey('PEM'))

hasher = SHA256.new(message)
signer = PKCS1_v1_5.new(key)
signature = signer.sign(hasher)
```

Verifying the signature works similarly but uses the public key rather than the private key:

```
with open('pubkey.pem', 'rb') as f:
    key = RSA.importKey(f.read())
hasher = SHA256.new(message)
verifier = PKCS1_v1_5.new(key)
if verifier.verify(hasher, signature):
    print('Nice, the signature is valid!')
else:
    print('No, the message was signed with the wrong private key or modified')
```

Note : The above examples use PKCS#1 v1.5 signing algorithm which is very common. pycrypto also implements the newer PKCS#1 PSS algorithm, replacing `PKCS1_v1_5` by `PKCS1_PSS` in the examples should work if you want to use that one. Currently there seems to be little reason to use it however.

Symmetric encryption using pycrypto

Python's built-in crypto functionality is currently limited to hashing. For encryption a third-party module like pycrypto is required. For example, it provides the AES algorithm which is considered state of the art for symmetric encryption.

```
import hashlib
import math
import os

from Crypto.Cipher import AES

message = b'Lorem ipsum'
password = b'highly secure encryption password'
iv = os.urandom(16)
key = hashlib.pbkdf2_hmac('sha256', password, b'aes key', 100000, dklen=32)

message = message.ljust(int(math.ceil(len(message) / 16.0) * 16), b'\0')
encrypted = AES.new(key, AES.MODE_CBC, iv).encrypt(message)
```

The AES algorithm takes three parameters: encryption key, initialization vector (IV) and the actual message to be encrypted. AES key length must be a multiple of 128 bits so a passphrase cannot be used directly. Here we use the built-in implementation of the PBKDF2 algorithm to generate a 256 bit key from the password.

The initialization vector is typically a random string, its purpose is making sure that encrypting the same message twice will not produce the same result again. It needs to be stored along with the encrypted message, as knowing it is necessary to successfully decrypt the message.

Finally, there is also something to consider about the message: its length must be a multiple of 16 bytes. The example above solves this issue by padding the message with zero bytes as necessary.

The following code will decrypt the message again:

```
key = hashlib.pbkdf2_hmac('sha256', password, b'aes key', 100000, dklen=32)
message = AES.new(key, AES.MODE_CBC, iv).decrypt(encrypted)
message = message.rstrip(b'\0')
```

Note that the zero bytes used for padding have to be removed after decryption.

Syntax

```
hashlib.new(name)
```

```
hashlib.pbkdf2_hmac(name, password, salt, rounds, dklen=None)
```

Parameters

Remarks

Many of the methods in `hashlib` will require you to pass values interpretable as buffers of bytes, rather than strings. This is the case for `hashlib.new().update()` as well as `hashlib.pbkdf2_hmac`. If you have a string, you can convert it to a byte buffer by prepending the character `b` to the start of the string:

```
  "This is a string"
b"This is a buffer of bytes"
```