A Working Introduction to Crypto with PyCrypto

Kyle Isom
coder@kyleisom.net
https://www.github.com/kisom/crypto_intro

November 25, 2012

1 Introduction

Recently at work I have been using the PyCrypto[1] libraries quite a bit. The documentation is pretty good, but there are a few areas that took me a bit to figure out. In this post, I'll be writing up a quick overview of the PyCrypto library and cover some general things to know when writing cryptographic code in general. I'll go over symmetric, public-key, hybrid, and message authentication codes. Keep in mind this is a quick introduction and a lot of gross simplifications are made. For a more complete introduction to cryptography, take a look at the references at the end of this article. This article is just an appetite-whetter - if you have a real need for information security you should hire an expert. Real data security goes beyond this quick introduction (you wouldn't trust the design and engineering of a bridge to a student with a quick introduction to civil engineering, would you?)

Some quick terminology: for those unfamiliar, I introduce the following terms:

plaintext : the original message

ciphertext : the message after cryptographic transformations are applied to obscure the original message.

encrypt : producing ciphertext by applying cryptographic transformations to plaintext.

decrypt : producing plaintext by applying cryptographic transformations to ciphertext.

cipher : a particular set of cryptographic transformations providing means of both encryption and decryption.

hash : a set of cryptographic transformations that take a large input and transform it to a unique (typically fixed-size) output. For hashes to be cryptographically secure, collisions should be practically nonexistent. It should be practically impossible to determine the input from the output.

Cryptography is an often misunderstood component of information security, so an overview of what it is and what role it plays is in order. There are four major roles that cryptography plays:

confidentiality : ensuring that only the intended recipients receive the plaintext of the message.

data integrity : the plaintext message arrives unaltered.

entity authentication : the identity of the sender is verified. An entity may be a person or a machine.

message authentication : the message is verified as having been unaltered.

[1] https://www.dlitz.net/software/pycrypto/

Note that cryptography is used to obscure the contents of a message and verify its contents and source. It will **not** hide the fact that two entities are communicating.

There are two basic types of ciphers: symmetric and public-key ciphers. A symmetric key cipher employs the use of shared secret keys. They also tend to be much faster than public-key ciphers. A public-key cipher is so-called because each key consists of a private key which is used to generate a public key. Like their names imply, the private key is kept secret while the public key is passed around. First, I'll take a look at a specific type of symmetric ciphers: block ciphers.

2 Block Ciphers

There are two further types of symmetric keys: stream and block ciphers. Stream ciphers operate on data streams, i.e. one byte at a time. Block ciphers operate on blocks of data, typically 16 bytes at a time. The most common block cipher and the standard one you should use unless you have a very good reason to use another one is the AES[2] block cipher, also documented in FIPS PUB 197[3]. AES is a specific subset of the Rijndael cipher. AES uses block size of 128-bits (16 bytes); data should be padded out to fit the block size - the length of the data block must be multiple of the block size. For example, given an input of `ABCDABCDABCDABCD ABCDABCDABCDABCD` no padding would need to be done. However, given `ABCDABCDABCDABCD ABCDABCDABCD` an additional 4 bytes of padding would need to be added. A common padding scheme is to use `0x80` as the first byte of padding, with `0x00` bytes filling out the rest of the padding. With padding, the previous example would look like: `ABCDABCDABCDABCD ABCDABCDABCD\x80\x00\x00\x00`.

Writing a padding function is pretty easy:

```
def pad_data(data):
    # return data if no padding is required
    if len(data) % 16 == 0:
        return data

    # subtract one byte that should be the 0x80
    # if 0 bytes of padding are required, it means only
    # a single \x80 is required.

    padding_required     = 15 - (len(data) % 16)

    data = '%s\x80' % data
    data = '%s%s' % (data, '\x00' * padding_required)

    return data
```

Similarly, removing padding is also easy:

```
def unpad_data(data):
    if not data:
        return data

    data = data.rstrip('\x00')
    if data[-1] == '\x80':
        return data[:-1]
    else:
        return data
```

[2]https://secure.wikimedia.org/wikipedia/en/wiki/Advanced_Encryption_Standard
[3]http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

I've included these functions in the example code for this tutorial.

Encryption with a block cipher requires selecting a block mode[4]. By far the most common mode used is **cipher block chaining** or *CBC* mode. Other modes include *counter (CTR)*, *cipher feedback (CFB)*, and the extremely insecure *electronic codebook (ECB)*. CBC mode is the standard and is well-vetted, so I will stick to that in this tutorial. Cipher block chaining works by XORing the previous block of ciphertext with the current block. You might recognise that the first block has nothing to be XOR'd with; enter the *initialisation vector*[5]. This comprises a number of randomly-generated bytes of data the same size as the cipher's block size. This initialisation vector should random enough that it cannot be recovered; one manner of doing this is to combine a standard UNIX timestamp with a block-size group of random data, using a standard hashing algorithm such as MD5 to make it unique.

One of the most critical components to encryption is properly generating random data. Fortunately, most of this is handled by the PyCrypto library's `Crypto.Random.OSRNG module`. You should know that the more entropy sources available (such as network traffic and disk activity), the faster the system can generate cryptographically-secure random data. I've written a function that can generate a *nonce*[6] suitable for use as an initialisation vector. This will work on a UNIX machine; the comments note how easy it is to adapt it to a Windows machine. This function requires a version of PyCrypto at least 2.1.0 or higher.

```
import time
import Crypto.Random.OSRNG.posix

def generate_nonce():
    rnd = Crypto.Random.OSRNG.posix.new().read(BLOCK_SIZE)
    rnd = '%s%s' % (rnd, str(time.time()))
    nonce = Crypto.Hash.MD5.new(data = rnd)

    return nonce.digest()
```

I will note here that the python 'random' module is completely unsuitable for cryptography (as it is completely deterministic). You shouldn't use it for cryptographic code.

Symmetric ciphers are so-named because the key is shared across any entities. There are three key sizes for AES: 128-bit, 192-bit, and 256-bit, aka 16-byte, 24-byte, and 32-byte key sizes. If you want to use a passphrase, you should use a digest algorithm that produces an appropriately sized digest, and hash that passphrase. For example, for AES-256, you would want to use SHA-256. Here is a sample function to generate an AES-256 key from a passphrase:

```
# generate an AES-256 key from a passphrase
def passphrase(password, readable = False):
    """
    Converts a passphrase to a format suitable for use as an AES key.

    If readable is set to True, the key is output as a hex digest. This is
    suitable for sharing with users or printing to screen when debugging
    code.

    By default readable is set to False, in which case the value it
    returns is suitable for use directly as an AES-256 key.
    """
    key     = Crypto.Hash.SHA256.new(password)
```

[4]https://secure.wikimedia.org/wikipedia/en/wiki/Block_cipher_modes_of_operation
[5]https://secure.wikimedia.org/wikipedia/en/wiki/Initialization_vector
[6]https://secure.wikimedia.org/wikipedia/en/wiki/Cryptographic_nonce

```
    if readable:
        return key.hexdigest()
    else:
        return key.digest()
```

We could include this a set of AES encryption and decryption functions:

```
mode = Crypto.Cipher.AES.MODE_CBC       # shortcut to clean up code

# AES-256 encryption using a passphrase
def passphrase_encrypt(password, iv, data):
    key     = passphrase(password)
    data    = pad_data(data)
    aes     = Crypto.Cipher.AES.new(key, mode, iv)

    return aes.encrypt(data)

# AES-256 decryption using a passphrase
def passphrase_decrypt(password, iv, data):
    key     = passphrase(password)
    aes     = Crypto.Cipher.AES.new(key, mode, iv)
    data    = aes.decrypt(data)

    return unpad_data(data)
```

Notice how the data is padded before being encrypted and unpadded after decryption - the decryption process will not remove the padding on its own.

Of course, storing the password is another matter entirely; I recommend PBKDF2[7]. Other options include bcrypt[8] and scrypt[9]. For symmetric ciphers, it is better to use a completely randomly generated password.

The `pbkdf2` package[10] in PyPI is a Python implementation of PBKDF2. The example from the packages' webpage[11] is very clear:

```
from PBKDF2 import PBKDF2
    from Crypto.Cipher import AES
    import os

    salt = os.urandom(8)    # 64-bit salt
    key = PBKDF2("This passphrase is a secret.", salt).read(32) # 256-bit key
    iv = os.urandom(16)     # 128-bit IV
    cipher = AES.new(key, AES.MODE_CBC, iv)
```

Unless you are you doing interactive encryption passphrase encryption won't be terribly useful. Instead, we just need to generate 32 random bytes (and make sure we keep track of it) and use that as the key:

```
# generate a random AES-256 key
def generate_aes_key():
    rnd     = Crypto.Random.OSRNG.posix.new().read(KEY_SIZE)
    return rnd
```

[7] https://en.wikipedia.org/wiki/Pbkdf2
[8] https://en.wikipedia.org/wiki/Bcrypt
[9] https://en.wikipedia.org/wiki/Scrypt
[10] http://pypi.python.org/pypi/pbkdf2/1.3
[11] https://www.dlitz.net/software/python-pbkdf2/

We can use this key directly in the AES transformations:

```
def encrypt(key, iv, data):
    aes     = Crypto.Cipher.AES.new(key, mode, iv)
    data    = pad_data(data)

    return aes.encrypt(data)

def decrypt(key, iv, data):
    aes     = Crypto.Cipher.AES.new(key, mode, iv)
    data    = aes.decrypt(data)

    return unpad_data(data)
```

That should cover the basics of block cipher encryption. We've gone over key generation, padding, and encryption / decryption. AES-256 isn't the only block cipher provided by the PyCrypto package, but again: it is the standard and well vetted.

3 ASCII-Armouring

I'm going to take a quick detour and talk about ASCII armouring. If you've played with the crypto functions above, you'll notice they produce an annoying dump of binary data that can be a hassle to deal with. One common technique for making the data a little bit easier to deal with is to encode it with base64[12]. There are a few ways to incorporate this into python:

3.1 Absolute Base64 Encoding

The easiest way is to just base64 encode everything in the encrypt function. Everything that goes into the decrypt function should be in base64 - if it's not, the `base64` module will throw an error: you could catch this and then try to decode it as binary data.

3.2 A Simple Header

A slightly more complex option, and the one I adopt in this article, is to use a `\x00` as the first byte of the ciphertext for binary data, and to use `\x41` (an ASCII "`A`") for ASCII encoded data. This will increase the complexity of the encryption and decryption functions slightly. We'll also pack the initialisation vector at the beginning of the file as well. Given now that the `iv` argument might be `None` in the decrypt function, I will have to rearrange the arguments a bit; for consistency, I will move it in both functions. I leave adding it into the `passphrase_encrypt` and `passphrase_decrypt` functions as an exercise for the reader. My modified functions look like this now:

```
def encrypt(key, data, iv, armour = False):
    aes     = Crypto.Cipher.AES.new(key, mode, iv)
    data    = pad_data(data)
    ct      = aes.encrypt(data)         # ciphertext
    ct      = iv + ct                   # pack the initialisation vector in

    # ascii-armouring
    if armour:
        ct = '\x41' + base64.encodestring(ct)
    else:
```

[12] https://secure.wikimedia.org/wikipedia/en/wiki/Base64

```
            ct = '\x00' + ct

        return ct

    def decrypt(key, data, iv = None):
        # remove ascii-armouring if present
        if data[0] == '\x00':
            data = data[1:]
        elif data[0] == '\x41':
            data = base64.decodestring(data[1:])

        iv      = data[:16]
        data    = data[16:]
        aes     = Crypto.Cipher.AES.new(key, mode, iv)
        data    = aes.decrypt(data)
        return unpad_data(data)
```

3.3 A More Complex Container

There are more complex ways to do it (and you'll see it with the public keys in the next section) that involve putting the base64 into a container of sorts that contains additional information about the key.

4 Public Key Cryptography

Now it is time to take a look at public-key cryptography. Public-key cryptography, or PKC, involves the use of two-part keys. The private key is the sensitive key that should be kept private by the owning entity, whereas the public key (which is generated from the private key) is meant to be distributed to any entities which must communicate securely with the entity owning the private key. Confusing? Let's look at this using the venerable Alice and Bob, patron saints of cryptography.

Alice wants to talk to Bob, but doesn't want Eve to know the contents of the message. Both Alice and Bob generate a set of private keys. From those private keys, they both generate public keys. Let's say they post their public keys on their websites. Alice wants to send a private message to Bob, so she looks up Bob's public key from his site. (In fact, there is a way to distribute keys via a central site or entity; this is called a public key infrastructure (PKI). The public key can be used as the key to encrypt a message with PKC. The resulting ciphertext can only be decrypted using Bob's private key. Alice sends Bob the resulting ciphertext, which Eve cannot decrypt without Bob's private key. Hopefully this is a little less confusing.

One of the most common PKC systems is RSA (which is an acronym for the last names of the designers of the algorithm). Generally, RSA keys are 1024-bit, 2048-bit, or 4096-bits long. The keys are most often in PEM[13] or DER[14] format. Generating RSA keys with PyCrypto is extremely easy:

```
    def generate_key(size):
        PRNG    = Crypto.Random.OSRNG.posix.new().read
        key     = Crypto.PublicKey.RSA.generate(size, PRNG)

        return key
```

The `key` that is returned isn't like the keys we used with the block ciphers: it is an RSA object and comes with several useful built-in methods. One of these is the `size()` method, which returns the size of the key in bits minus one. For example:

[13] https://secure.wikimedia.org/wikipedia/en/wiki/Privacy-enhanced_Electronic_Mail
[14] https://secure.wikimedia.org/wikipedia/en/wiki/Distinguished_Encoding_Rules

```
>>> import publickey
>>> key = publickey.generate_key( 1024 )
>>> key.size()
1023
>>>
```

A quick note: I will use 1024-bit keys in this tutorial because they are faster to generate, but in practice you should be using at least 2048-bit keys. The key also includes encryption and decryption methods in the class:

```
>>> import publickey
>>> import base64
>>> message = 'Test message...'
>>> ciphertext = key.encrypt(message, None)
>>> print base64.encodestring(ciphertext[0])
gzA9gXfHqnkValdhhYjRVVSxuygx48i66h0vFUnmVu8FZXJtmaACvNDo43D0vjjHzFiblE1eCFiI
xlhVuHxldWXJSnARgWX1bTY7imR9Hve+WQC8rl+qB5xpq3xnKH7/z8/5YdLvCo/knXYE1cI/XYJP
EP1nA6bUZNj6bD1Zx4w=
```

The `None` that is passed into the encryption function is part of the PyCrypto API for those publickey ciphers requiring an additional random number function to be passed in. It returns a tuple containing only the encrypted message. In order to pass this to the decryption function, we need to pass only the encrypted message as a string:

```
>>> ciphertext = key.encrypt(message, None)[0]
>>> key.decrypt(ciphertext)
'Test message...'
```

While these are simple enough, we could put them into a pair of functions that also include ASCII-armouring:

```
def encrypt(key, message, armour = True):
    ciphertext  = key.encrypt( message, None )
    ciphertext  = ciphertext[0]

    if armour:
        ciphertext = '\x41' + base64.encodestring( ciphertext )
    else:
        ciphertext = '\x00' + base64.encodestring( ciphertext )

    return ciphertext

def decrypt(key, message):
    if   '\x00' == message[0]:
        message = message[1:]
    elif '\x41' == message[0]:
        message == base64.decodestring( message[1:] )

    plaintext   = key.decrypt( message )
    return plaintext
```

These two functions present a common API that will simplify encryption and decryption and make it a little easier to read. Assuming we still have the same `message` definition as before:

```
>>> ciphertext = publickey.encrypt(key, message)
>>> publickey.decrypt(key, ciphertext)
'Test message...'
```

Now, what if we want to export this generated key and read it in later? The key comes with the method `exportKey()`. If the key is a private key, it will export the private key; if it is a public key, it will export the public key. We can write functions to backup our private key (which **absolutely** needs to be kept secure) and a function to export our public key, suitable for uploading to our web page or to a PKI keystore:

```
# backup our key, whether public or private
def export_key(filename, key):
    try:
        f = open(filename, 'w')
    except IOError as e:
        print e
        raise
    else:
        f.write( key.exportKey() )
        f.close()

# will only export the public key
def export_pubkey(filename, key):
    try:
        f = open(filename, 'w')
    except IOError as e:
        print e
        raise
    else:
        f.write( key.publickey().exportKey() )
        f.close()
```

Importing a key is done using the RSA.importKey function:

```
def load_key(filename):
    try:
        f = open(filename)
    except IOError as e:
        print e
        raise
    else:
        key = Crypto.PublicKey.RSA.importKey(f.read())
        f.close()
    return key
```

We can take a look at the difference between the public and private keys:

```
>>> key = publickey.generate_key( 1024 )
>>> print key.exportKey()
-----BEGIN RSA PRIVATE KEY-----
MIICXAIBAAKBgQCpVA2pqLuS1fmutvx/lBhlk+UMXWcZKVzh+n5D6Hv/ZWhlzRuC
q408uhVBUD32ylbQ2iFdhA1leq0xWRGQ8Y3LlO6tQZ0gC2oOHetX3YOghO3q4yMe
wvuU+Wb6bS1aRDc9YV3IMPjQW47MOROUldjMEdJJhfxko5YZuaghhpd56wIDAQAB
AoGAaRznellnT2iLHXO0U1IwruXXOwzEUmdN5G4mcathRhLCcueXW095VqhBR5Ez
```

```
Vf8XU4EFU1MFKei0mLys3ehFV4aoTfU1xm91jXNZrM/rIjHQQObx2fcDSgrM9iyd
kcgGrz5nDvsyxAxOwxCh96vNxZZYTWa8Zcqng1XYeW93nFkCQQC8Rqwn9Sa1UjBB
mIepkcdYfflkzmD7IBcgiTmGFQ9NXiehY6MQd0UJoFYGBEknPazzWQbNVpkZO4TR
oPuKNjSNAkEA5jyWJhKyq2BVD6UP77vYTJu48OhLx4J7qb3DKHnk5syOBnbke2Df
KV1VjRsipSjb4EXAWhWaqnTfPPDbvyWWVwJAWUgSP2iLkJSG+bRBMPJGW/pxF5Ke
fre6/9zTAHhgJ0os9OVw4FAO1v/Hi1bg8dDXgRaImTsloseMtnPmlKYbyQJAbmbr
EQKyTl95KnFaPPj0dXfOrSaW/+pf5jsqlAQvcUTxbcQhN9Bx8mHhHjK+4DfBh7+q
xwfJDKfSTGSq2vPpLQJBAL5irIeHoFESPZZI1NW7OkpKPcO/2ps9NkhgZJQ7Pc11
lWh6Ch2cnBzZmeh6lN/zC4l3mLVhdZSXkEKOzeuFpBs=
-----END RSA PRIVATE KEY-----
>>> pk = key.publickey()
>>> print pk.exportKey()
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCpVA2pqLuS1fmutvx/lBhlk+UM
XWcZKVzh+n5D6Hv/ZWhlzRuCq408uhVBUD32ylbQ2iFdhA1leq0xWRGQ8Y3Ll06t
QZ0gC2oOHetX3YOghO3q4yMewvuU+Wb6bS1aRDc9YV3IMPjQW47MOROUldjMEdJJ
hfxko5YZuaghhpd56wIDAQAB
-----END PUBLIC KEY-----
```

Using the `export_pubkey()` function, you can pass that file around to people to encrypt messages to you. Often, you wil want to generate a keypair to give to people. One convention is to name the secret key 'keyname.prv' (prv for private) and the public key 'keyname.pub'. We will follow that convention in an `export_keypair()` function:

```
def export_keypair(basename, key):
    pubkeyfile   = basename + '.pub'
    prvkeyfile   = basename + '.prv'

    export_key(prvkeyfile, key)
    export_pubkey(pubkeyfile, key)
```

For example, Bob generates a keypair and emails the public key to Alice:

```
>>> key = publickey.generate_key( 1024 )
>>> key.size()
1023
>>> key.has_private()
True
>>> publickey.export_keypair('bob.prv', key)
>>>
```

Then, Assuming Bob gave Alice `bob.pub`:

```
>>> bob = publickey.load_key('./bobpub')
>>> message = 'secret message from Alice to Bob'
>>> print publickey.encrypt(bob, message)
AN6RsuXEeKicUZKtZCsDeqGKeB5em+NG/bgoqr9l8ij2o1Gr9sT69tv0zxgmigK/Jt+gPxg/EDu61
nHmAK0XQV7BvJS5jLuBxdJ0mEpysVClu46XN1KHU2l2DsGht9e8OFvhEfDkI5t/cy/gXr0xz/EUi
rqo8qLd9Mw6TerM8gs8=
```

The ASCII-armoured format makes it convenient for Alice to paste the encrypted message to Bob, so she does, and now Bob has it on his computer. To read it, he does something similar:

```
>>> bob = publickey.load_key('tests/bob.prv')
>>> print publickey.decrypt(bob, message)
secret message from Alice to Bob
```

At this point, Bob can't be sure that the message came from Alice but he can read the message. We'll cover entity authentication in a later section, but first, there's something else I'd to point out:

You might have noticed at this point that public key cryptography appears to be a lot simpler than symmetric key cryptography. The key distribution problem is certainly easier, especially with a proper PKI. Why would anyone choose to use symmetric key cryptography over public key cryptography? The answer is performance: if you compare the block cipher test code (if you don't have a copy of this code, you can get it at the tutorial's github page[15]) with the public key test code, you will notice that the block cipher code is orders of magnitude faster - and it generates far more keys than the public key code. There is a solution to this problem: hybrid cryptosystems.

Hybrid cryptosystems use public key cryptography to establish a symmetric session key. Both **TLS**[16] (Transport Layer Security), and its predecessor **SSL** (Secure Sockets Layer), most often used to secure HTTP transactions, use a hybrid cryptosystem to speed up establishing a secure session. PGP (and hence GnuPG) also uses hybrid crypto.

Let's say Alice and Bob wish to use hybrid crypto; if Alice initiates the session, she should be the one to generate the session key. For example,

```
>>> import block, publickey
>>> session_key = block.generate_aes_key()
>>> alice_key   = publickey.load_key('keys/alice.prv')
>>> bob_key     = publickey.load_key('keys/bob.pub')
>>> encrypted_session_key = encrypt( bob_key, session_key )
```

At this point, Alice should send Bob the `encrypted_session_key`; she should retain a copy as well. They can then use this key to communicate using the much-faster AES256.

In communicating, it might be wise to create a message format that packs in the session key into a header, and encrypts the rest of the body with the session key. This is a subject beyond the realm of a quick tutorial - again, consult with the people who do this on a regular basis.

5 Digital Signatures

In all of the previous examples, we assumed that the identity of the sender wasn't a question. For a symmetric key, that's less of a stretch - there's no differentiation between owners. Public keys, however, are supposed to be associated with an entity. How can we prove the identity of the user? Without delving into too much into social sciences and trust metrics and a huge philosophical argument, let's look at the basics of signatures.

A signature works similarly to encryption, but it in reverse, and it is slightly different: a hash of the message is 'encrypted' by the private key to the public key. The public key is used to 'decrypt' this ciphertext. Contrast this to actual public key encryption: the entire message is encrypted to the private key by the public key, and the private key is used to decrypt the ciphertext. With signatures, the 'encrypted' hash of the message is called the signature, and the act of 'encryption' is termed 'signing'. Similarly, the 'decryption' is known as verification or verifying the signature.

PyCrypto's PublicKey implementations already come with signatures and verification methods for keys using sign() and verify(). The signature is a long in a tuple:

```
>>> key.sign( d, None )
(1738423518152671545669571445860037944518162197656333123466248015147955424248
876723731383711018550231967374810686606623315483033485630977014574359346192927942
623807461783144628656796225504478196458051789241311033020911767301220653148276004
055135752638362705938208187879104016981500905101694922017804476413090 8L,)
```

[15] https://www.github.com/kisom/crypto_tutorial

[16] https://secure.wikimedia.org/wikipedia/en/wiki/Transport_Layer_Security

We can write our own functions to wrap around these two functions and perform ASCII-armouring if desired. Our signature function should take a key and a message (and optionally a flag to ASCII armour the signature), and sign a digest of the message:

```
def sign(key, message, armour = True):
    if not key.can_sign():
        return None

    digest      = Crypto.Hash.SHA256.new(message).digest()
    signature   = key.sign( digest, None )[0]

    if armour:
        sig     = base64.encodestring( str(signature) )
    else:
        sig     = str( signature )

    return sig.strip()
```

The signature is converted to a string to make it easier to pack it into structures and also to give us consistent input to the verify() function.

Verifying the signature requires that we determine if the signature is ASCII- armoured or not, then comparing a digest of the message to the signature:

```
def verify(key, message, signature):
    try:
        sig     = long( signature )
    except ValueError as e:
        sig     = long( base64.decodestring( signature.rstrip('\n') ), )

    digest      = Crypto.Hash.SHA256.new(message).digest()
    return key.verify( digest, (sig, ) )
```

The sign() function returns a signature and the verify() function returns a boolean. Now, Alice can sign her message to Bob, and Bob knows the key belongs to Alice. She sends Bob the signature and the encrypted message. Bob then makes sure Alice's key properly verifies the signature to the encrypted message.

6 Key Exchange

So how does Bob know the key actually belongs to Alice? There are two main schools of thought regarding the authentication of key ownership: centralised and decentralised. TLS/SSL follow the centralised school: a root certificate[17] authority (CA) signs intermediary CA keys, which then sign user keys. For example, if Bob runs Foo Widgets, LLC, he can generate an SSL keypair. From this, he generates a certificate signing request, and sends this to the CA. The CA, usually after taking some money and ostensibly actually verifying Bob's identity[18], then signs Bob's certificate. Bob sets up his webserver to use his SSL certificate for all secure traffic, and Alice sees that the CA did in fact sign his certificate. This relies on trusted central authorities, like VeriSign[19] Alice's web browser would ship with a keystore of select trusted CA public keys (like VeriSigns) that she could use to verify signatures on the certificates from various sites. This system is called a public key infrastructure.

[17] A certificate is a public key encoded with X.509 and which can have additional informational attributes attached, such as organisation name and country.

[18] The extent to which this actually happens varies widely based on the different CAs.

[19] There is some question as to whether VeriSign can actually be trusted, but that is another discussion for another day...

The other school of thought is followed by PGP (and GnuPG) - the decentralised model. In PGP, this is manifested as the Web of Trust[20]. For example, if Carol now wants to talk to Bob and gives Bob her public key, Bob can check to see if Carol's key has been signed by anyone else. We'll also say that Bob knows for a fact that Alice's key belongs to Alice, and he trusts her[21], and that Alice has signed Carol's key. Bob sees Alice's signature on Carol's key and then can be reasonably sure that Carol is who she says it was. If we repeat the process with Dave, whose key was signed by Carol (whose key was signed by Alice), Bob might be able to be more certain that the key belongs to Dave, but maybe he doesn't really trust Carol to properly verify identities. In PGP, Bob can mark keys as having various trust levels, and from this a web of trust emerges: a picture of how well you can trust that a given key belongs to a given user.

The key distribution problem is not a quick and easy problem to solve; a lot of very smart people have spent a lot of time coming up with solutions to the problem. There are key exchange protocols (such as the Diffie-Hellman key exchange[22] and IKE[23] (which uses Diffie-Hellman) that provide alternatives to the web of trust and public key infrastructures.

7 References

- A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *The Handbook of Applied Cryptography*, CRC Press, 5th printing, October 1996.
- B. Schneier. *Applied Cryptography, Second Edition*, John Wiley and Sons, 1996.
- PyCrypto API Documentation. https://www.dlitz.net/software/pycrypto/apidoc/

[20] http://www.rubin.ch/pgp/weboftrust.en.html

[21] It is quite often important to distinguish between *I know this key belongs to that user* and *I trust that user*. This is especially important with key signatures - if Bob cannot trust Alice to properly check identities, she might sign a key for an identity she hasn't checked.

[22] http://is.gd/Tr0zLP

[23] https://secure.wikimedia.org/wikipedia/en/wiki/Internet_Key_Exchange