A Working Introduction to Crypto with PyCrypto

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

October 29, 2013

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, Ill be writing up a quick overview of the PyCrypto library and cover some general things to know when writing cryptographic code in general. Ill 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 wouldnt 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, Ill 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

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 of the most critical components to encryption is properly generating random data. Fortunately, most of this is handled by the PyCrypto librarys `Crypto.Random.OSRNG` module. You should know that the more entropy sources that are 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 Crypto.Random.OSRNG.posix as RNG

def generate_nonce():
    return RNG.new().read(AES.block_size)
```

I will note here that the python 'random' module is completely unsuitable for cryptography (as it is completely deterministic). You shouldnt 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. Instead, we just need to generate 32 random bytes (and make sure we keep track of it) and use that as the key:

```
KEYSIZE = 32


def generate_key():
    return RNG.new().read(KEY_SIZE)
```

We can use this key to encrypt and decrypt data. To encrypt, we need the initialisation vector (i.e. a nonce), the key, and the data. However, the IV isn't a secret. When we encrypt, we'll prepend the IV to our encrypted data and make that part of the output. We can (and should) generate a completely random IV for each new message.

```
def encrypt(data, key):
    """
    Encrypt data using AES in CBC mode. The IV is prepended to the
    ciphertext.
    """
    data = pad_data(data)
    ivec = generate_nonce()
    aes = AES.new(key, AES.MODE_CBC, ivec)
    ctxt = aes.encrypt(data)
    return ivec + ctxt


def decrypt(ciphertext, key):
    """
```

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

```
    Decrypt a ciphertext encrypted with AES in CBC mode; assumes the IV
    has been prepended to the ciphertext.
    """
    if len(ciphertext) <= AES.block_size:
        raise Exception("Invalid ciphertext.")
    ivec = ciphertext[:AES.block_size]
    ciphertext = ciphertext[AES.block_size:]
    aes = AES.new(key, AES.MODE_CBC, ivec)
    data = aes.decrypt(ciphertext)
    return unpad_data(data)
```

However, this is only part of the equation for securing messages: AES only gives us confidentiality. Remember how we had a few other criteria? We still need to add integrity and authenticity to our process. Readers with some experience might immediately think of hashing algorithms, like MD5 (which should be avoided like the plague) and SHA. The problem with these is that they are malleable: it is easy to change a digest produced by one of these algorithms, and there is no indication it's been changed. We need, a hash function that uses a key to generate the digest; the one we'll use is called HMAC. We do not want the same key used to encrypt the message; we should have a new, freshly generated key that is the same size as the digest's output size (although in many cases, this will be overkill).

In order to encrypt properly, then, we need to modify our code a bit. The first thing, you need to know is that HMAC is based on a particular SHA function. Since we're using AES-256, we'll use SHA-384. We say our message tags are computed using HMAC-SHA-384. This produces a 48-byte digest. Let's add a few new constants in, and update the `KEYSIZE` variable:

```
__aes_keylen = 32
__tag_keylen = 48
KEYSIZE = __aes_keylen + __tag_keylen
```

Now, let's add message tagging in:

```
import Crypto.Hash.HMAC as HMAC
import Crypto.Hash.SHA384 as SHA384

def new_tag(ciphertext, key):
    """Compute a new message tag using HMAC-SHA-384."""
    return HMAC.new(key, msg=ciphertext, digestmod=SHA384).digest()
```

Here's our updated encrypt function:

```
def encrypt(data, key):
    """
    Encrypt data using AES in CBC mode. The IV is prepended to the
    ciphertext.
    """
    data = pad_data(data)
    ivec = generate_nonce()
    aes = AES.new(key[:__aes_keylen], AES.MODE_CBC, ivec)
    ctxt = aes.encrypt(data)
    tag = new_tag(ctxt, key[__aes_keylen:])
    return ivec + ctxt + tag
```

Decryption has a snag: what we want to do is check to see if the message tag matches what we think it should be. However, the Python == operator stops matching on the first character it finds that doesn't match.

This opens a verification based on the == operator to a timing attack. Without going into much detail, note that several cryptosystems have fallen prey to this exact attack; the keyczar system, for example, use the == operator and suffered an attack on the system. We'll use the `streql` package (i.e. `pip install streql`) to perform a constant-time comparison of the tags.

```
import streql


__taglen = 48


def verify_tag(ciphertext, key):
    """Verify the tag on a ciphertext."""
    tag_start = len(ciphertext) - __taglen
    data = ciphertext[:tag_start]
    tag = ciphertext[tag_start:]
    actual_tag = new_tag(data, key)
    return streql.equals(actual_tag, tag)
```

We'll also change our decrypt function to return a tuple: the original message (or `None` on failure), and a boolean that will be True if the tag was authenticated and the message decrypted

```
def decrypt(ciphertext, key):
    """
    Decrypt a ciphertext encrypted with AES in CBC mode; assumes the IV
    has been prepended to the ciphertext.
    """
    if len(ciphertext) <= AES.block_size:
        return None, False
    tag_start = len(ciphertext) - __TAG_LEN
    ivec = ciphertext[:AES.block_size]
    data = ciphertext[AES.block_size:tag_start]
    if not verify_tag(ciphertext, key[__AES_KEYLEN:]):
        return None, False
    aes = AES.new(key[:__AES_KEYLEN], AES.MODE_CBC, ivec)
    data = aes.decrypt(data)
    return unpad_data(data), True
```

We could also generate a key using a passphrase, but this is significantly more complex: you should use a key derivation algorithm, such as PBKDF2[7]. A function to derivate a key from a passphrase will also need to store the salt that goes with the passphrase. PBKDF2 will generate a salt to go along with the password; the salt is analogous to the initialisation vector and can be stored alongside the password.

That should cover the basics of block cipher encryption. Weve gone over key generation, padding, and encryption / decryption.

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. There are a few ways to incorporate this into python:

[7]https://en.wikipedia.org/wiki/Pbkdf2

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` decoding 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. My modified functions look like this now:

```
def encrypt(data, key, armour=False):
    """
    Encrypt data using AES in CBC mode. The IV is prepended to the
    ciphertext.
    """
    data = pad_data(data)
    ivec = generate_nonce()
    aes = AES.new(key[:__AES_KEYLEN], AES.MODE_CBC, ivec)
    ctxt = aes.encrypt(data)
    tag = new_tag(ivec+ctxt, key[__AES_KEYLEN:])
    if armour:
        return '\x41' + (ivec + ctxt + tag).encode('base64')
    else:
        return '\x00' + ivec + ctxt + tag

def decrypt(ciphertext, key):
    """
    Decrypt a ciphertext encrypted with AES in CBC mode; assumes the IV
    has been prepended to the ciphertext.
    """
    if ciphertext[0] == '\x41':
        ciphertext = ciphertext[1:].decode('base64')
    else:
        ciphertext = ciphertext[1:]
    if len(ciphertext) <= AES.block_size:
        return None, False
    tag_start = len(ciphertext) - __TAG_LEN
    ivec = ciphertext[:AES.block_size]
    data = ciphertext[AES.block_size:tag_start]
    if not verify_tag(ciphertext, key[__AES_KEYLEN:]):
        return None, False
    aes = AES.new(key[:__AES_KEYLEN], AES.MODE_CBC, ivec)
    data = aes.decrypt(data)
    return unpad_data(data), True
```

3.3 A More Complex Container

There are more complex ways to do it (and youll 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

The original version of this document had examples of using RSA cryptography with Python. However, RSA should be avoided for modern secure systems, and I haven't been using Python, so I'm not very familiar with the options for elliptic curve cryptography. Rather than encouraging the use of a weaker cipher, I've opted to elide this. A possible starting point is to look at Yann Guibet's `pyelliptic`[8] package. It should provide ECDSA for signatures, and ECDH for encryption.

[8] https://github.com/yann2192/pyelliptic