A Working Intro to Cryptography

Illustrated with Python

Kyle Isom

A Working Intro to Cryptography

Illustrated with Python

Kyle Isom

This book is for sale at http://leanpub.com/pycrypto

This version was published on 2014-01-08

Tweet This Book!

Please help Kyle Isom by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought A Working Intro to Cryptography, Illustrated with Python by @kyleisom #pycryptointro

The suggested hashtag for this book is #pycryptointro.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#pycryptointro

Contents

Block Ciphers **1**

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[1] block cipher, also documented in FIPS PUB 197[2]. 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`.

Here's our padding function:

```python
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
```

Our function to remove padding is similar:

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

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

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

Encryption with a block cipher requires selecting a block mode[3]. 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*[4]. 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 library's `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*[5] 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.

```python
import Crypto.Random.OSRNG.posix as RNG

def generate_nonce():
    """Generate a random number used once."""
    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 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. Instead, we just need to generate 32 random bytes (and make sure we keep track of it) and use that as the key:

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

```python
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.

```python
import Crypto.Cipher.AES as AES

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):
    """
    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:

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

Now, let's add message tagging in:

```python
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:

```python
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(ivec + 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.

```python
import streql


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

```python
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; to do so, you should use a key derivation algorithm, such as PBKDF2[6]. A function to derive a key from a passphrase will also need to store the salt that goes with the passphrase. PBKDf2 takes three arguments: the passphrase, the salt, and the number of iterations to run through. The currently recommended minimum number of iterations in 16384; this is a sensible default for programs using PBKDF2.

What is a salt? A salt is a randomly generated value used to make sure the output of two runs of PBKDF2 are unique for the same passphrase. Generally, this should be a minimum of 16 bytes (128-bits).

Here are two functions to generate a random salt and generate a secret key from PBKDF2:

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

```python
import pbkdf2
def generate_salt(salt_len):
    """Generate a salt for use with PBKDF2."""
    return RNG.new().read(salt_len)


def password_key(passphrase, salt=None):
    """Generate a key from a passphrase. Returns the tuple (salt, key)."""
    if salt is None:
        salt = generate_salt(16)
    passkey = pbkdf2.PBKDF2(passphrase, salt, iterations=16384).read(KEYSIZE)
    return salt, passkey
```

Keep in mind that the salt, while a public and non-secret value, must be present to recover the key. To generate a new key, pass `None` as the salt value, and a random salt will be generated. To recover the same key from the passphrase, the salt must be provided (and it must be the same salt generated when the passphrase key is generated). As an example, the salt could be provided as the first `len(salt)` bytes of the ciphertext.

That should cover the basics of block cipher encryption. We've gone over key generation, padding, and encryption / decryption. This code has been packaged up in the example source directory as `secretkey`.