# Cryptography with python3

## Why Cryptography ?

- confidentiality
- data integrity
- entity authentication

## Types of ciphers

- Symmetric
  - Stream ciphers (PSK)
    - XOR
    - ARC4
  - Block ciphers
    - AES
    - DES
    - DES3
    - CAST5
    - Blowfish
- Asymmetric
  - RSA
  - DSA
  - ELGamal
- Digest
  - MD5
  - SHA1
  - SHA224
  - SHA256
  - SHA384
  - SHA512
  - HMAC

```bash
pip3 install pycrypto
```

# Digest

*hashlib* is one of the modules for calculating digest on **binary** data.

```Python
import hashlib
>>> hashlib.algorithms_guaranteed
{'sha3_256', 'md5', 'blake2b', 'sha224', 'sha256', 'sha384', 'sha512', 'sha1', 'sha3_22
```

## MD5

```Python
>>> d = hashlib.md5('peyman'.encode())
>>> d.hexdigest()
'63143b65c08954aba4763ed55365f39f'
>>> d.digest()
b'c\x14;e\xc0\x89T\xab\xa4v>\xd5Se\xf3\x9f'
>>> d.digest().hex()
'63143b65c08954aba4763ed55365f39f'
>>> len(d.digest())
16
>>> bin(int.from_bytes(d.digest(), byteorder='big'))[2:].zfill(128)
'01100011000101000011101101100101110000001000100101010100101010111010010001110110001111
```

## SHA1

```Python
>>> d = hashlib.sha1('peyman'.encode())
>>> d.hexdigest()
'cf1949bdcb6fc268196eeca78d9f46b1e471f808'
>>> len(d.digest())
20
```

## SHA512

```python
>>> d = hashlib.sha512('peyman'.encode())
>>> d.hexdigest()
'd8b486c8688b48a3d877b608c9b4cba515c2cb2aa23834e067138757bba7bf9dd4f28d50ecc6ea2e3c6f53
>>> len(d.digest())
64
```

### HMAC

Any hash function may be used for HMAC.

```python
>>> from Crypto.Hash import MD5
>>> from Crypto.Hash import HMAC
>>> key = 'foobar'
>>> msg = 'The req queen is crying'
>>> HMAC.new(key.encode(), msg.encode(), digestmod=MD5).hexdigest()
'd5e3e6c2dcfbe910ff1a5982b5b2f0e4'
```

## Stream Ciphers

- Stream ciphers operate on data streams, one byte at a time.

### ARC4

```python
>>> from Crypto.Cipher import ARC4
>>> key = 'I am a cat'
>>> arc4 = ARC4.new(key)
>>> cipher = arc4.encrypt('The quick brown fox jumps over the lazy dog')
>>> cipher
b'\xc2\xd9\x8d<\xdby\r\xd6\x03\xde\x9e\x8e\x94\xd1\x8b=U\xf8\xc7R\xe5 \xc9\x8c\xc4"\xed
>>> arc4 = ARC4.new(key)
>>> arc4.decrypt(cipher)
b'The quick brown fox jumps over the lazy dog'
```

## Block Ciphers

- Block ciphers operate on blocks of data, typically 16 bytes at a time.

- data should be padded out to fit the block size.
- A common padding scheme is to use `0x80` as the first byte of padding, with `0x00` bytes filling out the rest of the padding.

## Block cipher modes

- **CBC** (cipher block chaining)
- **CTR** (counter)
- **CFB** (cipher feedback)
- **ECB** (electronic codebook) *insecure*

**CBC** is the most secure mode. Cipher block chaining works by XORing the previous block of ciphertext with the current block. for the first block there is no previous block to be XOR'd with, so a predefined block named **initialization vector** must be used.

## Using openssl to generate random strings

```Bash
$ openssl rand -base64 16
```

## AES

Python

```python
from Crypto.Random.OSRNG import posix as RNG
from Crypto.Cipher import AES

def pad_data(data):
    if type(data) is str:
        data = data.encode()
    if len(data) % AES.block_size == 0:
        return data
    n = AES.block_size - (len(data) % AES.block_size) - 1
    data += b'\x80'
    data += b'\x00' * n
    return data

def unpad_data(data):
    data = data.rstrip(b'\x00')
    return data[:-1]

def keygen(size=AES.block_size):
    return RNG.new().read(size)

def encrypt(data, key):
    data = pad_data(data)
    iv = keygen()
    aes = AES.new(key, AES.MODE_CBC, iv)
    cipher = aes.encrypt(data)
    return iv + cipher

def decrypt(cipher, key):
    iv = cipher[:AES.block_size]
    cipher_stripped = cipher[AES.block_size:]
    aes = AES.new(key, AES.MODE_CBC, iv)
    data = aes.decrypt(cipher_stripped)
    return unpad_data(data)

message = b'Hello everybody. It is a message.'
cipher = encrypt(message, b'asderfvbgtfgfdfg')
print(cipher)
print(decrypt(cipher,'asderfvbgtfgfdfg'))
```



## Base64

```python
>>> import base64
>>> b1 = b'\x128\x01\x00\x15'
>>> c = base64.b64encode(b1)
>>> c
b'EjgBABU='
>>> b2 = base64.b64decode(c)
>>> b2
b'\x128\x01\x00\x15'
>>> c.decode()
'EjgBABU='
```

## RSA

```python
>>> key = RSA.generate(1024)
>>> p = key.publickey()
>>> c = p.encrypt(b'The red queen is crying', 32)
>>> c
(b"9$\xb6\x04\x9cS/QL\xf6\xa7T\xc8Ce8O\x1dp\xbdu\xba\xd5is8\xb3!G\xb5\x9d\xcdK\x89\x95
>>> key.decrypt(c)
b'The red queen is crying'
```