python-camellia Documentation

Release 1.0

Simon Biewald

Oct 04, 2020

Contents

This is the documentation of python-camellia, a cryptographic library implementing the Camellia cipher in python.

Documentation | Source | Issue tracker

```
>>> import camellia
>>> plain = b"This is a text. "
>>> c1 = camellia.CamelliaCipher(key=b'16 byte long key', IV=b'16 byte iv. abcd',␣
↪mode=camellia.MODE_CBC)
>>> encrypted = c1.encrypt(plain)
>>> c2 = camellia.CamelliaCipher(key=b'16 byte long key', IV=b'16 byte iv. abcd',␣
↪mode=camellia.MODE_CBC)
>>> c2.decrypt(encrypted)
b'This is a text. '
```

Because it's build direct on top of the reference implementation, the python-camellia library provides direct access to extreme low-level functions like *Camellia-Ekeygen* but also provides a nearly PEP-272-compliant cryptographic interface. This semi low-level interface supports encryption (and decryption) in ECB, CBC, CFB, OFB and CTR modes of operation.

See the installation instructions for details regarding installation.

This software contains encryption algorithms, thus it may be restricted by law in some countries.

CHAPTER 1

Tree of contents

1.1 Installation

Install with pip:

```
$ pip install python-camellia
$ # Or:
$ python -m pip install python-camellia
```

1.1.1 Notes on the C extension

The camellia implementation is written in C, it is glued to Python using cffi. *pip* tries to automatically install prebuilt packages. Those are available for x86 and x64 Windows, recent MacOS (x64 only) and Linux. Additionally prebuilt are available for Linux for ARMv8 (aarch64), z/Architecture (s390x) and 64-bit PowerPC (ppc64le).

When those prebuilt packages are not available, the C code is compiled at installation. In this case a C compiler is required (usually gcc on Linux, XCode command line tools on MacOS, Visual Studio on Windows).

1.1.2 List of dependencies

Dependencies are automatically installed during installation.

- pep272-encryption providing block cipher modes
- cffi

1.2 API

Warning: The documentations assumes you know the risks of using cryptography. This library is low level with all benefits and dangers.

Here be dragons!

1.2.1 The *new* constructor

camellia.**new**(*key*, *mode*, *IV=None*, ***kwargs*)
Create an "CamelliaCipher" object.

Parameters

- **key** (*bytes*) – The key for encrytion/decryption. Must be 16/24/32 in length.
- **mode** (*int, one of MODE_* constants*) – Mode of operation.
- **IV** (*bytes*) – Initialization vector for CBC/CFB/OFB blockcipher modes of operation, must be 16 bytes in length.
- **counter** (*callable*) – Counter for CTR blockcipher mode of operation. Each call must return 16 bytes.

Returns CamelliaCipher

Raises ValueError, NotImplementedError

1.2.2 Modes of operation

camellia.**MODE_ECB = 1**
ECB mode of operation

camellia.**MODE_CBC = 2**
CBC mode of operation

camellia.**MODE_CFB = 3**
CFB mode of operation

camellia.**MODE_OFB = 5**
OFB mode of operation

camellia.**MODE_CTR = 6**
CTR mode of operation

1.2.3 The *CamelliaCipher* class

class camellia.**CamelliaCipher**(*key*, *mode*, ***kwargs*)
The CamelliaCipher object.

encrypt(*string*)
Encrypt data with the key and the parameters set at initialization.

The cipher object is stateful; encryption of a long block of data can be broken up in two or more calls to *encrypt()*. That is, the statement:

```
>>> c.encrypt(a) + c.encrypt(b)
```

is always equivalent to:

```
>>> c.encrypt(a+b)
```

That also means that you cannot reuse an object for encrypting or decrypting other data with the same key.

This function does not perform any padding.

- For *MODE_ECB*, *MODE_CBC* *string* length (in bytes) must be a multiple of *block_size*.
- For *MODE_CFB*, *string* length (in bytes) must be a multiple of *segment_size*/8.
- For *MODE_CTR* and *MODE_OFB*, *string* can be of any length.

Parameters **string** (*bytes*) – The piece of data to encrypt.

Raises

- **ValueError** – When a mode of operation has be requested this code cannot handle.
- **ValueError** – When len(string) has a wrong length, as described above.
- **TypeError** – When the counter callable in CTR returns data with the wrong length.

Returns The encrypted data, as a byte string. It is as long as *string*.

Return type bytes

decrypt(*string*)

Decrypt data with the key and the parameters set at initialization.

The cipher object is stateful; decryption of a long block of data can be broken up in two or more calls to *decrypt()*. That is, the statement:

```
>>> c.decrypt(a) + c.decrypt(b)
```

is always equivalent to:

```
>>> c.decrypt(a+b)
```

That also means that you cannot reuse an object for encrypting or decrypting other data with the same key.

This function does not perform any padding.

- For *MODE_ECB*, *MODE_CBC* *string* length (in bytes) must be a multiple of *block_size*.
- For *MODE_CFB*, *string* length (in bytes) must be a multiple of *segment_size*/8.
- For *MODE_CTR* and *MODE_OFB*, *string* can be of any length.

Parameters **string** (*bytes*) – The piece of data to decrypt.

Raises

- **ValueError** – When a mode of operation has be requested this code cannot handle.
- **ValueError** – When len(string) has a wrong length, as described above.
- **TypeError** – When the counter in CTR returns data of the wrong length.

Returns The decrypted data, as a byte string. It is as long as *string*.

Return type bytes

block_size = 16
block size of the camellia cipher

decrypt(*string*)
Decrypt data with the key and the parameters set at initialization.

The cipher object is stateful; decryption of a long block of data can be broken up in two or more calls to *decrypt()*. That is, the statement:

```
>>> c.decrypt(a) + c.decrypt(b)
```

is always equivalent to:

```
>>> c.decrypt(a+b)
```

That also means that you cannot reuse an object for encrypting or decrypting other data with the same key.

This function does not perform any padding.

- For *MODE_ECB*, *MODE_CBC* *string* length (in bytes) must be a multiple of *block_size*.
- For *MODE_CFB*, *string* length (in bytes) must be a multiple of *segment_size*/8.
- For *MODE_CTR* and *MODE_OFB*, *string* can be of any length.

Parameters **string** (*bytes*) – The piece of data to decrypt.

Raises

- **ValueError** – When a mode of operation has be requested this code cannot handle.
- **ValueError** – When len(string) has a wrong length, as described above.
- **TypeError** – When the counter in CTR returns data of the wrong length.

Returns The decrypted data, as a byte string. It is as long as *string*.

Return type bytes

decrypt_block(*key*, *block*, ***kwargs*)
Decrypt a single block with camellia.

encrypt(*string*)
Encrypt data with the key and the parameters set at initialization.

The cipher object is stateful; encryption of a long block of data can be broken up in two or more calls to *encrypt()*. That is, the statement:

```
>>> c.encrypt(a) + c.encrypt(b)
```

is always equivalent to:

```
>>> c.encrypt(a+b)
```

That also means that you cannot reuse an object for encrypting or decrypting other data with the same key.

This function does not perform any padding.

- For *MODE_ECB*, *MODE_CBC* *string* length (in bytes) must be a multiple of *block_size*.
- For *MODE_CFB*, *string* length (in bytes) must be a multiple of *segment_size*/8.
- For *MODE_CTR* and *MODE_OFB*, *string* can be of any length.

Parameters **string** (*bytes*) – The piece of data to encrypt.

Raises

- **ValueError** – When a mode of operation has be requested this code cannot handle.
- **ValueError** – When len(string) has a wrong length, as described above.
- **TypeError** – When the counter callable in CTR returns data with the wrong length.

Returns The encrypted data, as a byte string. It is as long as *string*.

Return type bytes

encrypt_block(*key*, *block*, ***kwargs*)
Encrypt a single block with camellia.

1.2.4 Low-level camellia functions

camellia.**Camellia_Ekeygen**(*rawKey*)
Make a keytable from a key.

Parameters **rawKey** (*bytes*) – raw encryption key, 128, 192 or 256 bits long

Returns keytable

camellia.**Camellia_Encrypt**(*keyLength*, *keytable*, *plainText*)
Encrypt a plaintext block by given arguments.

Parameters

- **keyLength** – key length (128, 192 or 256 bits
- **keytable** (*list*) – keytable returned by Camellia_Ekeygen
- **plainText** (*bytes*) – one plaintext block to encrypt (16 bytes in length)

Returns ciphertext block

camellia.**Camellia_Decrypt**(*keyLength*, *keytable*, *cipherText*)
Decrypt a plaintext block by given arguments.

Parameters

- **keyLength** – key length (128, 192 or 256 bits)
- **keytable** (*list*) – keytable returned by Camellia_Ekeygen
- **cipherText** (*bytes*) – one cipher block to decrypt (16 bytes in length)

Returns plaintext block

1.3 Examples

1.3.1 Authenticated encryption with password

Below is the source for a command line tool that can be used to encrypt and decrypt files with a password. It derives key from a user supplied password, uses Camellia with a 256-bit key in CBC mode and uses HMAC-SHA512 to authenticate the cipher text. The example is written for Python 3.5 or newer.

```python
import base64
import getpass
import hashlib
import hmac
import os
import sys

import camellia

HMAC_ALGO = "sha512"

PBKDF_ROUNDS = 10000000  # Larger = better but slower
PBKDF_HASH = "sha512"


def _print_usage():
    print("Usage: {} --encrypt|--decrypt INFILE OUTFILE".format(sys.argv[0]))


def _pad(data):
    byte_and_len = camellia.block_size - len(data) % camellia.block_size
    return data + bytes([byte_and_len] * byte_and_len)


def _unpad(data):
    return data[0:-data[-1]]


def encrypt(password: str, plaintext: bytes) -> str:
    salt = os.urandom(16)  # Random salt each time
    # Derive key from password, to compensate weaker passwords
    key = hashlib.pbkdf2_hmac(PBKDF_HASH, password.encode(), salt,
                              PBKDF_ROUNDS, dklen=64)
    # Use individual keys for encryption and authentication
    key_encryption, key_authentication = key[:32], key[32:]

    iv = os.urandom(16)  # Random IV, this is important
    encrypter = camellia.new(key_encryption, camellia.MODE_CBC, IV=iv)

    # The data is padded with PKCS#5
    cipher_text = encrypter.encrypt(_pad(plaintext))

    # Authentication tag
    mac = hmac.new(key_authentication, iv + cipher_text, HMAC_ALGO).digest()

    # Rounds are serialized to potentially increase it for new files
    return "{}.{}.{}.{}".format(
        base64.b64encode(salt).decode(),
        PBKDF_ROUNDS,
        base64.b64encode(iv + cipher_text).decode(),
        base64.b64encode(mac).decode()
    )


def decrypt(password: str, encrypted: str) -> bytes:
    encoded_salt, rounds, encoded_iv_cipher, encoded_mac = encrypted.split(".")
    rounds = int(rounds)
```

(continues on next page)

(continued from previous page)

```
    # Generate key
    key = hashlib.pbkdf2_hmac(PBKDF_HASH, password.encode(),
                              base64.b64decode(encoded_salt),
                              rounds, dklen=64)

    key_encryption, key_authentication = key[:32], key[32:]

    iv_cipher = base64.b64decode(encoded_iv_cipher)

    # Compare in time-safe manner, to prevent an attacker learning
    # about the newly computed MAC.
    if not hmac.compare_digest(hmac.new(key_authentication,
                                        iv_cipher, HMAC_ALGO).digest(),
                               base64.b64decode(encoded_mac)):
        raise ValueError("mac does not match, invalid password or data")

    iv, cipher_text = iv_cipher[:16], iv_cipher[16:]

    decrypter = camellia.new(key_encryption, mode=camellia.MODE_CBC, IV=iv)

    # Decrypt and remove padding
    return _unpad(decrypter.decrypt(cipher_text))


if __name__ == "__main__":
    if len(sys.argv) != 4:
        _print_usage()
        exit(1)

    if not os.path.isfile(sys.argv[2]):
        print("Not found: {}".format(sys.argv[2]))
        exit(2)

    password = getpass.getpass()

    try:
        if sys.argv[1] == "--encrypt":
            with open(sys.argv[2], 'rb') as infile:
                with open(sys.argv[3], 'wt') as outfile:
                    outfile.write(encrypt(password, infile.read()))
        elif sys.argv[1] == "--decrypt":
            with open(sys.argv[2], 'rt') as infile:
                with open(sys.argv[3], 'wb') as outfile:
                    outfile.write(decrypt(password, infile.read()))
        else:
            _print_usage()
            exit(1)
    except (IOError, ValueError) as e:
        print(e)
        exit(4)
```

1.4 Changelog of python-camellia

1.4.1 1.1.0 - TBD

New

- Add PEP-484 type hints

Changed

- Adapt Semantic Versioning
- The C extension is directly build using setuptools, this allows ABI3 wheels for multiple Python versions
- Documentation is at Readthedocs
- When safe, do not create ffi objects, but directly pass *bytes* to cffi. It is safe if it replaces a *const char**, or it is freshly created for exactly that purpose.
- ECB and CBC modes of operation are directly implemented in C, resulting in a speed increase of about 20% on CPython.

1.4.2 1.0 - 2018-05-11

New

Changed

- The "normal" camellia version is used instead of the mini or reference version.
- Camellia is now loaded using CFFI. This improves speed and avoids shipped DLLs. It's better than the self-made-on-first-use compilation, faster and less error-prone.
- Supports all standart modes of operation (ECB, CBC, CFB, OFB, CTR)
- Electronic code book mode of operation is not implicit default anymore.
- Now camellia.Camellia_Ekeygen returns a list instead of an CFFI array.

1.4.3 0.1.1 - 2015-09-05

New

- More metadata on PyPi

Changed

1.4.4 0.1 - 2015-08-30

- Initial release

1.5 Licenses

python-camellia is under two licenses:

- The Python code is MIT licensed:

```
Copyright (c) 2015 Simon Biewald

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.
```

- The C code - containing the official camellia engine by NTT - is 2-Clause-BSD licensed:

```
 Copyright (c) 2006,2007
 NTT (Nippon Telegraph and Telephone Corporation) . All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
  notice, this list of conditions and the following disclaimer as
  the first lines of this file unmodified.
2. Redistributions in binary form must reproduce the above copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY NTT ``AS IS'' AND ANY EXPRESS OR
IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
IN NO EVENT SHALL NTT BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

Index