bcl

Release 3.0.0

Nth Party, Ltd.

Oct 31, 2023

CONTENTS

Python library that provides a simple interface for symmetric (*i.e.*, secret-key) and asymmetric (*i.e.*, public-key) encryption/decryption primitives.

CHAPTER

ONE

PURPOSE

This library provides simple and straightforward methods for symmetric (*i.e.*, secret-key) and asymmetric (*i.e.*, public-key) cryptographic encryption and decryption capabilities. The library's interface is designed for ease of use and therefore hides from users some of the flexibilities and performance trade-offs that can be leveraged via direct use of the underlying cryptographic libraries.

The library's name is a reference to boron trichloride, as it is a wrapper and binding for a limited set of capabilities found in libsodium. However, it can also be an acronym for *basic cryptographic library*.

CHAPTER

TWO

INSTALLATION AND USAGE

This library is available as a package on PyPI:

```
python -m pip install bcl
```

The library can be imported in the usual ways:

```
import bcl
from bcl import *
```

2.1 Examples

This library provides concise methods for implementing symmetric encryption workflows:

```
>>> from bcl import symmetric
>>> s = symmetric.secret() # Generate a secret key.
>>> c = symmetric.encrypt(s, 'abc'.encode())
>>> symmetric.decrypt(s, c).decode('utf-8')
'abc'
```

Asymmetric encryption workflows are also supported:

```
>>> from bcl import asymmetric
>>> s = asymmetric.secret() # Generate a secret key.
>>> p = asymmetric.public(s) # Generate a corresponding public key.
>>> c = asymmetric.encrypt(p, 'abc'.encode())
>>> asymmetric.decrypt(s, c).decode('utf-8')
'abc'
```

This library also provides a number of classes for representing keys (secret and public), nonces, plaintexts, and ciphertexts. All methods expect and return instances of the appropriate classes:

```
>>> from bcl import secret, public, cipher
>>> s = asymmetric.secret()
>>> isinstance(s, secret)
True
>>> p = asymmetric.public(s)
>>> isinstance(p, public)
True
>>> c = symmetric.encrypt(s, 'abc'.encode())
```

(continues on next page)

(continued from previous page)

```
>>> type(c)
<class 'bcl.bcl.cipher'>
>>> symmetric.decrypt(bytes(s), c)
Traceback (most recent call last):
  ...
TypeError: can only decrypt using a symmetric secret key
>>> symmetric.decrypt(s, bytes(c))
Traceback (most recent call last):
  ...
TypeError: can only decrypt a ciphertext
```

Furthermore, the above classes are derived from `bytes`, so all methods and other operators supported by `bytes` objects are supported:

```
>>> p.hex()
'0be9cece7fee92809908bd14666eab96b77deebb488c738445d842a6613b7b48'
```

In addition, Base64 conversion methods are included for all of the above classes to support concise encoding and decoding of objects:

```
>>> p.to_base64()
'C+nOzn/ukoCZCL0UZm6rlrd97rtIjHOERdhCpmE7e0g='
>>> b = 'C+nOzn/ukoCZCL0UZm6rlrd97rtIjHOERdhCpmE7e0g='
>>> type(public.from_base64(b))
<class 'bcl.bcl.public'>
```

CHAPTER

THREE

DEVELOPMENT, BUILD, AND MANUAL INSTALLATION INSTRUCTIONS

All development and installation dependencies are managed using setuptools and are fully specified in `setup.cfg`. The `extras_require` option is used to specify optional requirements for various development tasks. This makes it possible to specify additional options (such as `docs`, `lint`, and so on) when performing installation using pip (assuming that the library has already been built successfully):

```
python -m pip install .[docs,lint]
```

3.1 Building from Source

The library can be built manually from source **within Linux and macOS** using the sequence of commands below:

```
python -m pip install .[build]
python -m build --sdist --wheel .
```

Developing the library further in a local environment and/or building the library from source requires libsodium. The step `python -m build --sdist --wheel .` in the above attempts to automatically locate a copy of the libsodium source archive `src/bcl/libsodium.tar.gz`. If the archive corresponding to the operating system is not found, the build process attempts to download it. To support building offline, it is necessary to first download the appropriate libsodium archive to its designated location:

```
wget -O src/bcl/libsodium.tar.gz https://github.com/jedisct1/libsodium/releases/download/
↪1.0.18-RELEASE/libsodium-1.0.18.tar.gz
```

The process for building manually from source within a Windows environment is not currently documented, but an example of one sequence of steps can be found in the Windows job entry within the GitHub Actions workflow defined in the file `.github/workflows/lint-test-cover-docs-build-upload.yml`.

3.2 Preparation for Local Development

Before *documentation can be generated* or *tests can be executed*, it is necessary to *run the build process* and then to use the command below to move the compiled libsodium shared/dynamic library file into its designated location (so that the module file `src/bcl/bcl.py` is able to import it):

```
cp build/lib*/bcl/_sodium.py src/bcl
```

3.3 Manual Installation

Once the package is *built*, it can be installed manually using the command below:

```
python -m pip install -f dist . --upgrade
```

3.4 Documentation

Once the libsodium shared library file is compiled and moved into its designated location (as described in *the relevant subsection above*), the documentation can be generated automatically from the source files using Sphinx:

```
python -m pip install .[docs]
cd docs
sphinx-apidoc -f -E --templatedir=_templates -o _source .. ../src/bcl/_sodium_build.py &&
↪ make html
```

3.5 Testing and Conventions

Before unit tests can be executed, it is first necessary to prepare for local development by compiling and moving into its designated location the libsodium shared library file (as described in *the relevant subsection above*).

All unit tests are executed and their coverage is measured when using pytest (see `pyproject.toml` for configuration details):

```
python -m pip install .[test]
python -m pytest
```

Alternatively, all unit tests are included in the module itself and can be executed using doctest:

```
python src/bcl/bcl.py -v
```

Style conventions are enforced using Pylint:

```
python -m pip install .[lint]
python -m pylint src/bcl src/bcl/_sodium.tmpl src/bcl/_sodium_build.py --
↪disable=duplicate-code
```

3.6 Contributions

In order to contribute to the source code, open an issue or submit a pull request on the GitHub page for this library.

3.7 Versioning

The version number format for this library and the changes to the library associated with version number increments conform with Semantic Versioning 2.0.0.

3.8 Publishing

This library can be published as a package on PyPI by a package maintainer. First, install the dependencies required for packaging and publishing:

```
python -m pip install .[publish]
```

Ensure that the correct version number appears in `setup.cfg`, and that any links in this README document to the Read the Docs documentation of this package (or its dependencies) have appropriate version numbers. Also ensure that the Read the Docs project for this library has an automation rule that activates and sets as the default all tagged versions. Create and push a tag for this version (replacing `?.?.?` with the version number):

```
git tag ?.?.?
git push origin ?.?.?
```

Remove any old build/distribution files. Then, package the source into a distribution archive:

```
rm -rf build dist src/*.egg-info
python -m build --sdist .
```

Next, navigate to the appropriate GitHub Actions run of the workflow defined in `lint-test-cover-docs-build-upload.yml`. Click on the workflow and scroll down to the **Artifacts** panel. Download the archive files to the `dist` directory. Unzip all the archive files so that only the `*.whl` files remain:

```
cd dist && for i in `ls *.zip`; do unzip $i; done && rm *.zip && cd ..
```

Finally, upload the package distribution archive to PyPI:

```
python -m twine upload dist/*
```

3.8.1 bcl module

Python library that provides a simple interface for symmetric (*i.e.*, secret-key) and asymmetric (*i.e.*, public-key) encryption/decryption primitives.

This library exports a number of classes (derived from `bytes`) for representing keys, nonces, plaintexts, and ciphertexts. It also exports two classes *`symmetric`* and *`asymmetric`* that have only static methods (for key generation and encryption/decryption).

class bcl.bcl.**raw**

Bases: `bytes`

Wrapper class for a raw bytes-like object that represents a key, nonce, plaintext, or ciphertext. The derived classes *`secret`*, *`public`*, *`nonce`*, *`plain`*, and *`cipher`* all inherit the methods defined in this class.

```
>>> s = secret.from_base64('1P3mjNnadofjTUkzTmipYl+xdo9z/EaGLbWcJ8MAPBQ=')
>>> s.hex()
'd4fde68cd9da7687e34d49334e68a9625fb1768f73fc46862db59c27c3003c14'
>>> n = nonce.from_base64('JVN9IKBLZi3lEq/eDgkV+y6n4v7x2edI')
>>> c = symmetric.encrypt(s, 'abc'.encode(), n)
>>> c.to_base64()
'JVN9IKBLZi3lEq/eDgkV+y6n4v7x2edI9dvFXD+om1dHB6UUCt1y4BqrBw=='
```

classmethod from_base64(*s: str*) → *bcl.bcl.raw*
Convert Base64 UTF-8 string representation of a raw value.

to_base64() → str
Convert to equivalent Base64 UTF-8 string representation.

class bcl.bcl.**nonce**(*argument: Optional[Union[bytes, bytearray, int]] = None*)
Bases: *bcl.bcl.raw*

Wrapper class for a bytes-like object that represents a nonce.

```
>>> n = nonce()
>>> n = nonce(bytes(n))
>>> isinstance(n, nonce) and isinstance(n, bytes)
True
```

While the constructor works like the constructor for bytes-like objects in also accepting an integer argument, an instance can only have the exact length permitted for a nonce.

```
>>> nonce(nonce.length).hex()
'000000000000000000000000000000000000000000000000'
```

The constructor for this class checks that the supplied bytes-like object or integer argument satisfy the conditions for a valid nonce.

```
>>> nonce('abc')
Traceback (most recent call last):
  ...
TypeError: nonce constructor argument must be a bytes-like object or an integer
>>> try:
...     nonce(bytes([1, 2, 3]))
... except ValueError as e:
...     str(e) == 'nonce must have exactly '  + str(nonce.length) + ' bytes'
True
>>> try:
...     nonce(123)
... except ValueError as e:
...     str(e) == 'nonce must have exactly '  + str(nonce.length) + ' bytes'
True
```

length: **int** = **None**
Length (in number of bytes) of nonce instances.

static __new__(*cls*, *argument: Optional[Union[bytes, bytearray, int]] = None*) → *bcl.bcl.nonce*
Create a nonce object.

class bcl.bcl.**key**
Bases: *bcl.bcl.raw*

Wrapper class for a bytes-like object that represents a key. The derived classes *secret* and *public* inherit the methods defined in this class.

Any *key* objects (including instances of classes derived from *key*) have a few features and behaviors that distinguish them from bytes-like objects.

- Comparison of keys (using the built-in == and != operators via the *__eq__* and *__ne__* methods) is performed in constant time.
- Keys of different types are not equivalent even if their binary representation is identical.

```
>>> b = 'd6vGTIjbxZyMolCW+/p1QFF5hjsYC5Q4x07s+RIMKK8='
>>> secret.from_base64(b) == public.from_base64(b)
False
>>> secret.from_base64(b) != public.from_base64(b)
True
```

- Consistent with the above property, keys having different classes are distinct when used as keys or items within containers.

```
>>> b = 'd6vGTIjbxZyMolCW+/p1QFF5hjsYC5Q4x07s+RIMKK8='
>>> len({secret.from_base64(b), public.from_base64(b)})
2
```

__eq__(*other:* bcl.bcl.key) → bool

Compare two keys (including their subclass). The portion of the method that compares byte values runs in constant time.

```
>>> key(bytes([0] * 32)) == key(bytes([1] * 32))
False
>>> key(bytes([1] * 32)) == key(bytes([1] * 32))
True
>>> secret(bytes([0] * 32)) == public(bytes([0] * 32))
False
```

__ne__(*other:* bcl.bcl.key) → bool

Compare two keys (including their subclass). The portion of the method that compares byte values runs in constant time.

```
>>> key(bytes([0] * 32)) != key(bytes([1] * 32))
True
>>> key(bytes([1] * 32)) != key(bytes([1] * 32))
False
>>> secret(bytes([0] * 32)) != public(bytes([0] * 32))
True
```

class bcl.bcl.**secret**(*argument: Optional[Union[bytes, bytearray, int]] = None*)

Bases: *bcl.bcl.key*

Wrapper class for a bytes-like object that represents a secret key. The constructor for this class can be used to generate an instance of a secret key or to convert a bytes-like object into a secret key.

```
>>> s = secret()
>>> s = secret(bytes(s))
>>> isinstance(s, secret) and isinstance(s, key)and isinstance(s, bytes)
True
```

While the constructor works like the constructor for bytes-like objects in also accepting an integer argument, an instance can only have the exact length permitted for a secret key.

```
>>> secret(secret.length).hex()
'0000000000000000000000000000000000000000000000000000000000000000'
```

The constructor for this class checks that the supplied bytes-like object or integer argument satisfy the conditions for a valid secret key.

```
>>> secret('abc')
Traceback (most recent call last):
  ...
TypeError: secret key constructor argument must be a bytes-like object or an integer
>>> try:
...     secret(bytes([1, 2, 3]))
... except ValueError as e:
...     str(e) == 'secret key must have exactly '  + str(secret.length) + ' bytes'
True
>>> try:
...     secret(123)
... except ValueError as e:
...     str(e) == 'secret key must have exactly '  + str(secret.length) + ' bytes'
True
```

The methods *symmetric.encrypt*, *symmetric.decrypt*, and *asymmetric.decrypt* only accept key parameters that are objects of this class.

length: *int* **= None**
Length (in number of bytes) of secret key instances.

static __new__(*cls*, *argument: Optional[Union[bytes, bytearray, int]] = None*) → *bcl.bcl.secret*
Create a secret key object.

class bcl.bcl.**public**(*argument: Optional[Union[bytes, bytearray, int]] = None*)
Bases: *bcl.bcl.key*

Wrapper class for a bytes-like object that represents a public key. The constructor for this class can be used to generate an instance of a public key or to convert a bytes-like object into a public key.

```
>>> p = public()
>>> p = public(bytes(p))
>>> isinstance(p, public) and isinstance(p, key)and isinstance(p, bytes)
True
```

While the constructor works like the constructor for bytes-like objects in also accepting an integer argument, an instance can only have the exact length permitted for a public key.

```
>>> public(public.length).hex()
'0000000000000000000000000000000000000000000000000000000000000000'
```

The constructor for this class checks that the supplied bytes-like object or integer argument satisfy the conditions for a valid public key.

```
>>> public('abc')
Traceback (most recent call last):
  ...
```

(continues on next page)

(continued from previous page)

```
TypeError: public key constructor argument must be a bytes-like object or an integer
>>> try:
...     public(bytes([1, 2, 3]))
... except ValueError as e:
...     length = _CRYPTO_BOX_PUBLICKEYBYTES
...     str(e) == 'public key must have exactly '  + str(length) + ' bytes'
True
>>> try:
...     public(123)
... except ValueError as e:
...     length = _CRYPTO_BOX_PUBLICKEYBYTES
...     str(e) == 'public key must have exactly '  + str(length) + ' bytes'
True
```

The method *asymmetric.encrypt* only accepts key parameters that are objects of this class.

length: *int* **= None**
Length (in number of bytes) of public key instances.

static __new__(*cls*, *argument: Optional[Union[bytes, bytearray, int]] = None*) → *bcl.bcl.public*
Create a public key object.

class bcl.bcl.**plain**
Bases: *bcl.bcl.raw*

Wrapper class for a bytes-like object that represents a plaintext.

```
>>> x = plain(os.urandom(1024))
>>> x == plain.from_base64(x.to_base64())
True
```

The methods *symmetric.decrypt* and *asymmetric.decrypt* return objects of this class.

class bcl.bcl.**cipher**
Bases: *bcl.bcl.raw*

Wrapper class for a bytes-like object that represents a ciphertext.

```
>>> c = cipher(os.urandom(1024))
>>> c == cipher.from_base64(c.to_base64())
True
```

The methods *symmetric.encrypt* and *asymmetric.encrypt* return objects of this class, and the methods *symmetric.decrypt* and *asymmetric.decrypt* can only be applied to objects of this class.

class bcl.bcl.**symmetric**
Bases: object

Symmetric (*i.e.*, secret-key) encryption/decryption primitives. This class encapsulates only static methods and should not be instantiated.

```
>>> x = 'abc'.encode()
>>> s = symmetric.secret()
>>> isinstance(s, key) and isinstance(s, secret)
True
>>> s == secret.from_base64(s.to_base64())
```

(continues on next page)

(continued from previous page)

```
True
>>> c = symmetric.encrypt(s, x)
>>> isinstance(c, raw) and isinstance(c, cipher)
True
>>> c == cipher.from_base64(c.to_base64())
True
>>> symmetric.decrypt(s, c) == x
True
>>> isinstance(symmetric.decrypt(s, c), plain)
True
```

Encryption is non-deterministic if no *nonce* parameter is supplied.

```
>>> symmetric.encrypt(s, x) == symmetric.encrypt(s, x)
False
```

Deterministic encryption is possible by supplying a *nonce* parameter.

```
>>> n = nonce()
>>> symmetric.encrypt(s, x, n) == symmetric.encrypt(s, x, n)
True
```

static secret() → *bcl.bcl.secret*

Generate a *secret* key.

static encrypt(*secret_key:* bcl.bcl.secret, *plaintext: Union[*bcl.bcl.plain*, bytes, bytearray]*, *noncetext: Optional[*bcl.bcl.nonce*] = None*) → *bcl.bcl.cipher*

Encrypt a plaintext (a bytes-like object) using the supplied *secret* key (and an optional *nonce*, if applicable).

```
>>> m = plain(bytes([1, 2, 3]))
>>> s = symmetric.secret()
>>> c = symmetric.encrypt(s, m)
>>> m == symmetric.decrypt(s, c)
True
```

All parameters supplied to this method must have appropriate types.

```
>>> c = symmetric.encrypt(bytes([0, 0, 0]), m)
Traceback (most recent call last):
  ...
TypeError: can only encrypt using a symmetric secret key
>>> c = symmetric.encrypt(s, 'abc')
Traceback (most recent call last):
  ...
TypeError: can only encrypt a plaintext object or bytes-like object
>>> c = symmetric.encrypt(s, m, bytes([0, 0, 0]))
Traceback (most recent call last):
  ...
TypeError: nonce parameter must be a nonce object
```

static decrypt(*secret_key:* bcl.bcl.secret, *ciphertext:* bcl.bcl.cipher) → *bcl.bcl.plain*

Decrypt a ciphertext (an instance of *cipher*) using the supplied *secret* key.

```
>>> m = plain(bytes([1, 2, 3]))
>>> s = symmetric.secret()
>>> c = symmetric.encrypt(s, m)
>>> m == symmetric.decrypt(s, c)
True
```

All parameters supplied to this method must have appropriate types.

```
>>> c = symmetric.decrypt(bytes([0, 0, 0]), m)
Traceback (most recent call last):
  ...
TypeError: can only decrypt using a symmetric secret key
>>> c = symmetric.decrypt(s, 'abc')
Traceback (most recent call last):
  ...
TypeError: can only decrypt a ciphertext
>>> symmetric.decrypt(s, cipher(c + bytes([0, 0, 0])))
Traceback (most recent call last):
  ...
RuntimeError: ciphertext failed verification
```

class bcl.bcl.**asymmetric**

Bases: object

Asymmetric (*i.e.*, public-key) encryption/decryption primitives. This class encapsulates only static methods and should not be instantiated.

```
>>> x = 'abc'.encode()
>>> s = asymmetric.secret()
>>> isinstance(s, key) and isinstance(s, secret)
True
>>> p = asymmetric.public(s)
>>> isinstance(p, key) and isinstance(p, public)
True
>>> p == public.from_base64(p.to_base64())
True
>>> c = asymmetric.encrypt(p, x)
>>> asymmetric.decrypt(s, c) == x
True
```

static secret() → *bcl.bcl.secret*

Generate a *secret* key.

```
>>> s = symmetric.secret()
>>> isinstance(s, key) and isinstance(s, secret)
True
```

static public(*secret_key:* bcl.bcl.secret) → *bcl.bcl.public*

Generate a *public* key using a *secret* key.

```
>>> s = asymmetric.secret()
>>> p = asymmetric.public(s)
>>> isinstance(p, key) and isinstance(p, public)
True
```

static encrypt(*public_key:* bcl.bcl.public, *plaintext: Union[*bcl.bcl.plain, *bytes*, *bytearray]*) → *bcl.bcl.cipher*

Encrypt a plaintext (any bytes-like object) using the supplied *public* key.

```
>>> m = plain(bytes([1, 2, 3]))
>>> s = asymmetric.secret()
>>> p = asymmetric.public(s)
>>> c = asymmetric.encrypt(p, m)
>>> m == asymmetric.decrypt(s, c)
True
```

All parameters supplied to this method must have appropriate types.

```
>>> c = asymmetric.encrypt(s, m)
Traceback (most recent call last):
  ...
TypeError: can only encrypt using a public key
>>> c = asymmetric.encrypt(p, 'abc')
Traceback (most recent call last):
  ...
TypeError: can only encrypt a plaintext object or bytes-like object
```

static decrypt(*secret_key:* bcl.bcl.secret, *ciphertext:* bcl.bcl.cipher) → *bcl.bcl.plain*

Decrypt a ciphertext (an instance of *cipher*) using the supplied *secret* key.

```
>>> m = plain(bytes([1, 2, 3]))
>>> s = asymmetric.secret()
>>> p = asymmetric.public(s)
>>> c = asymmetric.encrypt(p, m)
>>> m == asymmetric.decrypt(s, c)
True
```

All parameters supplied to this method must have appropriate types.

```
>>> c = asymmetric.decrypt(p, m)
Traceback (most recent call last):
  ...
TypeError: can only decrypt using an asymmetric secret key
>>> c = asymmetric.decrypt(s, 'abc')
Traceback (most recent call last):
  ...
TypeError: can only decrypt a ciphertext
>>> try:
...     asymmetric.decrypt(s, cipher(bytes([0])))
... except ValueError as e:
...     length = _CRYPTO_BOX_SEALBYTES
...     str(e) == 'asymmetric ciphertext must have at least '  + str(length) +
↪' bytes'
True
```

PYTHON MODULE INDEX

INDEX