PyNaCl Documentation

Release 0.1.0

Donald Stufft

October 27, 2013

CONTENTS

Contents:

CHAPTER ONE

PUBLIC KEY ENCRYPTION

Imagine Alice wants something valuable shipped to her. Because it's valuable, she wants to make sure it arrives securely (i.e. hasn't been opened or tampered with) and that it's not a forgery (i.e. it's actually from the sender she's expecting it to be from and nobody's pulling the old switcheroo)

One way she can do this is by providing the sender (let's call him Bob) with a high-security box of her choosing. She provides Bob with this box, and something else: a padlock, but a padlock without a key. Alice is keeping that key all to herself. Bob can put items in the box then put the padlock onto it, but once the padlock snaps shut, the box cannot be opened by anyone who doesn't have Alice's private key.

Here's the twist though, Bob also puts a padlock onto the box. This padlock uses a key Bob has published to the world, such that if you have one of Bob's keys, you know a box came from him because Bob's keys will open Bob's padlocks (let's imagine a world where padlocks cannot be forged even if you know the key). Bob then sends the box to Alice.

In order for Alice to open the box, she needs two keys: her private key that opens her own padlock, and Bob's well-known key. If Bob's key doesn't open the second padlock then Alice knows that this is not the box she was expecting from Bob, it's a forgery.

This bidirectional guarantee around identity is known as mutual authentication.

1.1 Example

The `Box` class uses the given public and private (secret) keys to derive a shared key, which is used with the nonce given to encrypt the given messages and decrypt the given ciphertexts. The same shared key will generated from both pairing of keys, so given two keypairs belonging to alice (pkalice, skalice) and bob(pkbob, skbob), the key derived from (pkalice, skbob) with equal that from (pkbob, skalice). This is how the system works:

```python
import nacl.utils
from nacl.public import PrivateKey, Box

# generate the private key which must be kept secret
skbob = PrivateKey.generate()

# the public key can be given to anyone wishing to send
# Bob an encrypted message
pkbob = skbob.public_key

# Alice does the same and then
#     sends her public key to Bob and Bob his public key to Alice
skalice = PrivateKey.generate()
pkalice = skalice.public_key

# Bob wishes to send Alice an encrypted message
# So Bob must make a Box with his private key and Alice's public key
bob_box = Box(skbob, pkalice)

# This is our message to send, it must be a bytestring as Box will
#   treat is as just a binary blob of data.
message = b"Kill all humans"

# This is a nonce, it *MUST* only be used once, but it is not considered
#   secret and can be transmitted or stored alongside the ciphertext. A
#   good source of nonce is just 24 random bytes.
nonce = nacl.utils.random(Box.NONCE_SIZE)

# Encrypt our message, it will be exactly 40 bytes longer than the original
#   message as it stores authentication information and nonce alongside it.
encrypted = bob_box.encrypt(message, nonce)

# Alice creates a second box with her private key to decrypt the message
alice_box = Box(skalice, pkbob)

# Decrypt our message, an exception will be raised if the encryption was
#   tampered with or there was otherwise an error.
plaintext = alice_box.decrypt(encrypted)
```

1.2 Reference

class `nacl.public.`**`PublicKey`**(*public_key*, *encoder=<class 'nacl.encoding.RawEncoder'>*)

The public key counterpart to an Curve25519 `nacl.public.PrivateKey` for encrypting messages.

Parameters

- **public_key** – [`bytes`] Encoded Curve25519 public key
- **encoder** – A class that is able to decode the *public_key*

Variables **SIZE** – The size that the public key is required to be

class `nacl.public.`**`PrivateKey`**(*private_key*, *encoder=<class 'nacl.encoding.RawEncoder'>*)

Private key for decrypting messages using the Curve25519 algorithm.

Warning: This **must** be protected and remain secret. Anyone who knows the value of your `PrivateKey` can decrypt any message encrypted by the corresponding `PublicKey`

Parameters

- **private_key** – The private key used to decrypt messages
- **encoder** – The encoder class used to decode the given keys

Variables **SIZE** – The size that the private key is required to be

classmethod **`generate`**()

Generates a random `PrivateKey` object

Return type `PrivateKey`

class `nacl.public.`**`Box`**(*private_key*, *public_key*)

The Box class boxes and unboxes messages between a pair of keys

The ciphertexts generated by `Box` include a 16 byte authenticator which is checked as part of the decryption. An invalid authenticator will cause the decrypt function to raise an exception. The authenticator is not a signature. Once you've decrypted the message you've demonstrated the ability to create arbitrary valid message, so messages you send are repudiable. For non-repudiable messages, sign them after encryption.

Parameters

- **private_key** – `PrivateKey` used to encrypt and decrypt messages
- **public_key** – `PublicKey` used to encrypt and decrypt messages

Variables **NONCE_SIZE** – The size that the nonce is required to be.

`decrypt` (*ciphertext*, *nonce=None*, *encoder=<class 'nacl.encoding.RawEncoder'>*)

Decrypts the ciphertext using the given nonce and returns the plaintext message.

Parameters

- **ciphertext** – [`bytes`] The encrypted message to decrypt
- **nonce** – [`bytes`] The nonce used when encrypting the ciphertext
- **encoder** – The encoder used to decode the ciphertext.

Return type [`bytes`]

`encrypt` (*plaintext*, *nonce*, *encoder=<class 'nacl.encoding.RawEncoder'>*)

Encrypts the plaintext message using the given *nonce* and returns the ciphertext encoded with the encoder.

Warning: It is **VITALLY** important that the nonce is a nonce, i.e. it is a number used only once for any given key. If you fail to do this, you compromise the privacy of the messages encrypted.

Parameters

- **plaintext** – [`bytes`] The plaintext message to encrypt
- **nonce** – [`bytes`] The nonce to use in the encryption
- **encoder** – The encoder to use to encode the ciphertext

Return type [`nacl.utils.EncryptedMessage`]

class `nacl.utils.`**`EncryptedMessage`**

A bytes subclass that holds a messaged that has been encrypted by a `SecretBox`.

`ciphertext`

The ciphertext contained within the `EncryptedMessage`.

`nonce`

The nonce used during the encryption of the `EncryptedMessage`.

CHAPTER

TWO

SECRET KEY ENCRYPTION

Secret key encryption is analogous to a safe. You can store something secret through it and anyone who has the key can open it and view the contents. `SecretBox` functions as just such a safe, and like any good safe any attempts to tamper with the contents is easily detected.

Secret Key Encryption allows you to store or transmit data over insecure channels without leaking the contents of that message, nor anything about it other than the length.

2.1 Example

```python
import nacl.secret
import nacl.utils

# This must be kept secret, this is the combination to your safe
key = nacl.utils.random(nacl.secret.SecretBox.KEY_SIZE)

# This is your safe, you can use it to encrypt or decrypt messages
box = nacl.secret.SecretBox(key)

# This is our message to send, it must be a bytestring as SecretBox will
#   treat is as just a binary blob of data.
message = b"The president will be exiting through the lower levels"

# This is a nonce, it *MUST* only be used once, but it is not considered
#   secret and can be transmitted or stored alongside the ciphertext. A
#   good source of nonce is just 24 random bytes.
nonce = nacl.utils.random(nacl.secret.SecretBox.NONCE_SIZE)

# Encrypt our message, it will be exactly 40 bytes longer than the original
#   message as it stores authentication information and nonce alongside it.
encrypted = box.encrypt(message, nonce)

# Decrypt our message, an exception will be raised if the encryption was
#   tampered with or there was otherwise an error.
plaintext = box.decrypt(encrypted)
```

2.2 Requirements

2.2.1 Key

The 32 bytes key given to `SecretBox` must be kept secret. It is the combination to your "safe" and anyone with this key will be able to decrypt the data, or encrypt new data.

2.2.2 Nonce

The 24 bytes nonce (Number used once) given to `encrypt()` and `decrypt()` must **NEVER** be reused for a particular key. Reusing the nonce means an attacker will have enough information to recover your secret key and encrypt or decrypt arbitrary messages. A nonce is not considered secret and may be freely transmitted or stored in plaintext alongside the ciphertext.

A nonce does not need to be random, nor does the method of generating them need to be secret. A nonce could simply be a counter incremented with each message encrypted.

Both the sender and the receiver should record every nonce both that they've used and they've received from the other. They should reject any message which reuses a nonce and they should make absolutely sure never to reuse a nonce. It is not enough to simply use a random value and hope that it's not being reused (simply generating random values would open up the system to a Birthday Attack).

One good method of generating nonces is for each person to pick a unique prefix, for example `b"p1"` and `b"p2"`. When each person generates a nonce they prefix it, so instead of `nacl.utils.random(24)` you'd do `b"p1" + nacl.utils.random(22)`. This prefix serves as a guarantee that no two messages from different people will inadvertently overlap nonces while in transit. They should still record every nonce they've personally used and every nonce they've received to prevent reuse or replays.

2.3 Reference

class `nacl.secret.`**`SecretBox`**(*key*, *encoder=<class 'nacl.encoding.RawEncoder'>*)

The SecretBox class encrypts and decrypts messages using the given secret key.

The ciphertexts generated by `Secretbox` include a 16 byte authenticator which is checked as part of the decryption. An invalid authenticator will cause the decrypt function to raise an exception. The authenticator is not a signature. Once you've decrypted the message you've demonstrated the ability to create arbitrary valid message, so messages you send are repudiable. For non-repudiable messages, sign them after encryption.

Parameters

- **key** – The secret key used to encrypt and decrypt messages
- **encoder** – The encoder class used to decode the given key

Variables

- **KEY_SIZE** – The size that the key is required to be.
- **NONCE_SIZE** – The size that the nonce is required to be.

`decrypt`(*ciphertext*, *nonce=None*, *encoder=<class 'nacl.encoding.RawEncoder'>*)

Decrypts the ciphertext using the given nonce and returns the plaintext message.

Parameters

- **ciphertext** – [`bytes`] The encrypted message to decrypt

- **nonce** – [`bytes`] The nonce used when encrypting the ciphertext
- **encoder** – The encoder used to decode the ciphertext.

Return type [`bytes`]

`encrypt` (*plaintext*, *nonce*, *encoder=<class 'nacl.encoding.RawEncoder'>*)

Encrypts the plaintext message using the given nonce and returns the ciphertext encoded with the encoder.

> **Warning:** It is **VITALLY** important that the nonce is a nonce, i.e. it is a number used only once for any given key. If you fail to do this, you compromise the privacy of the messages encrypted. Give your nonces a different prefix, or have one side use an odd counter and one an even counter. Just make sure they are different.

Parameters

- **plaintext** – [`bytes`] The plaintext message to encrypt
- **nonce** – [`bytes`] The nonce to use in the encryption
- **encoder** – The encoder to use to encode the ciphertext

Return type [`nacl.utils.EncryptedMessage`]

class `nacl.utils.`**`EncryptedMessage`**

A bytes subclass that holds a messaged that has been encrypted by a `SecretBox`.

`EncryptedMessage.`**`ciphertext`**

The ciphertext contained within the `EncryptedMessage`.

`EncryptedMessage.`**`nonce`**

The nonce used during the encryption of the `EncryptedMessage`.

2.4 Algorithm details

Encryption Salsa20 steam cipher

Authentication Poly1305 MAC

CHAPTER

THREE

DIGITAL SIGNATURES

You can use a digital signature for many of the same reasons that you might sign a paper document. A valid digital signature gives a recipient reason to believe that the message was created by a known sender such that they cannot deny sending it (authentication and non-repudiation) and that the message was not altered in transit (integrity).

Digital signatures allow you to publish a public key, and then you can use your private signing key to sign messages. Others who have your public key can then use it to validate that your messages are actually authentic.

3.1 Example

Signer's perspective (`SigningKey`)

```
import nacl.encoding
import nacl.signing

# Generate a new random signing key
signing_key = nacl.signing.SigningKey.generate()

# Sign a message with the signing key
signed = signing_key.sign("Attack at Dawn")

# Obtain the verify key for a given signing key
verify_key = signing_key.verify_key

# Serialize the verify key to send it to a third party
verify_key_hex = verify_key.encode(encoder=nacl.encoding.HexEncoder)
```

Verifier's perspective (`VerifyKey`)

```
import nacl.signing

# Create a VerifyKey object from a hex serialized public key
verify_key = nacl.signing.VerifyKey(verify_key_hex, encoder=nacl.encoding.HexEncoder)

# Check the validity of a message's signature
# Will raise nacl.signing.BadSignatureError if the signature check fails
verify_key.verify(signed)
```

3.2 Reference

class `nacl.signing.`**`SigningKey`**`(`*seed*, *encoder=<class 'nacl.encoding.RawEncoder'>*`)`

Private key for producing digital signatures using the Ed25519 algorithm.

Signing keys are produced from a 32-byte (256-bit) random seed value. This value can be passed into the `SigningKey` as a `bytes()` whose length is 32.

> **Warning:** This **must** be protected and remain secret. Anyone who knows the value of your `SigningKey` or it's seed can masquerade as you.

Parameters

- **seed** – [`bytes`] Random 32-byte value (i.e. private key)
- **encoder** – A class that is able to decode the seed

Ivar verify_key: [`VerifyKey`] The verify (i.e. public) key that corresponds with this signing key.

classmethod **`generate`**`()`

Generates a random `SingingKey` object.

Return type `SigningKey`

`sign``(`*message*, *encoder=<class 'nacl.encoding.RawEncoder'>*`)`

Sign a message using this key.

Parameters

- **message** – [`bytes`] The data to be signed.
- **encoder** – A class that is used to encode the signed message.

Return type `SignedMessage`

class `nacl.signing.`**`VerifyKey`**`(`*key*, *encoder=<class 'nacl.encoding.RawEncoder'>*`)`

The public key counterpart to an Ed25519 SigningKey for producing digital signatures.

Parameters

- **key** – [`bytes`] Serialized Ed25519 public key
- **encoder** – A class that is able to decode the *key*

`verify``(`*smessage*, *signature=None*, *encoder=<class 'nacl.encoding.RawEncoder'>*`)`

Verifies the signature of a signed message, returning the message if it has not been tampered with else raising `BadSignatureError`.

Parameters

- **smessage** – [`bytes`] Either the original messaged or a signature and message concated together.
- **signature** – [`bytes`] If an unsigned message is given for smessage then the detached signature must be provded.
- **encoder** – A class that is able to decode the secret message and signature.

Return type `bytes`

class `nacl.signing.`**`SignedMessage`**

A bytes subclass that holds a messaged that has been signed by a `SigningKey`.

`message`

The message contained within the `SignedMessage`.

signature
The signature contained within the `SignedMessage`.

class `nacl.signing.`**`BadSignatureError`**
Raised when the signature was forged or otherwise corrupt.

3.3 Ed25519

Ed25519 is a public-key signature system with several attractive features:

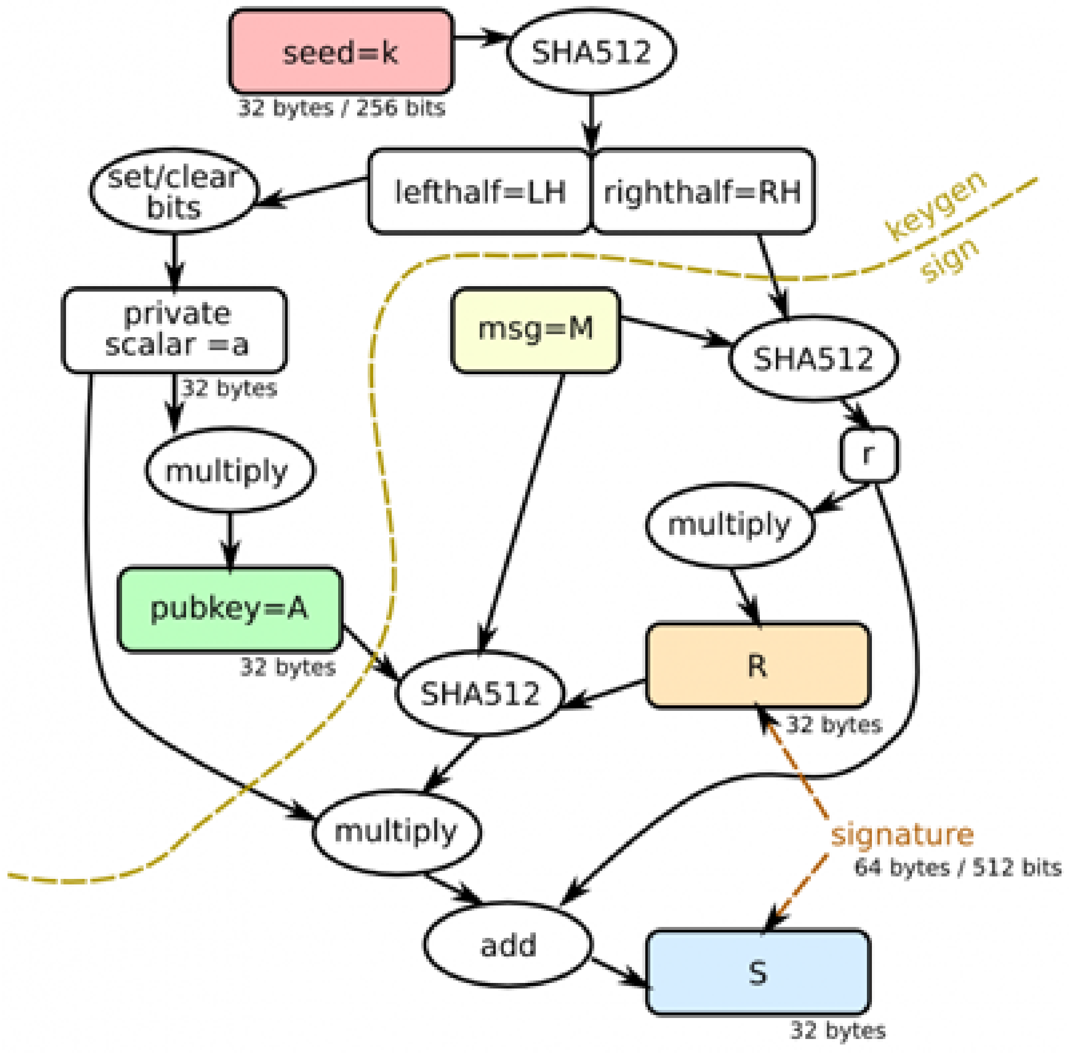
- **Fast single-signature verification:** Ed25519 takes only 273364 cycles to verify a signature on Intel's widely deployed Nehalem/Westmere lines of CPUs. (This performance measurement is for short messages; for very long messages, verification time is dominated by hashing time.) Nehalem and Westmere include all Core i7, i5, and i3 CPUs released between 2008 and 2010, and most Xeon CPUs released in the same period.
- **Even faster batch verification:** Ed25519 performs a batch of 64 separate signature verifications (verifying 64 signatures of 64 messages under 64 public keys) in only 8.55 million cycles, i.e., under 134000 cycles per signature. Ed25519 fits easily into L1 cache, so contention between cores is negligible: a quad-core 2.4GHz Westmere verifies 71000 signatures per second, while keeping the maximum verification latency below 4 milliseconds.
- **Very fast signing:** Ed25519 takes only 87548 cycles to sign a message. A quad-core 2.4GHz Westmere signs 109000 messages per second.
- **Fast key generation:** Key generation is almost as fast as signing. There is a slight penalty for key generation to obtain a secure random number from the operating system; /dev/urandom under Linux costs about 6000 cycles.
- **High security level:** This system has a 2^128 security target; breaking it has similar difficulty to breaking NIST P-256, RSA with ~3000-bit keys, strong 128-bit block ciphers, etc. The best attacks known actually cost more than 2^140 bit operations on average, and degrade quadratically in success probability as the number of bit operations drops.
- **Collision resilience:** Hash-function collisions do not break this system. This adds a layer of defense against the possibility of weakness in the selected hash function.
- **No secret array indices:** Ed25519 never reads or writes data from secret addresses in RAM; the pattern of addresses is completely predictable. Ed25519 is therefore immune to cache-timing attacks, hyperthreading attacks, and other side-channel attacks that rely on leakage of addresses through the CPU cache.
- **No secret branch conditions:** Ed25519 never performs conditional branches based on secret data; the pattern of jumps is completely predictable. Ed25519 is therefore immune to side-channel attacks that rely on leakage of information through the branch-prediction unit.
- **Small signatures:** Ed25519 signatures are only 512-bits (64 bytes), one of the smallest signature sizes available.
- **Small keys:** Ed25519 keys are only 256-bits (32 bytes), making them small enough to easily copy and paste. Ed25519 also allows the public key to be derived from the private key, meaning that it doesn't need to be included in a serialized private key in cases you want both.
- **Deterministic:** Unlike (EC)DSA, Ed25519 does not rely on an entropy source when signing messages (which has lead to catastrophic private key compromises), but instead computes signature nonces from a combination of a hash of the signing key's "seed" and the message to be signed. This avoids using an entropy source for nonces, which can be a potential attack vector if the entropy source is not generating good random numbers. Even a single reused nonce can lead to a complete disclosure of the private key in these schemes, which Ed25519 avoids entirely by being deterministic instead of tied to an entropy source.

The numbers 87548 and 273364 shown above are official *eBATS <http://bench.cr.yp.to/>* reports for a Westmere CPU (Intel Xeon E5620, hydra2).

Ed25519 signatures are elliptic-curve signatures, carefully engineered at several levels of design and implementation to achieve very high speeds without compromising security.

3.3.1 Algorithm

- **Public Keys:** Curve25519 high-speed elliptic curve cryptography
- **Signatures:** Ed25519 digital signature system



k Ed25519 private key (passed into `SigningKey`)

A Ed25519 public key derived from k

M message to be signed

R a deterministic nonce value calculated from a combination of private key data RH and the message M

S Ed25519 signature

CHAPTER

FOUR

SUPPORT FEATURES

4.1 Encoders

PyNaCl supports a simple method of encoding and decoding messages in different formats. Encoders are simple classes with staticmethods that encode/decode and are typically passed as a keyword argument *encoder* to various methods.

For example you can generate a signing key and encode it in hex with:

```
hex_key = nacl.signing.SigningKey.generate().encode(encoder=nacl.encoding.HexEncoder)
```

Then you can later decode it from hex:

```
signing_key = nacl.signing.SigningKey(hex_key, encoder=nacl.encoding.HexEncoder)
```

4.1.1 Built in Encoders

class `nacl.encoding.`**`RawEncoder`**

class `nacl.encoding.`**`HexEncoder`**

class `nacl.encoding.`**`Base16Encoder`**

class `nacl.encoding.`**`Base32Encoder`**

class `nacl.encoding.`**`Base64Encoder`**

4.1.2 Defining your own Encoder

Defining your own encoder is easy. Each encoder is simply a class with 2 static methods. For example here is the hex encoder:

```
import binascii

class HexEncoder(object):

    @staticmethod
    def encode(data):
        return binascii.hexlify(data)

    @staticmethod
    def decode(data):
        return binascii.unhexlify(data)
```

CHAPTER

FIVE

API DOCUMENTATION

5.1 nacl.hash

CHAPTER

SIX

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX