PyNaCl

Release 1.2.0.dev1

Sep 06, 2017

Contents

CHAPTER 1

Contents

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.

Examples

nacl.public.Box

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 to decrypt the given ciphertexts. The same shared key will be 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) will equal that from (pkbob, skalice).

This is how the system works:

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

# Generate Bob's private key, which must be kept secret
skbob = PrivateKey.generate()

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

# Alice does the same and then Alice and Bob exchange public keys
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 it
#   as just a binary blob of data.
message = b"Kill all humans"
```

PyNaCl can automatically generate a random nonce for us, making the encryption very simple:

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

However, if we need to use an explicit nonce, it can be passed along with the message:

```
# 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 nonces are just sequences of 24 random bytes.
nonce = nacl.utils.random(Box.NONCE_SIZE)

encrypted = bob_box.encrypt(message, nonce)
```

Finally, the message is decrypted (regardless of how the nonce was generated):

```
# 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)
```

nacl.public.SealedBox

The *SealedBox* class encrypts messages addressed to a specified key-pair by using ephemeral sender's keypairs, which will be discarded just after encrypting a single plaintext message.

This kind of construction allows sending messages, which only the recipient can decrypt without providing any kind of cryptographic proof of sender's authorship.

Warning: By design, the recipient will have no means to trace the ciphertext to a known author, since the sending keypair itself is not bound to any sender's identity, and the sender herself will not be able to decrypt the ciphertext she just created, since the private part of the key cannot be recovered after use.

This is how the system works:

```
import nacl.utils
from nacl.public import PrivateKey, SealedBox

# Generate Bob's private key, as we've done in the Box example
skbob = PrivateKey.generate()
pkbob = skbob.public_key

# Alice wishes to send a encrypted message to Bob,
# but prefers the message to be untraceable
sealed_box = SealedBox(pkbob)

# This is Alice's message
message = b"Kill all kittens"

# Encrypt the message, it will carry the ephemeral key public part
# to let Bob decrypt it
encrypted = sealed_box.encrypt(message)
```

Now, Bob wants to read the secret message he just received; therefore he must create a SealedBox using his own private key:

```
unseal_box = SealedBox(skbob)
# decrypt the received message
plaintext = unseal_box.decrypt(encrypted)
```

Reference

class nacl.public.**PublicKey**(*public_key*, *encoder*)

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

Parameters

- **public_key** (*bytes*) – Encoded Curve25519 public key.
- **encoder** – A class that is able to decode the public_key.

class nacl.public.**PrivateKey**(*private_key*, *encoder*)

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** (*bytes*) – The private key used to decrypt messages.
- **encoder** – A class that is able to decode the private_key.

public_key
An instance of *PublicKey* that corresponds with the private key.

classmethod generate()
Generates a random *PrivateKey* object

Returns An instance of *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** – An instance of *PrivateKey* used to encrypt and decrypt messages
- **public_key** – An instance of *PublicKey* used to encrypt and decrypt messages

classmethod decode(*encoded*, *encoder*)
Decodes a serialized *Box*.

Returns An instance of *Box*.

encrypt(*plaintext*, *nonce*, *encoder*)
Encrypts the plaintext message using the given *nonce* (or generates one randomly if omitted) 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** – A class that is able to decode the ciphertext.

Returns An instance of *EncryptedMessage*.

decrypt(*ciphertext*, *nonce*, *encoder*)
Decrypts the ciphertext using the *nonce* (explicitly, when passed as a parameter or implicitly, when omitted, as part of the ciphertext) and returns the plaintext message.

Parameters

- **ciphertext** (*bytes*) – The encrypted message to decrypt.
- **nonce** (*bytes*) – The nonce to use in the decryption.
- **encoder** – A class that is able to decode the plaintext.

Return bytes The decrypted plaintext.

shared_key()
Returns the Curve25519 shared secret, that can then be used as a key in other symmetric ciphers.

> **Warning:** It is **VITALLY** important that you use a nonce with your symmetric cipher. If you fail to do this, you compromise the privacy of the messages encrypted. Ensure that the key length of your cipher is 32 bytes.

Return bytes The shared secret.

class nacl.public.**SealedBox**(*receiver_key*)

The SealedBox class can box and unbox messages sent to a receiver key using an ephemeral sending keypair.

encrypt(*plaintext*, *encoder*)

Encrypt the message using a Box constructed from an ephemeral key-pair and the receiver key.

The public part of the ephemeral key-pair will be enclosed in the returned ciphertext.

The private part of the ephemeral key-pair will be scrubbed before returning the ciphertext, therefore, the sender will not be able to decrypt the message.

Parameters

- **plaintext** (*bytes*) – The plaintext message to encrypt.
- **encoder** – A class that is able to decode the ciphertext.

Return bytes The public part of the ephemeral keypair, followed by the encrypted ciphertext

decrypt(*ciphertext*, *encoder*)

Decrypt the message using a Box constructed from the receiver key and the ephemeral key enclosed in the ciphertext.

Parameters

- **ciphertext** (*bytes*) – The ciphertext message to decrypt.
- **encoder** – A class that is able to decode the ciphertext.

Return bytes The decrypted message

Algorithm

- **Public Keys:** Curve25519 high-speed elliptic curve cryptography

Secret Key Encryption

Secret key encryption (also called symmetric 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.

Example

```
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 it as just a binary blob of data.
message = b"The president will be exiting through the lower levels"
```

PyNaCl can automatically generate a random nonce for us, making the encryption very simple:

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

However, if we need to use an explicit nonce, it can be passed along with the message:

```
# 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 nonces are just sequences of 24 random bytes.
nonce = nacl.utils.random(nacl.secret.SecretBox.NONCE_SIZE)

encrypted = box.encrypt(message, nonce)
```

Finally, the message is decrypted (regardless of how the nonce was generated):

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

Requirements

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.

Nonce

The 24-byte nonce (Number used once) given to *encrypt()* and *decrypt()* must **NEVER** be reused for a particular key. Reusing a nonce may give an attacker enough information to decrypt or forge other 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 or unpredictable, nor does the method of generating them need to be secret. A nonce could simply be a counter incremented with each message encrypted, which can be useful in connection-oriented protocols to reject duplicate messages ("replay attacks"). A bidirectional connection could use the same key for both directions, as long as their nonces never overlap (e.g. one direction always sets the high bit to "1", the other always sets it to "0").

If you use a counter-based nonce along with a key that is persisted from one session to another (e.g. saved to disk), you must store the counter along with the key, to avoid accidental nonce reuse on the next session. For this reason, many protocols derive a new key for each session, reset the counter to zero with each new key, and never store the derived key or the counter.

You can safely generate random nonces by calling *random()* with `SecretBox.NONCE_SIZE`.

Reference

class `nacl.secret.`**`SecretBox`**(*key*, *encoder*)

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`** (*bytes*) – The secret key used to encrypt and decrypt messages.
- **`encoder`** – A class that is able to decode the `key`.

`encrypt`(*plaintext*, *nonce*, *encoder*)

Encrypts the plaintext message using the given *nonce* (or generates one randomly if omitted) 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`** – A class that is able to decode the ciphertext.

Returns An instance of *EncryptedMessage*.

`decrypt`(*ciphertext*, *nonce*, *encoder*)

Decrypts the ciphertext using the *nonce* (explicitly, when passed as a parameter or implicitly, when omitted, as part of the ciphertext) and returns the plaintext message.

Parameters

- **`ciphertext`** (*bytes*) – The encrypted message to decrypt.
- **`nonce`** (*bytes*) – The nonce to use in the decryption.
- **`encoder`** – A class that is able to decode the plaintext.

Return bytes The decrypted plaintext.

Algorithm details

Encryption Salsa20 stream cipher

Authentication Poly1305 MAC

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.

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(b"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.exceptions.BadSignatureError if the signature check fails
verify_key.verify(signed)
```

Reference

class `nacl.signing.`**`SigningKey`**(*seed*, *encoder*)

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 its 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`.

verify_key
An instance of *VerifyKey* (i.e. public key) that corresponds with the signing key.

classmethod generate()
Generates a random *SigningKey* object

Returns An instance of *SigningKey*.

sign(*message*, *encoder*)
Sign a message using this key.

Parameters

- **message** (*bytes*) – The data to be signed.
- **encoder** – A class that is able to decode the signed message.

Returns An instance of *SignedMessage*.

class nacl.signing.**VerifyKey**(*key*, *encoder*)
The public key counterpart to an Ed25519 *SigningKey* for producing digital signatures.

Parameters

- **key** (*bytes*) – A serialized Ed25519 public key.
- **encoder** – A class that is able to decode the `key`.

verify(*smessage*, *signature*, *encoder*)
Verifies the signature of a signed message.

Parameters

- **smessage** (*bytes*) – The signed message to verify. This is either the original message or the concated signature and message.
- **signature** (*bytes*) – The signature of the message to verify against. If the value of `smessage` is the concated signature and message, this parameter can be `None`.
- **encoder** – A class that is able to decode the secret message and signature.

Return bytes The message if successfully verified.

Raises **nacl.exceptions.BadSignatureError** – This is raised if the signature is invalid.

class nacl.signing.**SignedMessage**
A bytes subclass that holds a messaged that has been signed by a *SigningKey*.

signature
The signature contained within the *SignedMessage*.

message
The message contained within the *SignedMessage*.

Ed25519

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

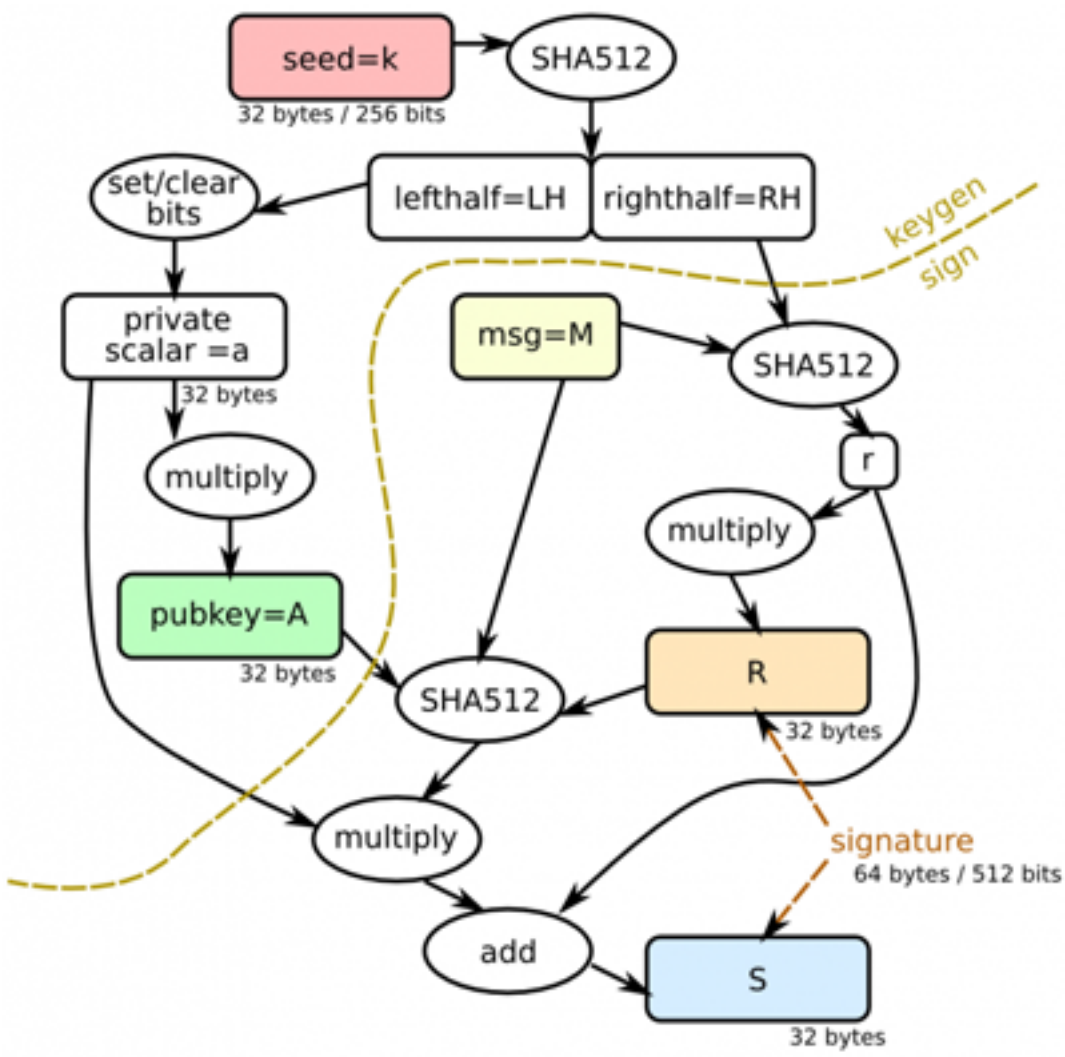
- **Fast single-signature verification:** Ed25519 takes only 273364 cycles to verify a signature on Intel's widely deployed Nehalem/Westmere lines of CPUs. (This performance measurement is for short messages; for very long messages, verification time is dominated by hashing time.) Nehalem and Westmere 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 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.

Algorithm

- **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

Hashing

Cryptographic secure hash functions are irreversible transforms of input data to a fixed length *digest*.

The standard properties of a cryptographic hash make these functions useful both for standalone usage as data integrity checkers, as well as `black-box` building blocks of other kind of algorithms and data structures.

All of the hash functions exposed in `nacl.hash` can be used as data integrity checkers.

Integrity check examples

Message's creator perspective (*sha256()*, *sha512()*, *blake2b()*)

```
import nacl.encoding
import nacl.hash

HASHER = nacl.hash.sha256
# could be nacl.hash.sha512 or nacl.hash.blake2b instead

# define a 1024 bytes log message
msg = 16*b'256 BytesMessage'
digest = nacl.hash.HASHER(msg, encoder=nacl.encoding.HexEncoder)

# now send msg and digest to the user
print(nacl.encoding.HexEncoder.encode(msg))
print(digest)
```

Message's user perspective (*sha256()*, *sha512()*, *blake2b()*)

```
from nacl.bindings.utils import sodium_memcmp
import nacl.encoding
import nacl.hash

HASHER = nacl.hash.sha256
# could be nacl.hash.sha512 or nacl.hash.blake2b instead

# we received a 1024 bytes long message and it hex encoded digest
received_msg = nacl.encoding.HexEncoder.decode(
b'32353620427974657340d65737361676532353620427974657340d657373616765'
b'32353620427974657340d65737361676532353620427974657340d657373616765'
b'32353620427974657340d65737361676532353620427974657340d657373616765'
b'32353620427974657340d65737361676532353620427974657340d657373616765'
b'32353620427974657340d65737361676532353620427974657340d657373616765'
b'32353620427974657340d65737361676532353620427974657340d657373616765'
b'32353620427974657340d65737361676532353620427974657340d657373616765'
b'32353620427974657340d65737361676532353620427974657340d657373616765'
)

dgst = b'12b413c70c148d79bb57a1542156c5f35e24ad77c38e8c0e776d055e827cdd45'

shortened = received_msg[:-1]
modified = b'modified' + received_msg[:-8]

orig_dgs = HASHER(received_msg, encoder=nacl.encoding.HexEncoder)
shrt_dgs = HASHER(shortened, encoder=nacl.encoding.HexEncoder)
mdfd_dgs = HASHER(modified, encoder=nacl.encoding.HexEncoder)

def eq_chk(dgs0, dgs1):
    if sodium_memcmp(dgs0, dgs1):
        return 'equals'
    return 'is different from'

MSG = 'Digest of {0} message {1} original digest'

for chk in (('original', orig_dgs),
            ('truncated', shrt_dgs),
            ('modified', mdfd_dgs)):
```

```
    print(MSG.format(chk[0], eq_chk(dgst, chk[1])))
```

Additional hashing usages for `blake2b`

As already hinted above, traditional cryptographic hash functions can be used as building blocks for other uses, typically combining a secret-key with the message via some construct like the `HMAC` one.

The *`blake2b`* hash function can be used directly both for message authentication and key derivation, replacing the `HMAC` construct and the `HKDF` one by setting the additional parameters `key`, `salt` and `person`.

Please note that **key stretching procedures** like `HKDF` or the one outlined in *Key derivation* are **not** suited to derive a *cryptographically-strong* key from a *low-entropy input* like a plain-text password or to compute a strong *long-term stored* hash used as password verifier. See the *Password hashing* section for some more informations and usage examples of the password hashing constructs provided in `pwhash`.

Message authentication

To authenticate a message, using a secret key, the blake2b function must be called as in the following example.

Message authentication example

```
import nacl.encoding
from nacl.hash import blake2b

msg = 16*b'256 BytesMessage'
msg2 = 16*b'256 bytesMessage'

auth_key = nacl.utils.random(size=64)
# the simplest way to get a cryptographic quality auth_key
# is to generate it with a cryptographic quality
# random number generator

auth1_key = nacl.utils.random(size=64)
# generate a different key, just to show the mac is changed
# both with changing messages and with changing keys

mac0 = blake2b(msg, key=auth_key, encoder=nacl.encoding.HexEncoder)
mac1 = blake2b(msg, key=auth1_key, encoder=nacl.encoding.HexEncoder)
mac2 = blake2b(msg2, key=auth_key, encoder=nacl.encoding.HexEncoder)

for i, mac in enumerate((mac0, mac1, mac2)):
    print('Mac{0} is: {1}.'.format(i, mac))
```

Key derivation

The blake2b algorithm can replace a key derivation function by following the lines of:

Key derivation example

```
import nacl.encoding
import nacl.utils
from nacl.hash import blake2b

master_key = nacl.utils.random(64)

derivation_salt = nacl.utils.random(16)

personalization = b'<DK usage>'

derived = blake2b(b'', key=master_key, salt=derivation_salt,
                  personal=personalization,
                  encoding=nacl.encoding.RawEncoder)
```

By repeating the key derivation procedure before encrypting our messages, and sending the derivation_salt along with the encrypted message, we can expect to never reuse a key, drastically reducing the risks which ensue from such a reuse.

Password hashing

Password hashing and password based key derivation mechanisms in actual use are all based on the idea of iterating a hash function many times on a combination of the password and a random `salt`, which is stored along with the hash, and allows verifying a proposed password while avoiding clear-text storage.

The latest developments in password hashing have been *memory-hard* and *tunable* mechanisms, pioneered by the `scrypt` mechanism *[SD2012]*, and followed-on by the schemes submitted to the **Password Hashing Competition** *[PHC]*.

The `nacl.pwhash` module exposes both one of the **PHC** recommended `argon2` mechanisms and the `scrypt` one.

While some sources suggest to give preference to data dependent password hashing mechanisms, the only mechanism of the `argon2` family being implemented in libsodium as of version `1.0.12` is the data-independent `argon2i` one.

If you think in your use-case the risk of potential timing-attacks stemming from data-dependency is less than the potential time/memory trade-offs coming out of data-independency, you should consider staying with `scrypt` password hashing instead of jumping to `argon2i`

Password storage and verification

Both `argon2i_str()` and `scryptsalsa208sha256_str()` internally generate a random salt, and return a hash encoded in ascii modular crypt format, which can be stored in a shadow-like file:

```
>>> import nacl.pwhash
>>> password = b'my password'
>>> for i in range(4):
...     print(nacl.pwhash.argon2i_str(password))
...
b'$argon2i$v=19$m=32768,t=4,p=1$...'
b'$argon2i$v=19$m=32768,t=4,p=1$...'
b'$argon2i$v=19$m=32768,t=4,p=1$...'
b'$argon2i$v=19$m=32768,t=4,p=1$...'
>>> for i in range(4):
```

```
...     print(nacl.pwhash.scryptsalsa208sha256_str(password))
...
b'$7$C6..../....p9h...'
b'$7$C6..../....pVs...'
b'$7$C6..../....qW2...'
b'$7$C6..../....bxH...'
```

To verify a user-proposed password, the `verify_argon2i()` and `verify_scryptsalsa208sha256()` function extract the used salt, memory and operation count parameters from the modular format string and check the compliance of the proposed password with the stored hash:

```
>>> import nacl.pwhash
>>> hashed = (b'$7$C6..../....qv5tF9KG2WbuMeUOa0TCoqwLHQ8s0TjQdSagne'
...           b'9NvU0$3d218uChMvdvN6EwSvKHMASkZIG51XPIsZQDcktKyN7'
...           )
>>> correct = b'my password'
>>> wrong = b'My password'
>>> # while the result will be True on password match,
... # on mismatch an exception will be raised
... res = nacl.pwhash.verify_scryptsalsa208sha256(hashed, correct)
>>> print(res)
True
>>>
>>> res2 = nacl.pwhash.verify_scryptsalsa208sha256(hashed, wrong)
Traceback (most recent call last):
    ...
nacl.exceptions.InvalidkeyError: Wrong password
>>>
```

Future-proofing your password verification routine

A very nice aspect of modular crypt format is the presence of the `$identifier$` prefix, which can be used to dispatch the correct hash verifier.

This could help continued support for an hash format, even after choosing a different one as a default storage format for new passwords.

To prepare yourself in exploiting this feature, you should simply remember to test the serialized password hash identifier before verifying a proposed password:

```
from nacl.exceptions import ValueError
from nacl.pwhash import (verify_argon2i, verify_scryptsalsa208sha256)
def check_password(serialized, proposed):
    if serialized.startswith(b'$argon2i$'):
        res = verify_argon2i(serialized, proposed)
    elif serialized.startswith(b'$7$'):
        res = verify_scryptsalsa208sha256(serialized, proposed)
    else:
        raise ValueError('Unknown serialization format')
    return res
```

By doing so, if a future version of PyNaCl would get released with support for a new password hash mechanism, you'll be able to just insert another `elif serialized.startswith(b'$newhash$):` clause to your password checker to begin supporting `newhash` without harming the previously hashed passwords.

Key derivation

Alice needs to send a secret message to Bob, using a shared password to protect the content. She generates a random salt, combines it with the password using either `kdf_argon2i()` or `kdf_scryptsalsa208sha256()` and sends the message along with the salt and key derivation parameters.

```
from nacl import pwhash, secret, utils


password = b'password shared between Alice and Bob'
message = b"This is a message for Bob's eyes only"

kdf = pwhash.kdf_argon2i
salt = utils.random(pwhash.ARGON2I_SALTBYTES)
ops = pwhash.ARGON2I_OPSLIMIT_SENSITIVE
mem = pwhash.ARGON2I_MEMLIMIT_SENSITIVE

# or, if there is a need to use scrypt:
# kdf = pwhash.kdf_scryptsalsa208sha256
# salt = utils.random(pwhash.SCRYPT_SALTBYTES)
# ops = pwhash.SCRYPT_OPSLIMIT_SENSITIVE
# mem = pwhash.SCRYPT_MEMLIMIT_SENSITIVE

Alices_key = kdf(secret.SecretBox.KEY_SIZE, password, salt,
                 opslimit=ops, memlimit=mem)
Alices_box = secret.SecretBox(Alices_key)
nonce = utils.random(secret.SecretBox.NONCE_SIZE)

encrypted = Alices_box.encrypt(message, nonce)

# now Alice must send to Bob both the encrypted message
# and the KDF parameters: salt, opslimit and memlimit;
# using the same kdf mechanism, parameters **and password**
# Bob is able to derive the correct key to decrypt the message


Bobs_key = kdf(secret.SecretBox.KEY_SIZE, password, salt,
               opslimit=ops, memlimit=mem)
Bobs_box = secret.SecretBox(Bobs_key)
received = Bobs_box.decrypt(encrypted)
print(received)
```

if Eve manages to get the encrypted message, and tries to decrypt it with a incorrect password, even if she does know all of the key derivation parameters, she would derive a different key. Therefore the decryption would fail and an exception would be raised:

```
>>> from nacl import pwhash, secret, utils
>>>
>>> ops = pwhash.ARGON2I_OPSLIMIT_SENSITIVE
>>> mem = pwhash.ARGON2I_MEMLIMIT_SENSITIVE
>>>
>>> salt = utils.random(pwhash.ARGON2I_SALTBYTES)
>>>
>>> guessed_pw = b'I think Alice shared this password with Bob'
>>>
>>> Eves_key = pwhash.kdf_argon2i(secret.SecretBox.KEY_SIZE,
...                               guessed_pw, salt,
...                               opslimit=ops, memlimit=mem)
```

```
>>> Eves_box = secret.SecretBox(Eves_key)
>>> intercepted = Eves_box.decrypt(encrypted)
Traceback (most recent call last):
    ...
nacl.exceptions.CryptoError: Decryption failed. Ciphertext failed ...
```

Contrary to the hashed password storage case where a serialization format is well-defined, in the raw key derivation case the library user must take care to store (and retrieve) both a reference to the kdf used to derive the secret key and all the derivation parameters. These parameters are needed to later generate the same secret key from the password.

Module level constants for operation and memory cost tweaking

To help in selecting the correct values for the tweaking parameters for both the **scrypt** and the **argon2i** constructions, the `nacl.pwhash` provides suggested values for the *opslimit* and *memlimit* parameters, which are belived to be valid as of CPU/ASIC speeds current in year 2017.

The provided constants are:

for scrypt:

- `nacl.pwhash.SCRYPT_OPSLIMIT_INTERACTIVE`
- `nacl.pwhash.SCRYPT_MEMLIMIT_INTERACTIVE`
- `nacl.pwhash.SCRYPT_OPSLIMIT_SENSITIVE`
- `nacl.pwhash.SCRYPT_MEMLIMIT_SENSITIVE`

for argon2:

- `nacl.pwhash.ARGON2I_OPSLIMIT_INTERACTIVE`
- `nacl.pwhash.ARGON2I_MEMLIMIT_INTERACTIVE`
- `nacl.pwhash.ARGON2I_OPSLIMIT_MODERATE`
- `nacl.pwhash.ARGON2I_MEMLIMIT_MODERATE`
- `nacl.pwhash.ARGON2I_OPSLIMIT_SENSITIVE`
- `nacl.pwhash.ARGON2I_MEMLIMIT_SENSITIVE`

In general, the _INTERACTIVE values are recommended in the case of hashes stored for interactive password checking, and lead to a sub-second password verification time, with a memory consumption in the tens of megabytes range, while the _SENSITIVE values are meant to store hashes for password protecting sensitive data, and lead to hashing times exceeding one second, with memory consumption in the hundred of megabytes range. The _MODERATE values, suggested for argon2i are meant to run the construct at a runtime and memory cost intermediate between _INTERACTIVE and _SENSITIVE.

CHAPTER 2

Support Features

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)
```

Built in Encoders

class nacl.encoding.**RawEncoder**

class nacl.encoding.**HexEncoder**

class nacl.encoding.**Base16Encoder**

class nacl.encoding.**Base32Encoder**

class nacl.encoding.**Base64Encoder**

class nacl.encoding.**URLSafeBase64Encoder**

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)
```

Exceptions

All of the exceptions raised from PyNaCl-exposed methods/functions are subclasses of `nacl.exceptions.CryptoError`. This means downstream users can just wrap cryptographic operations inside a

```
try:
    # cryptographic operations
except nacl.exceptions.CryptoError:
    # cleanup after any kind of exception
    # raised from cryptographic-related operations
```

These are the exceptions implemented in `nacl.exceptions`:

PyNaCl specific exceptions

class `CryptoError`
Base exception for all nacl related errors

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

class `InvalidkeyError`
Raised on password/key verification mismatch

PyNaCl exceptions mixing-in standard library ones

Both for clarity and for compatibility with previous releases of the PyNaCl, the following exceptions mix-in the same-named standard library exception to `CryptoError`.

class `RuntimeError`
is a subclass of both CryptoError and standard library's RuntimeError, raised for internal library errors

class `AssertionError`
is a subclass of both CryptoError and standard library's AssertionError, raised by default from *`ensure()`* when the checked condition is *False*

class `TypeError`
is a subclass of both CryptoError and standard library's TypeError

class `ValueError`
is a subclass of both CryptoError and standard library's ValueError

Utilities

class nacl.utils.**EncryptedMessage**
A bytes subclass that holds a message that has been encrypted by a *SecretBox* or *Box*. The full content of the bytes object is the combined nonce and ciphertext.

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

ciphertext
The ciphertext contained within the *EncryptedMessage*.

nacl.utils.**random**(*size=32*)
Returns a random bytestring with the given size.

Parameters **size** (*bytes*) – The size of the random bytestring.

Return bytes The random bytestring.

nacl.utils.**ensure**(*cond*, **args*, *raising=nacl.exceptions.AssertionError*)
Returns if a condition is true, otherwise raise a caller-configurable Exception

Parameters

- **cond** (*bool*) – the condition to be checked
- **args** (*sequence*) – the arguments to be passed to the exception's constructor
- **raising** (*exception*) – the exception to be raised if *cond* is *False*

nacl.hash

nacl.hash.**sha256**(*message*, *encoder*)
Hashes message with SHA256.

Parameters

- **message** (*bytes*) – The message to hash.
- **encoder** – A class that is able to encode the hashed message.

Return bytes The hashed message.

nacl.hash.**sha512**(*message*, *encoder*)
Hashes message with SHA512.

Parameters

- **message** (*bytes*) – The message to hash.
- **encoder** – A class that is able to encode the hashed message.

Return bytes The hashed message.

nacl.hash.**blake2b**(*data*, *digest_size=BLAKE2B_BYTES*, *key=b''*, *salt=b''*, *person=b''*, *encoder=nacl.encoding.HexEncoder*)
One-shot blake2b digest

Parameters

- **data** (*bytes*) – the digest input byte sequence

- **digest_size** (*int*) – the requested digest size; must be at most BLAKE2B_BYTES_MAX; the default digest size is BLAKE2B_BYTES
- **key** (*bytes*) – the key to be set for keyed MAC/PRF usage; if set, the key must be at most BLAKE2B_KEYBYTES_MAX long
- **salt** (*bytes*) – an initialization salt at most BLAKE2B_SALTBYTES long; it will be zero-padded if needed
- **person** (*bytes*) – a personalization string at most BLAKE2B_PERSONALBYTES long; it will be zero-padded if needed
- **encoder** (*class*) – the encoder to use on returned digest

Returns encoded bytes data

Return type the return type of the choosen encoder

nacl.hash.**siphash24**(*message*, *key=b''*, *encoder=nacl.encoding.HexEncoder*)

Computes a keyed MAC of message using siphash-2-4

Parameters

- **message** (*bytes*) – The message to hash.
- **key** (bytes(SIPHASH_KEYBYTES)) – the message authentication key to be used It must be a SIPHASH_KEYBYTES long bytes sequence
- **encoder** – A class that is able to encode the hashed message.

Returns The hashed message.

Return type bytes(SIPHASH_BYTES) long bytes sequence

nacl.hash.**siphashx24**(*message*, *key=b''*, *encoder=nacl.encoding.HexEncoder*)

New in version 1.2.

Computes a keyed MAC of message using the extended output length variant of siphash-2-4

Parameters

- **message** (*bytes*) – The message to hash.
- **key** (bytes(SIPHASHX_KEYBYTES)) – the message authentication key to be used It must be a SIPHASHX_KEYBYTES long bytes sequence
- **encoder** – A class that is able to encode the hashed message.

Returns The hashed message.

Return type bytes(SIPHASHX_BYTES) long bytes sequence

nacl.hashlib

The nacl.hashlib module exposes directly usable implementations of raw constructs which libsodium exposes with simplified APIs, like the ones in nacl.hash and in nacl.pwhash.

The *blake2b* and *scrypt()* implementations are as API compatible as possible with the corresponding ones added to cpython standard library's hashlib module in cpython's version 3.6.

class nacl.hashlib.**blake2b**(*data=b''*, *digest_size=BYTES*, *key=b''*, *salt=b''*, *person=b''*)

Returns an hash object which exposes an API mostly compatible to python3.6's hashlib.blake2b (the only difference being missing support for tree hashing parameters in the contructor)

The methods `update()`, `copy()`, `digest()` and `hexdigest()` have the same semantics as described in hashlib documentation.

Each instance exposes the `digest_size`, `block_size` `name` properties as required by hashlib API.

MAX_DIGEST_SIZE
the maximum allowed value of the requested digest_size

MAX_KEY_SIZE
the maximum allowed size of the password parameter

PERSON_SIZE
the maximimum size of the personalization

SALT_SIZE
the maximimum size of the salt

`nacl.hashlib.`**`scrypt`**(*password*, *salt=''*, *n=2**20*, *r=8*, *p=1*, *maxmem=2**25*, *dklen=64*)
Derive a raw cryptographic key using the scrypt KDF.

Parameters

- **password** (*bytes*) – the input password
- **salt** (*bytes*) – a crypographically-strong random salt
- **n** (*int*) – CPU/Memory cost factor
- **r** (*int*) – block size multiplier: the used block size will be 128 * r
- **p** (*int*) – requested parallelism: the number of indipendently running scrypt constructs which will contribute to the final key generation
- **maxmem** (*int*) – maximum memory the whole scrypt construct will be entitled to use
- **dklen** (*int*) – length of the derived key

Returns a buffer dklen bytes long containing the derived key

Implements the same signature as the `hashlib.scrypt` implemented in cpython version 3.6

The recommended values for n, r, p in 2012 were n = 2**14, r = 8, p = 1; as of 2016, libsodium suggests using n = 2**14, r = 8, p = 1 in a "interactive" setting and n = 2**20, r = 8, p = 1 in a "sensitive" setting.

The total memory usage will respectively be a little greater than 16MB in the "interactive" setting, and a little greater than 1GB in the "sensitive" setting.

Installation

Binary wheel install

PyNaCl ships as a binary wheel on OS X, Windows and Linux `manylinux1`[1] , so all dependencies are included. Make sure you have an up-to-date pip and run:

```
$ pip install pynacl
```

[1] manylinux1 wheels are built on a baseline linux environment based on Centos 5.11 and should work on most x86 and x86_64 glibc based linux environments.

Linux source build

PyNaCl relies on libsodium, a portable C library. A copy is bundled with PyNaCl so to install you can run:

```
$ pip install pynacl
```

If you'd prefer to use the version of `libsodium` provided by your distribution, you can disable the bundled copy during install by running:

```
$ SODIUM_INSTALL=system pip install pynacl
```

> **Warning:** Usage of the legacy `easy_install` command provided by setuptools is generally discouraged, and is completely unsupported in PyNaCl's case.

Doing A Release

To run a PyNaCl release follow these steps:

- Update the version number in `src/nacl/__init__.py`.
- Update `README.rst` changelog section with the date of the release.
- Send a pull request with these items and wait for it to be merged.
- Run `invoke release {version}`

Once the release script completes you can verify that the sdist and wheels are present on PyPI and then send a new PR to bump the version to the next major version (e.g. `1.2.0.dev1`).

Changelog

1.2.0 (UNRELEASED)

- Update `libsodium` to 1.0.12.
- Infrastructure: add jenkins support for automatic build of `manylinux1` binary wheels
- Added support for `SealedBox` construction.
- Added support for `argon2i` password hashing construct
- Added support for 128 bit `siphashx24` variant of `siphash24`.
- Added support for `from_seed` APIs for X25519 keypair generation.

1.1.2 - 2017-03-31

- reorder link time library search path when using bundled libsodium

1.1.1 - 2017-03-15

- Fixed a circular import bug in `nacl.utils`.

1.1.0 - 2017-03-14

- Dropped support for Python 2.6.
- Added `shared_key()` method on `Box`.
- You can now pass `None` to `nonce` when encrypting with `Box` or `SecretBox` and it will automatically generate a random nonce.
- Added support for `siphash24`.
- Added support for `blake2b`.
- Added support for `scrypt`.
- Update `libsodium` to 1.0.11.
- Default to the bundled `libsodium` when compiling.
- All raised exceptions are defined mixing-in `nacl.exceptions.CryptoError`

1.0.1 - 2016-01-24

- Fix an issue with absolute paths that prevented the creation of wheels.

1.0 - 2016-01-23

- PyNaCl has been ported to use the new APIs available in cffi 1.0+. Due to this change we no longer support PyPy releases older than 2.6.
- Python 3.2 support has been dropped.
- Functions to convert between Ed25519 and Curve25519 keys have been added.

0.3.0 - 2015-03-04

- The low-level API (*nacl.c.**) has been changed to match the upstream NaCl C/C++ conventions (as well as those of other NaCl bindings). The order of arguments and return values has changed significantly. To avoid silent failures, *nacl.c* has been removed, and replaced with *nacl.bindings* (with the new argument ordering). If you have code which calls these functions (e.g. *nacl.c.crypto_box_keypair()*), you must review the new docstrings and update your code/imports to match the new conventions.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Bibliography

[SD2012] A nice overview of password hashing history is available in Solar Designer's presentation Password security: past, present, future

[PHC] The Argon2 recommendation is prominently shown in the Password Hashing Competition site, along to the special recognition shortlist and the original call for submissions.

Index

V