libnacl Documentation

Release 1.4.3

Thomas S Hatch

December 20, 2015

Contents

Contents:

CHAPTER 1

Public Key Encryption

Unlike traditional means for public key asymmetric encryption, the nacl encryption systems are very high speed. The CurveCP network protocol for instance only uses public key encryption for all transport.

Public key encryption is very simple, as is evidenced with this communication between Alice and Bob:

```
import libnacl.public

# Define a message to send
msg = b'You\'ve got two empty halves of coconut and you\'re bangin\' \'em together.'

# Generate the key pairs for Alice and bob, if secret keys already exist
# they can be passed in, otherwise new keys will be automatically generated
bob = libnacl.public.SecretKey()
alice = libnacl.public.SecretKey()

# Create the boxes, this is an object which represents the combination of the
# sender's secret key and the receiver's public key
bob_box = libnacl.public.Box(bob.sk, alice.pk)
alice_box = libnacl.public.Box(alice.sk, bob.pk)

# Bob's box encrypts messages for Alice
bob_ctxt = bob_box.encrypt(msg)
# Alice's box decrypts messages from Bob
bclear = alice_box.decrypt(bob_ctxt)
# Alice can send encrypted messages which only Bob can decrypt
alice_ctxt = alice_box.encrypt(msg)
aclear = bob_box.decrypt(alice_ctxt)
```

Note: Every encryption routine requires a nonce. The nonce is a 24 char string that must never be used twice with the same keypair. If no nonce is passed in then a nonce is generated based on random data. If it is desired to generate a nonce manually this can be done by passing it into the encrypt method.

1.1 SecretKey Object

The SecretKey object is used to manage both public and secret keys, this object contains a number of methods for both convenience and utility. The key data is also available.

1.1.1 Keys

The raw public key is available as SecretKey.sk, to generate a hex encoded version of the key the sk_hex method is available. The same items are available for the public keys:

```
import libnacl.public

fred = libnacl.public.SecretKey()

raw_sk = fred.sk
hex_sk = fred.hex_sk()

raw_pk = fred.pk
hex_pk = fred.hex_pk()
```

By saving only the binary keys in memory libnacl ensures that the minimal memory footprint is needed.

1.2 PublicKey Object

To manage only the public key end, a public key object exists:

```
import libnacl.public

tom = libnacl.public.PublicKey(tom_public_key_hex)

raw_pk = tom.pk
hex_pk = tom.hex_pk()
```

1.3 Saving Keys to Disk

All libnacl key objects can be safely saved to disk via the save method. This method changes the umask before saving the key file to ensure that the saved file can only be read by the user creating it and cannot be written to.

```
import libnacl.public

fred = libnacl.public.SecretKey()
fred.save('/etc/nacl/fred.key')
```

CHAPTER 2

Secret Key Encryption

Secret key encryption is the method of using a single key for both encryption and decryption of messages. One of the classic examples from history of secret key, or symmetric, encryption is the Enigma machine.

The SecretBox class in libnacl.secret makes this type of encryption very easy to execute:

```
msg = b'But then of course African swallows are not migratory.'
# Create a SecretBox object, if not passed in the secret key is
# Generated purely from random data
box = libnacl.secret.SecretBox()
# Messages can now be safely encrypted
ctxt = box.encrypt(msg)
# An additional box can be created from the original box secret key
box2 = libnacl.secret.SecretBox(box.sk)
# Messages can now be easily encrypted and decrypted
clear1 = box.decrypt(ctxt)
clear2 = box2.decrypt(ctxt)
ctxt2 = box2.encrypt(msg)
clear3 = box.decrypt(ctxt2)
```

Note: Every encryption routine requires a nonce. The nonce is a 24 char string that must never be used twice with the same keypair. If no nonce is passed in then a nonce is generated based on random data. If it is desired to generate a nonce manually this can be done by passing it into the encrypt method.

CHAPTER 3

Signing and Verifying Messages

The nacl libs have the capability to sign and verify messages. Please be advised that public key encrypted messages do not need to be signed, the nacl box construct verifies the validity of the sender.

To sign and verify messages use the Signer and Verifier classes:

```
import libnacl.sign

msg = (b'Well, that\'s no ordinary rabbit.  That\'s the most foul, '
       b'cruel, and bad-tempered rodent you ever set eyes on.')
# Create a Signer Object, if the key seed value is not passed in the
# signing keys will be automatically generated
signer = libnacl.sign.Signer()
# Sign the message, the signed string is the message itself plus the
# signature
signed = signer.sign(msg)
# If only the signature is desired without the message:
signature = signer.signature(msg)
# To create a verifier pass in the verify key:
veri = libnacl.sign.Verifier(signer.hex_vk())
# Verify the message!
verified = veri.verify(signed)
verified2 = veri.verify(signature + msg)
```

3.1 Saving Keys to Disk

All libnacl key objects can be safely saved to disk via the save method. This method changes the umask before saving the key file to ensure that the saved file can only be read by the user creating it and cannot be written to.

```
import libnacl.sign

signer = libnacl.sign.Signer()
signer.save('/etc/nacl/signer.key')
```

CHAPTER 4

Dual Key Management

The libnacl library abstracts a "Dual Key" model. The Dual Key creates a single key management object that can be used for both signing and encrypting, it generates and maintains a Curve25519 encryption key pair and an ED25519 signing keypair. All methods for encryption and signing work with and from Dual Keys.

To encrypt messages using Dual Keys:

```
import libnacl.dual

# Define a message to send
msg = b"You've got two empty halves of coconut and you're bangin' 'em together."

# Generate the key pairs for Alice and bob, if secret keys already exist
# they can be passed in, otherwise new keys will be automatically generated
bob = libnacl.dual.DualSecret()
alice = libnacl.dual.DualSecret()

# Create the boxes, this is an object which represents the combination of the
# sender's secret key and the receiver's public key
bob_box = libnacl.public.Box(bob.sk, alice.pk)
alice_box = libnacl.public.Box(alice.sk, bob.pk)

# Bob's box encrypts messages for Alice
bob_ctxt = bob_box.encrypt(msg)
# Alice's box decrypts messages from Bob
bclear = alice_box.decrypt(bob_ctxt)
# Alice can send encrypted messages which only Bob can decrypt
alice_ctxt = alice_box.encrypt(msg)
aclear = alice_box.decrypt(alice_ctxt)
```

Note: Every encryption routine requires a nonce. The nonce is a 24 char string that must never be used twice with the same keypair. If no nonce is passed in then a nonce is generated based on random data. If it is desired to generate a nonce manually this can be done by passing it into the encrypt method.

4.1 DualKey Object

The DualKey object is used to manage both public and secret keys, this object contains a number of methods for both convenience and utility. The key data is also available.

4.1.1 Keys

The raw public key is available as DualKey.pk, to generate a hex encoded version of the key the pk_hex method is available:

```
import libnacl.dual

fred = libnacl.dual.DualKey()

raw_sk = fred.sk
hex_sk = fred.hex_sk()

raw_pk = fred.pk
hex_pk = fred.hex_pk()
```

By saving only the binary keys in memory libnacl ensures that the minimal memory footprint is needed.

4.2 Saving Keys to Disk

All libnacl key objects can be safely saved to disk via the save method. This method changes the umask before saving the key file to ensure that the saved file can only be read by the user creating it and cannot be written to. When using dual keys the encrypting and signing keys will be saved togather in a single file.

```
import libnacl.dual

fred = libnacl.dual.DualKey()
fred.save('/etc/nacl/fred.key')
```

CHAPTER 5

Utility Functions

The libnacl system comes with a number of utility functions, these functions are made available to make some of the aspects of encryption and key management easier. These range from nonce generation to loading saved keys.

5.1 Loading Saved Keys

After keys are saved using the key save method reloading the keys is easy. The *libnacl.utils.load_key* function will detect what type of key object saved said key and then create the object from the key and return it.

```
import libnacl.utils

key_obj = libnacl.utils.load_key('/etc/keys/bob.key')
```

The load_key and save routines also support inline key serialization. The default is json but msgpack is also supported.

5.2 Salsa Key

A simple function that will return a random byte string suitable for use in SecretKey encryption.

```
import libnacl.utils

key = libnacl.utils.salsa_key()
```

This routine is only required with the raw encryption functions, as the *libnacl.secret.SecretBox* will generate the key automatically.

5.3 Nonce Routines

A few functions are available to help with creating nonce values, these routines are available because there is some debate about what the best approach is.

We recommend a pure random string for the nonce which is returned from *rand_nonce*, but some have expressed a desire to create nonces which are designed to avoid re-use by more than simply random data and therefore the *time_nonce* function is also available.

CHAPTER 6

Raw Public Key Encryption

Note: While these routines are perfectly safe, higher level convenience wrappers are under development to make these routines easier.

Public key encryption inside the nacl library has been constructed to ensure that all cryptographic routines are executed correctly and safely.

The public key encryption is executed via the functions which begin with *crypto_box* and can be easily executed.

First generate a public key and secret key keypair for the two communicating parties, who for tradition's sake, will be referred to as Alice and Bob:

```
import libnacl

alice_pk, alice_sk = libnacl.crypto_keypair()
bob_pk, bob_sk = libnacl.crypto_keypair()
```

Once the keys have been generated a cryptographic box needs to be created. The cryptographic box takes the party's secret key and the receiving party's public key. These are used to create a message which is both signed and encrypted.

Before creating the box a nonce is required. The nonce is a 24 character string which should only be used for this message, the nonce should never be reused. This means that the nonce needs to be generated in such a way that the probability of reusing the nonce string with the same keypair is very low. The libnacl wrapper ships with a convenience function which generates a nonce from random bytes:

```
import libnacl.utils
nonce = libnacl.utils.rand_nonce()
```

Now, with a nonce a cryptographic box can be created, Alice will send a message:

```
msg = 'Quiet, quiet.  Quiet!  There are ways of telling whether she is a witch.'
box = libnacl.crypto_box(msg, nonce, bob_pk, alice_sk)
```

Now with a box in hand it can be decrypted by Bob:

```
clear_msg = libnacl.crypto_box_open(box, nonce, alice_pk, bob_sk)
```

The trick here is that the box AND the nonce need to be sent to Bob, so he can decrypt the message. The nonce can be safely sent to Bob in the clear.

To bring it all together:

```
import libnacl
import libnacl.utils
```

```
alice_pk, alice_sk = libnacl.crypto_keypair()
bob_pk, bob_sk = libnacl.crypto_keypair()

nonce = libnacl.utils.rand_nonce()

msg = 'Quiet, quiet.  Quiet!  There are ways of telling whether she is a witch.'
box = libnacl.crypto_box(msg, nonce, bob_pk, alice_sk)

clear_msg = libnacl.crypto_box_open(box, nonce, alice_pk, bob_sk)
```

CHAPTER 7

Raw Secret Key Encryption

Note: While these routines are perfectly safe, higher level convenience wrappers are under development to make these routines easier.

Secret key encryption is high speed encryption based on a shared secret key.

Note: The nacl library uses the salsa20 stream encryption cipher for secret key encryption, more information about the salsa20 cipher can be found here: http://cr.yp.to/salsa20.html

The means of encryption assumes that the two sides of the conversation both have access to the same shared secret key. First generate a secret key, libnacl provides a convenience function for the generation of this key called libnacl.utils.salsa_key, then generate a nonce, a new nonce should be used every time a new message is encrypted. A convenience function to create a unique nonce based on random bytes:

```
import libnacl
import libnacl.utils

key = libnacl.utils.salsa_key()
nonce = libnacl.utils.rand_nonce()
```

With the key and nonce in hand, the cryptographic secret box can now be generated:

```
msg = 'Who are you who are so wise in the ways of science?'
box = libnacl.crypto_secretbox(msg, nonce, key)
```

Now the message can be decrypted on the other end. The nonce and the key are both required to decrypt:

```
clear_msg = libnacl.crypto_secretbox_open(box, nonce, key)
```

When placed all together the sequence looks like this:

```
import libnacl
import libnacl.utils

key = libnacl.utils.salsa_key()
nonce = libnacl.utils.rand_nonce()

msg = 'Who are you who are so wise in the ways of science?'
box = libnacl.crypto_secretbox(msg, nonce, key)

clear_msg = libnacl.crypto_secretbox_open(box, nonce, key)
```

CHAPTER 8

Raw Message Signatures

Note: While these routines are perfectly safe, higher level convenience wrappers are under development to make these routines easier.

Signing messages ensures that the message itself has not been tampered with. The application of a signature to a message is something that is is automatically applied when using the public key encryption and is not a required step when sending encrypted messages. This document however is intended to illustrate how to sign plain text messages.

The nacl libs use a separate keypair for signing then is used for public key encryption, it is a high performance key signing algorithm called ed25519, more information on ed25519 can be found here: http://ed25519.cr.yp.to/

The sign messages first generate a signing keypair, this constitutes the signing key which needs to be kept secret, and the verify key which is made available to message recipients.

```
import libnacl

vk, sk = libnacl.crypto_sign_keypair()
```

With the signing keypair in hand a message can be signed:

```
msg = 'And that, my liege, is how we know the Earth to be banana-shaped.'
signed = libnacl.crypto_sign(msg, sk)
```

The signed message is really just the plain text of the message prepended with the signature. The crypto_sign_open function will read the signed message and return me original message without the signature:

```
orig = libnacl.crypto_sign_open(signed, vk)
```

Put all together:

```
import libnacl

vk, sk = libnacl.crypto_sign_keypair()

msg = 'And that, my liege, is how we know the Earth to be banana-shaped.'
signed = libnacl.crypto_sign(msg, sk)

orig = libnacl.crypto_sign_open(signed, vk)
```

CHAPTER 9

Raw Hash Functions

The nacl library comes with sha256 and sha512 hashing libraries. They do not seem to offer any benefit over python's hashlib, but for completeness they are included. Creating a hash of a message is very simple:

```
import libnacl

msg = 'Is there someone else up there we could talk to?'
h_msg = libnacl.crypto_hash(msg)
```

crypto_hash defaults to sha256, sha512 is also available:

```
import libnacl

msg = 'Is there someone else up there we could talk to?'
h_msg = libnacl.crypto_hash_sha512(msg)
```

CHAPTER 10

Raw Generic Hash (Blake2b) Functions

The nacl library comes with blake hashing libraries.

More information on Blake can be found here: https://blake2.net

The blake2b hashing algorithm is a keyed hashing algorithm, which allows for a key to be associated with a hash. Blake can be executed with or without a key.

With a key (they key can should be between 16 and 64 bytes):

```
import libnacl

msg = 'Is there someone else up there we could talk to?'
key = libnacl.randombytes(32)
h_msg = libnacl.crypto_generichash(msg, key)
```

Without a key:

```
import libnacl

msg = 'Is there someone else up there we could talk to?'
h_msg = libnacl.crypto_generichash(msg)
```

CHAPTER 11

Release notes

11.1 libnacl 1.0.0 Release Notes

This is the first stable release of libnacl, the python bindings for Daniel J. Bernstein's nacl library via libsodium.

11.1.1 NaCl Base Functions

This release features direct access to the underlying functions from nacl exposed via importing libnacl. These functions are fully documented and can be safely used directly.

11.2 libnacl 1.1.0 Release Notes

This release introduces the addition of high level classes that make using NaCl even easier.

11.2.1 High level NaCl

The addition of the high level classes give a more pythonic abstraction to using the underlying NaCl cryptography.

These classes can be found in libnacl.public, libnacl.sign and libnacl.secret.

11.2.2 Easy Nonce Generation

The new classes will automatically generate a nonce value per encrypted message. The default nonce which is generated can be found in *libnacl.utils.time_nonce*.

11.3 libnacl 1.2.0 Release Notes

This release introduces the DualKey class, secure key saving and loading, as well as enhancements to the time_nonce function.

11.3.1 Dual Key Class

Dual Keys are classes which can encrypt and sign data. These classes generate and maintain both Curve25519 and Ed25519 keys, as well as all methods for both encryption and signing.

11.3.2 Time Nonce Improvements

The original time nonce routine used the first 20 chars of the 24 char nonce for the microsecond timestamp (based on salt's jid), leaving 4 chars for random data. This new nonce uses far fewer chars for the timestamp by hex encoding the float of microseconds into just 13 chars, leaving 11 chars of random data. This makes the default nonce safer and more secure.

11.4 libnacl 1.3.0 Release Notes

This release removes the time_nonce function and replaces it with the rand_nonce function.

11.5 libnacl 1.3.1 Release Notes

Bring back a safe time_nonce function.

11.6 libnacl 1.3.2 Release Notes

Add detection of the libsodium.so.10 lib created by libsodium 0.6

11.7 libnacl 1.3.3 Release Notes

Fix issue and add tests for bug where saving and loading a signing key caused a stack trace, se issue #18

11.8 libnacl 1.3.4 Release Notes

- Change the default ctype values to be more accurate and efficient
- Update soname detection on Linux for libsodium 0.7.0
- Make soname detection a little more future proof

11.9 libnacl 1.4.0 Release Notes

11.9.1 Blake Hash Support

Initial support has been added for the blake2b hash algorithm

11.9.2 Misc Fixes

- Fix issue with keyfile saves on windows
- Fix libsodium detection for Ubuntu manual installs and Windows dll detection

11.10 libnacl 1.4.1 Release Notes

11.10.1 Misc Fixes

- Fix for crypto_auth_verify and crypto_auth_onetimeverify
- Lint fixes and updates

11.11 libnacl 1.4.2 Release Notes

11.11.1 SecretBox key save and load

- Add support to save and load SecretBox keys

11.12 libnacl 1.4.3 Release Notes

11.12.1 crypto_onetimeauth_verify fixes

- Fix a call to the crypto_onetimeauth_verify routine into the right libsodium system
- Add tests for crypto_onetimeauth_verify

11.12.2 Improved support for MacOSX

- Improved the lookup procedure for finding libsodium on MacOSX

11.12.3 Add support for reading file streams for key loading

CHAPTER 12

Indices and tables

- genindex
- modindex
- search