

---

# Neustar TDI Documentation

*Release 0.20.0*

**Neustar**

**Aug 04, 2017**



---

# Contents

---

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Installation . . . . .	3
1.1.1	Setup & Installation . . . . .	3
1.2	Tutorials . . . . .	4
1.2.1	Tutorials . . . . .	4
1.3	API . . . . .	9
1.3.1	API . . . . .	9
1.4	Contributing . . . . .	20
1.4.1	Contributing . . . . .	20
<b>2</b>	<b>Indices and tables</b>	<b>23</b>
	<b>Python Module Index</b>	<b>25</b>



## Release 0.20.0

Neustar TDI (formerly oneID-connect) is an authentication framework for the Internet of Things (IoT), servers and end-users. By sending messages with digital signatures, you can authenticate the origin of the message and ensure the message hasn't been tampered with. Neustar TDI makes it simple for projects that need to send authenticated messages and verify the authentication of messages.

Neustar-TDI/python-sdk can be installed on IoT devices and servers that support python 2.7, 3.4, 3.5 or 3.6. Neustar-TDI/python-sdk depends on two external libraries: the cryptography.io python package and openssl.



Security for IoT devices can be challenging to implement and difficult to manage at scale. So Neustar has created a secure two-factor mutual authentication platform that securely connects users to their IoT devices, while enabling product servers to securely send firmware updates to those same IoT devices. We do this using state of the art [Elliptical Curve](#) cryptography.

## Installation

### Setup & Installation

#### IoT Device Setup

#### Installation for Intel Edison

First update to the latest [Yocto firmware update](#) After you've flashed and configured your Intel Edison, we can setup the Intel Edison with pip.

**Warning:** Please make sure that your Intel Edison is connected to the internet before continuing.

- Open the base-feeds config file:

```
$ vi /etc/opkg/base-feeds.conf
```

- Add the following repositories:

```
$ src/gz all http://repo.opkg.net/edison/repo/all
$ src/gz edison http://repo.opkg.net/edison/repo/edison
$ src/gz core2-32 http://repo.opkg.net/edison/repo/core2-32
```

- Update opkg:

```
$ opkg update
```

- Install python pip:

```
$ opkg install python-pip
```

- Install the SDK:

```
$ pip install oneid-connect
```

## Server Setup

### Installation for Mac OS X

The cryptography.io requirement is a statically linked build for Yosemite and above. You only need one step:

```
$ pip install oneid-connect
```

If you're using an older version of OS X or want to link to a specific version of openssl you will need a C compiler, development headers and possibly even the openssl Library. This is all provided by Apple's Xcode development tools.

```
$ xcode-select --install
```

This will install a compiler and the development libraries required.

### Installation for Ubuntu

The Python SDK depends on two external libraries, cryptography.io and openssl. cryptography.io is a library that exposes cryptographic primitives from openssl.

The SDK should build easily on most Linux distributions that have a C compiler, openssl headers and the libffi libraries.

```
$ sudo apt-get install build-essential libssl-dev libffi-dev python-dev
```

You should now be able to install the SDK with the usual

```
$ pip install oneid-connect
```

## Tutorials

### Tutorials

#### Hello World

Here is a simple "hello world" message with a digital signature and verified.

Before we can sign or verify any messages, we first need to create a secret key.



```

from oneid import service
# Directory to save the secret key (should be secure enclave)
secret_key_pem_path = '/Users/me/my_secret_key.pem'
service.create_secret_key(output=secret_key_pem_path)

```

You should now have a secret key pem file that begins with -----BEGIN PRIVATE KEY-----

Now we can create our “hello world” message and sign it.

```

from oneid.keychain import Keypair

message = 'hello world'

my_key = Keypair.from_secret_pem(path=secret_key_pem_path)
signature = my_key.sign(message)
print(signature)

```

To verify the signature, we need to pass in the message and the signature back into the Keypair.

```

my_key.verify(message, signature)

```

That’s it!

If you want to see what happens if the message has been tampered with, replace `hello world` with something else like `hello universe`.

```

# raises InvalidSignature
my_key.verify('hello universe', signature)

```

## Sending Two-Factor Authenticated Firmware update to IoT Device

Sending a firmware update to all of your devices should always be secure. The last thing you want is a malicious update sent to your entire fleet of devices.

For this example, we’re going to use Neustar TDI’s two-factor authentication service. Neustar TDI’s two-factor authentication service enables you to manage all your servers and IoT devices. If a server or IoT device has been compromised or taken out of commission, you can easily revoke it’s signing permissions.

Before we begin, you will need `oneID-cli` and a [Neustar TDI developer account](#).

```

$ pip install oneid-cli

```

## Intro to Neustar TDI’s Two-Factor Authentication

Two-factor means that there will be two signatures for each message. **BOTH** signatures must be verified before reading the message. Since there are two signatures that need to be verified on the IoT device, the IoT device will need to store two public keys that will be used for message verification. Neustar TDI will provide you with both of these public keys for the IoT device.

### Steps:

1. Server prepares a message for the IoT device and signs it.
2. Server makes a two-factor authentication request to Neustar TDI with the prepared message.

3. Neustar TDI verifies the server's identity and responds with Neustar TDI's signature for the message.
4. Server then re-signs the message with the shared Project key.
5. Server sends the message with the two signatures to the IoT device.
6. IoT device verifies **BOTH** signatures.

### Setup

First we need to configure your terminal.

```
$ oneid-cli configure
```

This will prompt you for your `ACCESS_KEY`, `ACCESS_SECRET`, and `ONEID_KEY`. You can find all these in your [Neustar TDI developer console](#)

### Creating a Project

All users, servers and edge devices need to be associated with a Project. Let's create a new Project.

```
$ oneid-cli create-project --name "my epic project"
```

This will prompt you to generate the public/private key pair for the Project. Answer 'Y' at this step. You will be given the Project ID and three keys. The first key is a Neustar TDI verification public key. The second is the Project verification public key. The third is your Project **SECRET** key.

**Danger:** SAVE THE PROJECT SECRET KEY IN A SAFE PLACE. If you lose this key, you will lose your ability to send authenticated messages to your devices.

The Neustar TDI verification public key will be given to all your edge devices and used to verify messages sent from a server.

In the following steps, we will assume a Project ID of "d47fedd0-729f-4941-b4bd-2ec4fe0f9ca9". You should substitute the one you get back from `oneid-cli create-project`.

### Server

The firmware update message we will send to the IoT devices will be very simple. The message will be a url to the CDN where the firmware update is hosted and a checksum the IoT device will use to verify the download.

Before we can sign any messages, we need to give the server an identity Neustar TDI can verify.

```
$ oneid-cli provision --project-id d47fedd0-729f-4941-b4bd-2ec4fe0f9ca9 --name "IoT_↵  
↵server" --type server
```

This will generate a new **SECRET** .pem file.

**Danger:** PLEASE STORE SECRET FILES IN A SAFE PLACE. Never post them in a public forum or give them to anyone.

If you created the server secret key on your personal computer, we need to copy it over to the server along with the Project key that was generated when you first created the Project.

```
$ scp /Users/me/secret/server_secret.pem ubuntu@10.1.2.3:/home/www/server_secret.pem
$ scp /Users/me/secret/project_secret.pem ubuntu@10.1.2.3:/home/www/project_secret.pem
$ scp /Users/me/secret/oneid_public.pem ubuntu@10.1.2.3:/home/www/oneid_public.pem
```

In Python, we're just going to hardcode the path to these keys for quick access.

```
import json
import logging

from oneid.keychain import Keypair, Credentials
from oneid.session import ServerSession

logging.basicConfig(level=logging.WARNING)

logger = logging.getLogger('fw_update.py')

# Unique Project ID provided by Neustar TDI
PROJECT_ID = 'b7f276d1-6c86-4f57-85e8-70105316225b'
PROJECT_PROJECT_ID = 'project/' + PROJECT_ID

# Unique Server ID,
SERVER_ID = '709ec376-7e8c-40fc-94ee-14887023c885'

# Secret keys we downloaded from Neustar TDI Developer Portal
server_secret_key_path = (
    './project-{pid}/server-{sid}/server-{sid}-priv.pem'.format(
        pid=PROJECT_ID, sid=SERVER_ID
    )
)
project_secret_key_path = (
    './project-{pid}/project-{pid}-priv.pem'.format(
        pid=PROJECT_ID, sid=SERVER_ID
    )
)

server_key = Keypair.from_secret_pem(path=server_secret_key_path)
server_key.identity = SERVER_ID
server_credentials = Credentials(SERVER_ID, server_key)

project_key = Keypair.from_secret_pem(path=project_secret_key_path)
project_key.identity = PROJECT_PROJECT_ID
project_credentials = Credentials(PROJECT_ID, project_key)

server_session = ServerSession(
    identity_credentials=server_credentials,
    project_credentials=project_credentials
)

device_msg = server_session.prepare_message(
    download_url='http://mycompany.com/firmwareupdate',
    checksum=0xdeadbeef,
)
logger.debug('device_msg=%s', device_msg)
```

The final step is to send the two-factor authenticated\_msg to the IoT device. You can use any network protocol you want, or a messaging protocol such as MQTT, RabbitMQ, Redis etc.

## IoT Device

Just like we did with the server, we need to provision our IoT device.

```
$ oneid-cli provision --project-id d47fedd0-729f-4941-b4bd-2ec4fe0f9ca9 --name "my_
↪edge device" --type edge_device
```

Now we need to copy over the Neustar TDI verifier key, Project verifier key and the new device secret key. The Neustar TDI verifier key can be downloaded from the [Neustar TDI developer console](#).

You can print out your Project verifier key by adding a snippet to the previous code example.

```
import base64
project_verifier = base64.b64encode(project_key.public_key_der)
print(project_verifier)
```

If you can SSH into your IoT device, you can do the same thing that we did with the server and copy over the device identity secret key. Since the Neustar TDI and Project verifier keys are static for all devices in a Project, we can hard code them in.

```
$ scp /Users/me/secret/device_secret.pem edison@10.1.2.3:/home/root/device_secret.pem
```

Now that we have the message that was sent to the IoT device, let's check the message's authenticity by verifying the digital signatures.

```
from oneid.keychain import Keypair, Credentials
from oneid.session import DeviceSession

oneid_public_key_path = './oneid-pub.pem'
oneid_keypair = Keypair.from_public_pem(path=oneid_public_key_path)
oneid_keypair.identity = PROJECT_ID

project_public_key_path = './project-pub.pem'
project_keypair = Keypair.from_public_pem(path=project_public_key_path)
project_keypair.identity = PROJECT_PROJECT_ID

device_session = DeviceSession(
    project_credentials=Credentials(
        identity=project_keypair.identity,
        keypair=project_keypair
    ),
    oneid_credentials=Credentials(
        identity=oneid_keypair.identity,
        keypair=oneid_keypair
    )
)

try:
    claims = device_session.verify_message(device_msg)
    msg = json.loads(claims.get('message', '{}'))

    logger.debug('claims=%s', claims)
    print('Success!')
    print('  URL={}'.format(msg.get('download_url')))
    print('  checksum=0x{:08x}'.format(msg.get('checksum')))
except:
    print('Failed.')
    logger.warning('error: ', exc_info=True)
```

## API

### API

#### oneid.keychain

##### Credentials

**class** `oneid.keychain.Credentials` (*identity, keypair*)  
 Container for User/Server/Device Encryption Key, Signing Key, Identity

##### Variables

- **identity** – UUID of the identity.
- **keypair** – BaseKeypair instance.

##### ProjectCredentials

**class** `oneid.keychain.ProjectCredentials` (*project\_id, keypair, encryption\_key*)

**encrypt** (*plain\_text*)

Encrypt plain text with the project encryption key.

**Parameters** **plain\_text** – String or bytes to encrypt with project encryption key.

**Returns** Dictionary with cipher text and encryption params.

**decrypt** (*cipher\_text*)

Decrypt cipher text that was encrypted with the project encryption key

**Parameters** **cipher\_text** – Encrypted dict as returned by :py:encrypt:

**Returns** plain text

**Return\_type** bytes

##### Keypair

**class** `oneid.keychain.Keypair` (*\*args, \*\*kwargs*)

**secret\_as\_der**

Write out the private key as a DER format

**Returns** DER encoded private key

**secret\_as\_pem**

Write out the private key as a PEM format

**Returns** Pem Encoded private key

**classmethod** **from\_secret\_pem** (*key\_bytes=None, path=None*)

Create a *Keypair* from a PEM-formatted private ECDSA key

**Returns** *Keypair* instance

**classmethod from\_public\_pem** (*key\_bytes=None, path=None*)  
Create a *Keypair* from a PEM-formatted public ECDSA key

Note that this keypair will not be capable of signing, only verifying.

**Returns** *Keypair* instance

**classmethod from\_secret\_der** (*der\_key*)  
Read a *der\_key*, convert it a private key

**Parameters** *path* – der formatted key

**Returns**

**classmethod from\_public\_der** (*public\_key*)  
Given a DER-format public key, convert it into a token to validate signatures

**Parameters** *public\_key* – der formatted key

**Returns** *Keypair* instance

**classmethod from\_jwk** (*jwk*)  
Create a *Keypair* from a JWK

**Parameters** *jwk* – oneID-standard JWK

**Returns** *Keypair* instance

**Raises** *InvalidFormatError* – if not a valid JWK

**jwk**

The keys as a JSON Web Key (JWK) Private key will be included only if present

**Returns** oneID-standard JWK

**jwk\_public**

The public key as a JSON Web Key (JWK)

**Returns** oneID-standard JWK

**jwk\_private**

The private key as a JSON Web Key (JWK)

**Returns** oneID-standard JWK

**Raises** *InvalidFormatError* – if not a private key

**verify** (*payload, signature*)

Verify that the token signed the data

**Parameters**

- **payload** (*String*) – message that was signed and needs verified
- **signature** (*Base64 URL Safe*) – Signature that can verify the sender's identity and payload

**Returns**

**sign** (*payload*)

Sign a payload

**Parameters** *payload* – String (usually jwt payload)

**Returns** URL safe base64 signature

**ecdh** (*peer\_keypair, algorithm='A256GCM', party\_u\_info=None, party\_v\_info=None*)

Derive a shared symmetric key for encrypting data to a given recipient

**Parameters**

- **peer\_keypair** (*Keypair*) – Public key of the recipient
- **algorithm** (*str*) – The algorithm associated with the operation (defaults to 'A256GCM')
- **party\_u\_info** (*str or bytes*) – shared identifying information about the sender (optional)
- **party\_v\_info** (*str or bytes*) – shared identifying information about the recipient (optional)

**Returns** a 256-bit encryption key

**Return\_type** bytes

**Raises** *InvalidFormatError* – if self is not a private key

**public\_key**

If the private key is defined, generate the public key

**Returns**

**public\_key\_der**

DER formatted public key

**Returns** Public Key in DER format

**public\_key\_pem**

PEM formatted public key

**Returns** Public Key in PEM format

**oneid.service**

Provides useful functions for interacting with the Neustar TDI Core API, including creation of keys, etc.

`oneid.service.create_secret_key` (*output=None*)

Create a secret key and save it to a secure location

**Parameters** **output** – Path to save the secret key

**Returns** `oneid.keychain.Keypair`

`oneid.service.create_aes_key` ()

Create an AES256 key for symmetric encryption

**Returns** Encryption key bytes

`oneid.service.encrypt_attr_value` (*attr\_value, aes\_key, legacy\_support=True*)

Convenience method to encrypt attribute properties

**Parameters**

- **attr\_value** – plain text (string or bytes) that you want encrypted
- **aes\_key** – symmetric key to encrypt attribute value with

**Returns** Dictionary (Flattened JWE) with base64-encoded ciphertext and base64-encoded iv

`oneid.service.decrypt_attr_value` (*attr\_ct, aes\_key*)

Convenience method to decrypt attribute properties

**Parameters**

- **attr\_ct** – Dictionary (may be a Flattened JWE) with base64-encoded ciphertext and base64-encoded iv
- **aes\_key** – symmetric key to decrypt attribute value with

**Returns** plaintext bytes

## oneid.session

### SessionBase

**class** `oneid.session.SessionBase` (*identity\_credentials=None*, *project\_credentials=None*,  
*oneid\_credentials=None*, *peer\_credentials=None*, *config=None*)

Abstract Session Class

#### Variables

- **identity\_credentials** – TDI identity *Credentials*
- **project\_credentials** – unique project credentials *Credentials*
- **oneid\_credentials** – TDI project credentials *Credentials*
- **oneid\_credentials** – peer credentials *Credentials*
- **config** – Dictionary or configuration keyword arguments

### DeviceSession

**class** `oneid.session.DeviceSession` (*identity\_credentials=None*, *project\_credentials=None*,  
*oneid\_credentials=None*, *peer\_credentials=None*, *config=None*)

**verify\_message** (*message*, *rekey\_credentials=None*)

Verify a message received from the Project

#### Parameters

- **message** – JSON formatted JWS with at least two signatures
- **rekey\_credentials** – List of *Credential*

**Returns** verified message or False if not valid

**prepare\_message** (*encrypt\_to\_peers=True*, *other\_recipients=None*, *\*args*, *\*\*kwargs*)

Prepare a message before sending

#### Parameters

- **encrypt\_to\_peers** (*bool*) – If True (default), and *peer\_credentials* available, encrypt the message to them
- **other\_recipients** (list of *Credential*) – Additional recipients to encrypt to

**Returns** Signed JWS

**add\_signature** (*message*, *default\_jwt\_kid=None*)

Add this Device's signature to a message

Note that the semantics of this signature are application-specific. If the application expects only verified messages to be co-signed, the caller is responsible for verifying the message first. Otherwise, the signature only indicates that the message was processed by this Device.



Likewise, this method will not decrypt a JWE. If the message was encrypted for this Device, and should be decrypted and re-signed, the caller should do that through other means, such as `verify_message()` and `prepare_message()`.

#### Parameters

- **message** (*str*) – Previously-signed JWS (Compact or JSON) or JWT
- **default\_jwt\_kid** (*str*) – (optional) value for ‘kid’ header field if passing a JWT without one

**Returns** Signed JWS with additional Device signature

## ServerSession

```
class oneid.session.ServerSession (identity_credentials=None, project_credentials=None,
                                   oneid_credentials=None, peer_credentials=None,
                                   config=None)
```

Enable Server to request two-factor Authentication from TDI Core

```
prepare_message (rekey_credentials=None, encrypt_to_peers=True, other_recipients=None,
                 **kwargs)
```

Build message that has two-factor signatures

#### Parameters

- **rekey\_credentials** (*list*) – (optional) rekey credentials
- **encrypt\_to\_peers** (*bool*) – If True (default), and peer\_credentials available, encrypt the message to them
- **other\_recipients** (*list of Credential*) – Additional recipients to encrypt to

**Returns** Signed JWS to be sent to devices

```
verify_message (message, device_credentials, get_oneid_cosignature=True)
```

Verify a message received from/through one or more Devices

#### Parameters

- **message** – JSON formatted JWS or JWT signed by the Device
- **device\_credentials** – *Credential* (or list of them) to verify Device signature(s) against
- **get\_oneid\_cosignature** – (default: True) verify with TDI Core first

**Returns** verified message or False if not valid

## AdminSession

```
class oneid.session.AdminSession (identity_credentials, project_credentials=None,
                                   oneid_credentials=None, config=None)
```

Admin Users will only interface with TDI Core service, They only need an `identity_credentials` and `oneid_credentials` to verify responses

## oneid.jwt

Provides useful functions for dealing with JWTs and JWSs

Based on the [JWT](#) and [JWS](#) IETF RFCs.

`oneid.jwts.make_jwt(raw_claims, keypair, json_encoder=<function dumps>)`

Convert claims into JWT

### Parameters

- **raw\_claims** (*dict*) – payload data that will be converted to json
- **keypair** – *Keypair* to sign the request
- **json\_encoder** – a function to encode a *dict* into JSON. Defaults to *json.dumps*

Returns JWT

`oneid.jwts.verify_jwt(jwt, keypair=None, json_decoder=<function loads>)`

Convert a JWT back to its claims, if validated by the *Keypair*

### Parameters

- **jwt** (*str* or *bytes*) – JWT to verify and convert
- **keypair** (*Keypair*) – *Keypair* to verify the JWT
- **json\_decoder** – a function to decode JSON into a *dict*. Defaults to *json.loads*

Returns claims

Return type *dict*

Raises *InvalidFormatError* if not a valid JWT

Raises *InvalidAlgorithmError* if unsupported algorithm specified

Raises *InvalidClaimsError* if missing or invalid claims, including expiration, re-used nonce, etc.

Raises *InvalidSignatureError* if signature is not valid

`oneid.jwts.make_jws(raw_claims, ordered_keypairs, multiple_sig_headers=None, json_encoder=<function dumps>)`

Convert claims into JWS format (compact or JSON)

### Parameters

- **raw\_claims** (*dict*) – payload data that will be converted to json
- **ordered\_keypairs** (*list*) – *Keypairs* to sign the request with (in signing order, with *ordered\_keypairs[0]* the first to sign the JWS)
- **json\_encoder** – a function to encode a *dict* into JSON. Defaults to *json.dumps*
- **multiple\_sig\_headers** (*list*) – optional list of headers for associated keypairs with the same list index

Returns JWS

`oneid.jwts.extend_jws_signatures(jws, ordered_keypairs, default_jwt_kid=None, multiple_sig_headers=None, json_encoder=<function dumps>, json_decoder=<function loads>)`

Add signatures to an existing JWS (or JWT)

### Parameters

- **jws** (*str*) – existing JWS (Compact or JSON) or JWT
- **ordered\_keypairs** (*list*) – *Keypairs* to sign the request with (in signing order, with `ordered_keypairs[0]` the first to sign the JWS)
- **default\_jwt\_kid** (*str*) – (optional) value for ‘kid’ header field if passing a JWT without one
- **json\_encoder** – a function to encode a `dict` into JSON. Defaults to `json.dumps`
- **json\_decoder** – a function to decode JSON into a `dict`. Defaults to `json.loads`
- **multiple\_sig\_headers** (*list*) – optional list of headers for associated keypairs with the same list index

**Returns** JWS

```
oneid.jwt.remove_jws_signatures(jws, kids_to_remove, json_encoder=<function dumps>,
                                json_decoder=<function loads>)
```

Remove signatures from an existing JWS

**Parameters**

- **jws** (*str*) – existing JWS (JSON format only)
- **kids\_to\_remove** (*list*) – Keypair identities to remove
- **json\_encoder** – a function to encode a `dict` into JSON. Defaults to `json.dumps`
- **json\_decoder** – a function to decode JSON into a `dict`. Defaults to `json.loads`

**Returns** JWS (may have empty signature list if last one removed)

```
oneid.jwt.get_jws_key_ids(jws, default_kid=None, json_decoder=<function loads>,
                          ordered=False)
```

Extract the IDs of the keys used to sign a given JWS

**Parameters**

- **jws** (*str or bytes*) – JWS to get key IDs from
- **default\_kid** (*str*) – Value to use for looking up keypair if no *kid* found in a given signature header, as may happen when extending a JWT
- **json\_decoder** – a function to decode JSON into a `dict`. Defaults to `json.loads`
- **ordered** – Bool if the key IDs should be returned in the order they signed the JWS. If this cannot be resolved, an exception is thrown.

**Returns** key IDs

**Return type** list

**Raises** *InvalidFormatError*: if not a valid JWS

```
oneid.jwt.verify_jws(jws, keypairs=None, verify_all=True, default_kid=None,
                    json_decoder=<function loads>)
```

Convert a JWS back to its claims, if validated by a set of required *Keypairs*

**Parameters**

- **jws** (*str or bytes*) – JWS to verify and convert
- **keypairs** (*list*) – *Keypairs* to verify the JWS with. Must include one for each specified in the JWS headers’ *kid* values.

- **verify\_all** (*bool*) – If True (default), all keypairs must validate a signature. If False, only one needs to. If any fail to validate, the JWS is still not validated. This allows the caller to send multiple keys that `_might_` have corresponding signatures, without requiring that `_all_` do.
- **default\_kid** (*str*) – Value to use for looking up keypair if no *kid* found in a given signature header, as may happen when extending a JWT
- **json\_decoder** – a function to decode JSON into a `dict`. Defaults to `json.loads`

**Returns** `claims`

**Return type** `dict`

**Raises** `InvalidFormatError`: if not a valid JWS

**Raises** `InvalidAlgorithmError`: if unsupported algorithm specified

**Raises** `InvalidClaimsError`: if missing or invalid claims, including expiration, re-used nonce, etc.

**Raises** `InvalidSignatureError`: if any relevant signature is not valid

`oneid.jwt`s.**get\_jws\_headers** (*jws*, *json\_decoder*=<function loads>)

Extract the headers of the signatures used to sign a given JWS

**Parameters**

- **jws** (*str* or *bytes*) – JWS to get headers from
- **json\_decoder** – a function to decode JSON into a `dict`. Defaults to `json.loads`

**Returns** `headers`

**Return type** `list`

**Raises** `InvalidFormatError`: if not a valid JWS

## oneid.jwes

Provides useful functions for dealing with JWEs

Based on the [JSON Web Encryption \(JWE\)](#), and [JSON Web Algorithms \(JWA\)](#), IETF RFCs.

`oneid.jwes`.**make\_jwe** (*raw\_claims*, *sender\_keypair*, *recipient\_keypairs*, *jsonify=True*,  
*json\_encoder*=<function dumps>)

Convert claims into a JWE with General JWE JSON Serialization syntax

**Parameters**

- **raw\_claims** (*dict*) – payload data that will be converted to json
- **recipient\_keypairs** (*list* or *Keypair*) – *Keypairs* to encrypt the claims for
- **jsonify** (*bool*) – If True (default), return JSON, otherwise keep as dict
- **json\_encoder** (*function*) – encodes a `dict` into JSON. Defaults to `json.dumps`

**Returns** `JWE`

**Return type** `str` or `dict`

`oneid.jwes`.**decrypt\_jwe** (*jwe*, *recipient\_keypair*, *json\_decoder*=<function loads>)

Decrypt the claims in a JWE for a given recipient

**Parameters**

- **jwe** (*str*) – JWE to verify and convert
- **recipient\_keypair** (*Keypair*) – *Keypair* to use to decrypt.
- **json\_encoder** – a function to encode a `dict` into JSON. Defaults to `json.dumps`

**Returns** claims

**Return type** dict

**Raises** `InvalidFormatError`: if not a valid JWE

## oneid.nonces

Helpful utility functions

`oneid.nonces.make_nonce` (*expiry=None*)

Create a nonce with expiration timestamp included

**Parameters** **expiry** – a *datetime* that indicates when the nonce self-expires, defaults to now + 30 minutes

**Returns** nonce

`oneid.nonces.set_nonce_handlers` (*nonce\_verifier, nonce\_burner*)

Sets the functions to verify nonces and record their use.

By default, the nonces are saved in a local file named `~/oneid/used_nonces.txt` (or equivalent)

**Parameters**

- **nonce\_burner** – function to be called to verify. Passed one argument, the nonce
- **nonce\_verifier** – function to be called to burn. Passed one argument, the nonce

`oneid.nonces.verify_nonce` (*nonce, expiry=None*)

Ensure that the nonce is correct, and not from the future

Callers should also store used nonces and reject messages with previously-used ones.

**Parameters**

- **nonce** – Nonce as created with `make_nonce()`
- **expiry** – If not `None`, a *datetime* before which the nonce is not valid

**Returns** True only if nonce meets validation criteria

**Return type** bool

## oneid.auth

`class oneid.auth.OneIDAuthenticationService` (*api\_id=None, api\_key=None, server\_flag='u'*)

Encapsulates a connection to the oneID servers

**Parameters**

- **api\_id** – Your OneID API ID credentials (from <https://keychain.oneid.com/register>)
- **api\_key** – Your OneID API Key credentials (from <https://keychain.oneid.com/register>)
- **server\_flag** – If you want to connect to a different API should be (for example) “-test” when using a non-production server

**set\_credentials** (*api\_id=u'', api\_key=u''*)

Set the credentials used for access to the OneID Helper Service

**Parameters**

- **api\_id** – Your OneID API ID
- **api\_key** – Your OneID API key

**validate** (*oneid\_payload*)

Validate the data received by a callback

**Parameters** **oneid\_payload** – The dictionary you want to validate, typically the payload from a OneID sign in call

**Returns** if successful, *oneid\_payload*, updated with the response from oneID. Otherwise, the error response from oneID.

**success** (*oneid\_response*)

Check errorcode in a response

**Parameters** **oneid\_response** – A response from *validate()*

**Returns** True if the response indicates success, False otherwise.

## oneid.jose

Utility functions for dealing with JOSE objects generally.

Mostly used internally, but may be useful for external callers.

**oneid.jose.is\_compact\_jws** (*msg*)

Determine if a given message is a compact JWS (or JWT) or not. Does not necessarily mean that it is valid or authentic.

**Parameters** **msg** (*str*) – message to inspect

**Returns** True if the message is a compact JWS, False otherwise

**Return type** bool

**oneid.jose.is\_jws** (*msg, json\_decoder=<function loads>*)

Determine if a given message is a JWS or not (compact or otherwise). Does not necessarily mean that it is valid or authentic.

**Parameters**

- **msg** (*str or dict*) – message to inspect
- **json\_decoder** – a function to decode JSON into a dict. Defaults to *json.loads*

**Returns** True if the message is a JWS, False otherwise

**Return type** bool

**oneid.jose.is\_jwe** (*msg, json\_decoder=<function loads>*)

Determine if a given message is a JWE or not. Does not necessarily mean that it is valid or authentic.

**Parameters**

- **msg** (*str or dict*) – message to inspect
- **json\_decoder** – a function to decode JSON into a dict. Defaults to *json.loads*

**Returns** True if the message is a JWS, False otherwise

**Return type** bool

`oneid.jose.get_jwe_shared_header(jwe, json_decoder=<function loads>)`

Extract shared (non-encrypted) header values from a JWE

**Parameters**

- **jwe** (*str or dict*) – JWE to extract header field values from
- **json\_decoder** – a function to decode JSON into a `dict`. Defaults to `json.loads`

**Returns** header fields and values

**Return type** `dict`

**Raises** `InvalidFormatError`: if not a valid JWE

`oneid.jose.normalize_claims(raw_claims, issuer=None)`

Return a set of claims based on the provided claim set that includes reasonable defaults for required claims.

Note that the claims may be in the form of a valid JWE, in which case the inner values may be inspected.

**Parameters**

- **raw\_claims** (*dict*) – Initial set of claims, may or may not include required claims
- **issuer** – (optional) identifier of the identity creating the message

**Returns** filled-out claims

**Return type** `dict`

`oneid.jose.as_dict(msg, json_decoder=<function loads>)`

Unpack a message (if necessary) into its dictionary form.

**Parameters**

- **msg** (*str or dict*) – message to convert
- **json\_decoder** – a function to decode JSON into a `dict`. Defaults to `json.loads`

**Returns** the message in dictionary form

**Return type** `dict`

## oneid.utils

Helpful utility functions

`oneid.utils.base64url_encode(msg)`

Default `b64_encode` adds padding, `jwt` spec removes padding :param input: :type input: string or bytes :return: base64 en :rtype: bytes

`oneid.utils.base64url_decode(msg)`

`JWT` spec doesn't allow padding characters. `base64url_encode` removes them, `base64url_decode`, adds them back in before trying to base64 decode the message

**Parameters** **msg** (*string or bytes*) – URL safe base64 message

**Returns** decoded data

**Return type** bytes

## oneid.exceptions

**exception** `oneid.exceptions.InvalidAuthentication`

**exception** `oneid.exceptions.InvalidFormatError`

**exception** `oneid.exceptions.InvalidAlgorithmError` (*message="invalid 'alg' specified"*)

**exception** `oneid.exceptions.InvalidClaimsError`

**exception** `oneid.exceptions.InvalidKeyError`

**exception** `oneid.exceptions.KeySignatureMismatch`

**exception** `oneid.exceptions.InvalidSignatureError`

**exception** `oneid.exceptions.KeyHeaderMismatch`

**exception** `oneid.exceptions.ReservedHeader`

**exception** `oneid.exceptions.InvalidSignatureIndexes`

**exception** `oneid.exceptions.IdentityRequired`

**exception** `oneid.exceptions.InvalidRecipient`

**exception** `oneid.exceptions.DecryptionFailed`

## Contributing

### Contributing

If you have improvements to `Neustar-TDI/python-sdk`, send us your pull requests! `Neustar-TDI/python-sdk` is hosted on GitHub at <https://github.com/Neustar-TDI/python-sdk> and we welcome contributions of all forms. We use GitHub's [issue tracking](#) and collaboration tools exclusively for managing development.

### Getting Started

Working on `Neustar-TDI/python-sdk` requires additional packages, such as `pytest`, `mock` and `Sphinx`.

```
$ pip install -r dev_requirements
$ pip install -e .
$ pip install -r docs_requirements
```

You are now ready to run tests and build documentation.

### Running Tests

Unit tests are found in the `tests/` directory and are designed to use python's `unittest` library. `pytest` will discover the tests automatically.

```
$ pytest
```

Cross-version tests require installing additional Python versions, and setting up the local `tox` environment:



```
$ pyenv install 3.6.1
$ pyenv install 3.5.1
$ pyenv install 3.4.0
$ pyenv install 2.7.13
$ pyenv local 3.6.1 3.5.1 3.4.0 2.7.13
```

## Building Documentation

Documentation is stored in the `docs/` directory. It is written in `reStructuredText` and rendered using `Sphinx`.

```
$ make html
```

## Submitting Bugs

---

**Important:** Please report security issues only to [security@oneID.com](mailto:security@oneID.com). This is a private list only open to highly trusted cryptography developers.

---

- Check that someone hasn't already filed the bug by searching GitHub's [issue tracker](#).
- Don't use GitHub's issue tracker for support questions. Use [Stack Overflow](#) and tag `Neustar-TDI/python-sdk` for that.
- Provide information so we can replicate the bug on our end. (Python version, SDK version, OS, code samples, et cetera).

## Requesting Features

We're always trying to make `Neustar-TDI/python-sdk` better, and your feature requests are a key part of that.

- Check that someone hasn't already requested the feature by searching GitHub's [issue tracker](#).
- Clearly and concisely describe the feature and how you would like to see it implemented. Include example code if possible.
- Explain why you would like the feature. Sometimes it's very obvious, other times a use case will help us understand the importance of the requested feature.

## Contributing Changes

**Attention:** Always make a new branch for your work, no matter how small!

**Warning:** Don't submit unrelated changes in the same branch/pull request! We would hate to see an amazing pull request get rejected because it contains a bug in unrelated code.

- All new functions and classes **MUST** be documented and accompanied with unit tests.
- Our coding style follows [PEP 8](#).
- In docstrings we use `reStructuredText` as described in [PEP 287](#)

```
def foo(bar):  
    """  
    Makes input string better.  
  
    :param bar: input to make better  
    :return: a better input  
    """  
    ...
```

- Patches should be small to facilitate easier review.
- New features should branch off of `master` and once finished, submit a pull request into `develop`.
- `develop` branch is used to gather all new features for an upcoming release.
- Bug fixes should be based off the branch named after the oldest supported release the bug affects.
- If a feature was introduced in 1.1 and the latest release is 1.3, and a bug is found in that feature, make your branch based on 1.1. The maintainer will then forward-port it to 1.3 and `master`.
- You **MUST** have legal permission to distribute any code you contribute to `Neustar-TDI/python-sdk`.
- Class names which contains acronyms or initials should always be capitalized. i.e. `AESDecrypt` not `AesDecrypt`.

## CHAPTER 2

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`



### O

- `oneid.auth`, 17
- `oneid.exceptions`, 20
- `oneid.jose`, 18
- `oneid.jwes`, 16
- `oneid.jwt`s, 14
- `oneid.nonces`, 17
- `oneid.service`, 11
- `oneid.utils`, 19



**A**

add\_signature() (oneid.session.DeviceSession method), 12  
 AdminSession (class in oneid.session), 13  
 as\_dict() (in module oneid.jose), 19

**B**

base64url\_decode() (in module oneid.utils), 19  
 base64url\_encode() (in module oneid.utils), 19

**C**

create\_aes\_key() (in module oneid.service), 11  
 create\_secret\_key() (in module oneid.service), 11  
 Credentials (class in oneid.keychain), 9

**D**

decrypt() (oneid.keychain.ProjectCredentials method), 9  
 decrypt\_attr\_value() (in module oneid.service), 11  
 decrypt\_jwe() (in module oneid.jwes), 16  
 DecryptionFailed, 20  
 DeviceSession (class in oneid.session), 12

**E**

ecdh() (oneid.keychain.Keypair method), 10  
 encrypt() (oneid.keychain.ProjectCredentials method), 9  
 encrypt\_attr\_value() (in module oneid.service), 11  
 extend\_jws\_signatures() (in module oneid.jwts), 14

**F**

from\_jwk() (oneid.keychain.Keypair class method), 10  
 from\_public\_der() (oneid.keychain.Keypair class method), 10  
 from\_public\_pem() (oneid.keychain.Keypair class method), 9  
 from\_secret\_der() (oneid.keychain.Keypair class method), 10  
 from\_secret\_pem() (oneid.keychain.Keypair class method), 9

**G**

get\_jwe\_shared\_header() (in module oneid.jose), 19  
 get\_jws\_headers() (in module oneid.jwts), 16  
 get\_jws\_key\_ids() (in module oneid.jwts), 15

**I**

IdentityRequired, 20  
 InvalidAlgorithmError, 20  
 InvalidAuthentication, 20  
 InvalidClaimsError, 20  
 InvalidFormatError, 20  
 InvalidKeyError, 20  
 InvalidRecipient, 20  
 InvalidSignatureError, 20  
 InvalidSignatureIndexes, 20  
 is\_compact\_jws() (in module oneid.jose), 18  
 is\_jwe() (in module oneid.jose), 18  
 is\_jws() (in module oneid.jose), 18

**J**

jwk (oneid.keychain.Keypair attribute), 10  
 jwk\_private (oneid.keychain.Keypair attribute), 10  
 jwk\_public (oneid.keychain.Keypair attribute), 10

**K**

KeyHeaderMismatch, 20  
 Keypair (class in oneid.keychain), 9  
 KeySignatureMismatch, 20

**M**

make\_jwe() (in module oneid.jwes), 16  
 make\_jws() (in module oneid.jwts), 14  
 make\_jwt() (in module oneid.jwts), 14  
 make\_nonce() (in module oneid.nonces), 17

**N**

normalize\_claims() (in module oneid.jose), 19

## O

oneid.auth (module), 17  
oneid.exceptions (module), 20  
oneid.jose (module), 18  
oneid.jwes (module), 16  
oneid.jwt (module), 14  
oneid.nonces (module), 17  
oneid.service (module), 11  
oneid.utils (module), 19  
OneIDAuthenticationService (class in oneid.auth), 17

## P

prepare\_message() (oneid.session.DeviceSession method), 12  
prepare\_message() (oneid.session.ServerSession method), 13  
ProjectCredentials (class in oneid.keychain), 9  
public\_key (oneid.keychain.Keypair attribute), 11  
public\_key\_der (oneid.keychain.Keypair attribute), 11  
public\_key\_pem (oneid.keychain.Keypair attribute), 11

## R

remove\_jws\_signatures() (in module oneid.jwt), 15  
ReservedHeader, 20

## S

secret\_as\_der (oneid.keychain.Keypair attribute), 9  
secret\_as\_pem (oneid.keychain.Keypair attribute), 9  
ServerSession (class in oneid.session), 13  
SessionBase (class in oneid.session), 12  
set\_credentials() (oneid.auth.OneIDAuthenticationService method), 17  
set\_nonce\_handlers() (in module oneid.nonces), 17  
sign() (oneid.keychain.Keypair method), 10  
success() (oneid.auth.OneIDAuthenticationService method), 18

## V

validate() (oneid.auth.OneIDAuthenticationService method), 18  
verify() (oneid.keychain.Keypair method), 10  
verify\_jws() (in module oneid.jwt), 15  
verify\_jwt() (in module oneid.jwt), 14  
verify\_message() (oneid.session.DeviceSession method), 12  
verify\_message() (oneid.session.ServerSession method), 13  
verify\_nonce() (in module oneid.nonces), 17