ItsDangerous Documentation (1.1.x)

Release 1.1.0

Pallets

Oct 20, 2022

CONTENTS

Sometimes you just want to send some data to untrusted environments. But how to do this safely? The trick involves signing. Given a key only you know, you can cryptographically sign your data and hand it over to someone else. When you get the data back you can easily ensure that nobody tampered with it.

Granted, the receiver can decode the contents and look into the package, but they can not modify the contents unless they also have your secret key. So if you keep the key secret and complex, you will be fine.

Internally ItsDangerous uses HMAC and SHA-512 for signing by default. The initial implementation was inspired by Django's signing module. It also supports JSON Web Signatures (JWS). The library is BSD licensed.

CHAPTER

ONE

INSTALLING

Install and update using pip:

```
pip install -U itsdangerous
```

CHAPTER

TWO

EXAMPLE USE CASES

- You can serialize and sign a user ID in a URL and email it to them to unsubscribe from a newsletter. This way you don't need to generate one-time tokens and store them in the database. Same thing with any kind of activation link for accounts and similar things.
- Signed objects can be stored in cookies or other untrusted sources which means you don't need to have sessions stored on the server, which reduces the number of necessary database queries.
- Signed information can safely do a round trip between server and client in general which makes them useful for passing server-side state to a client and then back.

CHAPTER

THREE

TABLE OF CONTENTS

3.1 Signing Interface

The most basic interface is the signing interface. The *Signer* class can be used to attach a signature to a specific string:

```
from itsdangerous import Signer
s = Signer("secret-key")
s.sign("my string")
b'my string.wh6tMHxLgJqB6oY1uT73iMlyrOA'
```

The signature is appended to the string, separated by a dot. To validate the string, use the *unsign()* method:

```
s.unsign(b"my string.wh6tMHxLgJqB6oY1uT73iMlyrOA")
b'my string'
```

If unicode strings are provided, an implicit encoding to UTF-8 happens. However after unsigning you won't be able to tell if it was unicode or a bytestring.

If the value is changed, the signature will no longer match, and unsigning will raise a *BadSignature* exception:

```
s.unsign(b"different string.wh6tMHxLgJqB6oY1uT73iMlyrOA")
Traceback (most recent call last):
  ...
itsdangerous.exc.BadSignature: Signature "wh6tMHxLgJqB6oY1uT73iMlyrOA" does not match
```

To record and validate the age of a signature, see *Signing With Timestamps*.

class itsdangerous.signer.**Signer**(*secret_key*, *salt=None*, *sep='.'*, *key_derivation=None*, *digest_method=None*, *algorithm=None*)

This class can sign and unsign bytes, validating the signature provided.

Salt can be used to namespace the hash, so that a signed string is only valid for a given namespace. Leaving this at the default value or re-using a salt value across different parts of your application where the same signed value in one part can mean something different in another part is a security risk.

See *The Salt* for an example of what the salt is doing and how you can utilize it.

New in version 0.18: `algorithm` was added as an argument to the class constructor.

New in version 0.14: `key_derivation` and `digest_method` were added as arguments to the class constructor.

static default_digest_method()

The digest method to use for the signer. This defaults to SHA1 but can be changed to any other function in the hashlib module.

New in version 0.14.

default_key_derivation = 'django-concat'
Controls how the key is derived. The default is Django-style concatenation. Possible values are `concat`, `django-concat` and `hmac`. This is used for deriving a key from the secret key with an added salt.

New in version 0.14.

derive_key()
This method is called to derive the key. The default key derivation choices can be overridden here. Key derivation is not intended to be used as a security method to make a complex key out of a short password. Instead you should use large random secret keys.

get_signature(*value*)
Returns the signature for the given value.

sign(*value*)
Signs the given string.

unsign(*signed_value*)
Unsigns the given string.

validate(*signed_value*)
Only validates the given signed value. Returns `True` if the signature exists and is valid.

verify_signature(*value*, *sig*)
Verifies the signature for the given value.

3.1.1 Signing Algorithms

class itsdangerous.signer.**NoneAlgorithm**
Provides an algorithm that does not perform any signing and returns an empty signature.

class itsdangerous.signer.**HMACAlgorithm**(*digest_method=None*)
Provides signature generation using HMACs.

3.2 Serialization Interface

The *Signing Interface* only signs strings. To sign other types, the *Serializer* class provides a `dumps`/`loads` interface similar to Python's `json` module, which serializes the object to a string then signs that.

Use *dumps()* to serialize and sign the data:

```
from itsdangerous.serializer import Serializer
s = Serializer("secret-key")
s.dumps([1, 2, 3, 4])
b'[1, 2, 3, 4].r7R9RhGgDPvvWl3iNzLuIIfELmo'
```

Use *loads()* to verify the signature and deserialize the data.

```
s.loads('[1, 2, 3, 4].r7R9RhGgDPvvWl3iNzLuIIfELmo')
[1, 2, 3, 4]
```

By default, data is serialized to JSON. If simplejson is installed, it is preferred over the built-in `json` module. This internal serializer can be changed by subclassing.

To record and validate the age of the signature, see *Signing With Timestamps*. To serialize to a format that is safe to use in URLs, see *URL Safe Serialization*.

3.2.1 The Salt

All classes also accept a salt argument. The name might be misleading because usually if you think of salts in cryptography you would expect the salt to be something that is stored alongside the resulting signed string as a way to prevent rainbow table lookups. Such salts are usually public.

In ItsDangerous, like in the original Django implementation, the salt serves a different purpose. You could describe it as namespacing. It's still not critical if you disclose it because without the secret key it does not help an attacker.

Let's assume that you have two links you want to sign. You have the activation link on your system which can activate a user account and you have an upgrade link that can upgrade a user's account to a paid account which you send out via email. If in both cases all you sign is the user ID a user could reuse the variable part in the URL from the activation link to upgrade the account. Now you could either put more information in there which you sign (like the intention: upgrade or activate), but you could also use different salts:

```
from itsdangerous.url_safe import URLSafeSerializer
s1 = URLSafeSerializer("secret-key", salt="activate")
s1.dumps(42)
'NDI.MHQqszw6Wc81wOBQszCrEE_RlzY'
s2 = URLSafeSerializer("secret-key", salt="upgrade")
s2.dumps(42)
'NDI.c0MpsD6gzpilOAeUPra3NShPXsE'
```

The second serializer can't load data dumped with the first because the salts differ:

```
s2.loads(s1.dumps(42))
Traceback (most recent call last):
  ...
itsdangerous.exc.BadSignature: Signature "MHQqszw6Wc81wOBQszCrEE_RlzY" does not match
```

Only the serializer with the same salt can load the data:

```
s2.loads(s2.dumps(42))
42
```

3.2.2 Responding to Failure

Exceptions have helpful attributes which allow you to inspect the payload if the signature check failed. This has to be done with extra care because at that point you know that someone tampered with your data but it might be useful for debugging purposes.

```
from itsdangerous.serializer import Serializer
from itsdangerous.exc import BadSignature, BadData

s = URLSafeSerializer("secret-key")
decoded_payload = None

try:
    decoded_payload = s.loads(data)
    # This payload is decoded and safe
except BadSignature as e:
    if e.payload is not None:
        try:
            decoded_payload = s.load_payload(e.payload)
        except BadData:
            pass
```

(continues on next page)

(continued from previous page)

```
        # This payload is decoded but unsafe because someone
        # tampered with the signature. The decode (load_payload)
        # step is explicit because it might be unsafe to unserialize
        # the payload (think pickle instead of json!)
```

If you don't want to inspect attributes to figure out what exactly went wrong you can also use *loads_unsafe()*:

```
sig_okay, payload = s.loads_unsafe(data)
```

The first item in the returned tuple is a boolean that indicates if the signature was correct.

3.2.3 API

class itsdangerous.serializer.**Serializer**(*secret_key*, *salt=b'itsdangerous'*, *serializer=None*, *serializer_kwargs=None*, *signer=None*, *signer_kwargs=None*, *fallback_signers=None*)

This class provides a serialization interface on top of the signer. It provides a similar API to json/pickle and other modules but is structured differently internally. If you want to change the underlying implementation for parsing and loading you have to override the *load_payload()* and *dump_payload()* functions.

This implementation uses simplejson if available for dumping and loading and will fall back to the standard library's json module if it's not available.

You do not need to subclass this class in order to switch out or customize the *Signer*. You can instead pass a different class to the constructor as well as keyword arguments as a dict that should be forwarded.

```
s = Serializer(signer_kwargs={'key_derivation': 'hmac'})
```

You may want to upgrade the signing parameters without invalidating existing signatures that are in use. Fallback signatures can be given that will be tried if unsigning with the current signer fails.

Fallback signers can be defined by providing a list of `fallback_signers`. Each item can be one of the following: a signer class (which is instantiated with `signer_kwargs`, `salt`, and `secret_key`), a tuple `(signer_class, signer_kwargs)`, or a dict of `signer_kwargs`.

For example, this is a serializer that signs using SHA-512, but will unsign using either SHA-512 or SHA1:

```
s = Serializer(
    signer_kwargs={"digest_method": hashlib.sha512},
    fallback_signers=[{"digest_method": hashlib.sha1}]
)
```

Changed in version 1.1.0:: Added support for `fallback_signers` and configured a default SHA-512 fallback. This fallback is for users who used the yanked 1.0.0 release which defaulted to SHA-512.

Changed in version 0.14:: The `signer` and `signer_kwargs` parameters were added to the constructor.

default_fallback_signers = [{'digest_method': <built-in function openssl_sha512>}]
The default fallback signers.

default_serializer = <module 'json' from '/home/docs/.pyenv/versions/3.7.9/lib/python3
If a serializer module or class is not passed to the constructor this one is picked up. This currently defaults to `json`.

default_signer
alias of *itsdangerous.signer.Signer*

dump(*obj*, *f*, *salt=None*)
Like *dumps()* but dumps into a file. The file handle has to be compatible with what the internal serializer expects.

dump_payload(*obj*)
Dumps the encoded object. The return value is always bytes. If the internal serializer returns text, the value will be encoded as UTF-8.

dumps(*obj*, *salt=None*)
Returns a signed string serialized with the internal serializer. The return value can be either a byte or unicode string depending on the format of the internal serializer.

iter_unsigners(*salt=None*)
Iterates over all signers to be tried for unsigning. Starts with the configured signer, then constructs each signer specified in `fallback_signers`.

load(*f*, *salt=None*)
Like *loads()* but loads from a file.

load_payload(*payload*, *serializer=None*)
Loads the encoded object. This function raises *BadPayload* if the payload is not valid. The `serializer` parameter can be used to override the serializer stored on the class. The encoded `payload` should always be bytes.

load_unsafe(*f*, **args*, ***kwargs*)
Like *loads_unsafe()* but loads from a file.

New in version 0.15.

loads(*s*, *salt=None*)
Reverse of *dumps()*. Raises *BadSignature* if the signature validation fails.

loads_unsafe(*s*, *salt=None*)
Like *loads()* but without verifying the signature. This is potentially very dangerous to use depending on how your serializer works. The return value is `(signature_valid, payload)` instead of just the payload. The first item will be a boolean that indicates if the signature is valid. This function never fails.

Use it for debugging only and if you know that your serializer module is not exploitable (for example, do not use it with a pickle serializer).

New in version 0.15.

make_signer(*salt=None*)
Creates a new instance of the signer to be used. The default implementation uses the *Signer* base class.

3.3 Exceptions

exception itsdangerous.exc.**BadData**(*message*)
Raised if bad data of any sort was encountered. This is the base for all exceptions that ItsDangerous defines.

New in version 0.15.

exception itsdangerous.exc.**BadSignature**(*message*, *payload=None*)
Raised if a signature does not match.

payload = None
The payload that failed the signature test. In some situations you might still want to inspect this, even if you know it was tampered with.

New in version 0.14.

exception itsdangerous.exc.**BadTimeSignature**(*message*, *payload=None*, *date_signed=None*)

Raised if a time-based signature is invalid. This is a subclass of *BadSignature*.

date_signed = None

If the signature expired this exposes the date of when the signature was created. This can be helpful in order to tell the user how long a link has been gone stale.

New in version 0.14.

exception itsdangerous.exc.**SignatureExpired**(*message*, *payload=None*, *date_signed=None*)

Raised if a signature timestamp is older than `max_age`. This is a subclass of *BadTimeSignature*.

exception itsdangerous.exc.**BadHeader**(*message*, *payload=None*, *header=None*, *original_error=None*)

Raised if a signed header is invalid in some form. This only happens for serializers that have a header that goes with the signature.

New in version 0.24.

header = None

If the header is actually available but just malformed it might be stored here.

original_error = None

If available, the error that indicates why the payload was not valid. This might be `None`.

exception itsdangerous.exc.**BadPayload**(*message*, *original_error=None*)

Raised if a payload is invalid. This could happen if the payload is loaded despite an invalid signature, or if there is a mismatch between the serializer and deserializer. The original exception that occurred during loading is stored on as *original_error*.

New in version 0.15.

original_error = None

If available, the error that indicates why the payload was not valid. This might be `None`.

3.4 Signing With Timestamps

If you want to expire signatures you can use the *TimestampSigner* class which will adds timestamp information and signs it. On unsigning you can validate that the timestamp did not expire:

```
from itsdangerous import TimestampSigner
s = TimestampSigner('secret-key')
string = s.sign('foo')
```

```
s.unsign(string, max_age=5)
Traceback (most recent call last):
  ...
itsdangerous.exc.SignatureExpired: Signature age 15 > 5 seconds
```

class itsdangerous.timed.**TimestampSigner**(*secret_key*, *salt=None*, *sep='.'*, *key_derivation=None*, *digest_method=None*, *algorithm=None*)

Works like the regular *Signer* but also records the time of the signing and can be used to expire signatures. The *unsign()* method can raise *SignatureExpired* if the unsigning failed because the signature is expired.

get_timestamp()
Returns the current timestamp. The function must return an integer.

sign(*value*)
Signs the given string and also attaches time information.

timestamp_to_datetime(*ts*)
Used to convert the timestamp from `get_timestamp()` into a datetime object.

unsign(*value*, *max_age=None*, *return_timestamp=False*)
Works like the regular `Signer.unsign()` but can also validate the time. See the base docstring of the class for the general behavior. If `return_timestamp` is `True` the timestamp of the signature will be returned as a naive `datetime.datetime` object in UTC.

validate(*signed_value*, *max_age=None*)
Only validates the given signed value. Returns `True` if the signature exists and is valid.

class itsdangerous.timed.**TimedSerializer**(*secret_key*, *salt=b'itsdangerous'*, *serializer=None*, *serializer_kwargs=None*, *signer=None*, *signer_kwargs=None*, *fallback_signers=None*)
Uses `TimestampSigner` instead of the default `Signer`.

default_signer
alias of `TimestampSigner`

loads(*s*, *max_age=None*, *return_timestamp=False*, *salt=None*)
Reverse of `dumps()`, raises `BadSignature` if the signature validation fails. If a `max_age` is provided it will ensure the signature is not older than that time in seconds. In case the signature is outdated, `SignatureExpired` is raised. All arguments are forwarded to the signer's `unsign()` method.

loads_unsafe(*s*, *max_age=None*, *salt=None*)
Like `loads()` but without verifying the signature. This is potentially very dangerous to use depending on how your serializer works. The return value is `(signature_valid, payload)` instead of just the payload. The first item will be a boolean that indicates if the signature is valid. This function never fails.

Use it for debugging only and if you know that your serializer module is not exploitable (for example, do not use it with a pickle serializer).

New in version 0.15.

3.5 URL Safe Serialization

Often it is helpful if you can pass these trusted strings in places where you only have a limited set of characters available. Because of this, ItsDangerous also provides URL safe serializers:

```
from itsdangerous.url_safe import URLSafeSerializer
s = URLSafeSerializer("secret-key")
s.dumps([1, 2, 3, 4])
'WzEsMiwzLDRd.wSPHqC0gR7VUqivlSukJ0IeTDgo'
s.loads("WzEsMiwzLDRd.wSPHqC0gR7VUqivlSukJ0IeTDgo")
[1, 2, 3, 4]
```

class itsdangerous.url_safe.**URLSafeSerializer**(*secret_key*, *salt=b'itsdangerous'*, *serializer=None*, *serializer_kwargs=None*, *signer=None*, *signer_kwargs=None*, *fallback_signers=None*)

Works like *Serializer* but dumps and loads into a URL safe string consisting of the upper and lowercase character of the alphabet as well as `'_'`, `'-'` and `'.'`.

class itsdangerous.url_safe.**URLSafeTimedSerializer**(*secret_key*, *salt=b'itsdangerous'*, *serializer=None*, *serializer_kwargs=None*, *signer=None*, *signer_kwargs=None*, *fallback_signers=None*)

Works like *TimedSerializer* but dumps and loads into a URL safe string consisting of the upper and lowercase character of the alphabet as well as `'_'`, `'-'` and `'.'`.

3.6 JSON Web Signature (JWS)

JSON Web Signatures (JWS) work similarly to the existing URL safe serializer but will emit headers according to draft-ietf-jose-json-web -signature.

```
from itsdangerous import JSONWebSignatureSerializer
s = JSONWebSignatureSerializer("secret-key")
s.dumps({"x": 42})
'eyJhbGciOiJIUzI1NiJ9.eyJ4Ijo0Mn0.ZdTn1YyGz9Yx5B5wNpWRL221G1WpVE5fPCPKNuc6UAo'
```

When loading the value back the header will not be returned by default like with the other serializers. However it is possible to also ask for the header by passing `return_header=True`. Custom header fields can be provided upon serialization:

```
s.dumps(0, header_fields={"v": 1})
'eyJhbGciOiJIUzI1NiIsInYiOjF9.MA.wT-RZI9YU06R919VBdAfTLn82_iIQD70J_j-3F4z_aM'
s.loads(
    "eyJhbGciOiJIUzI1NiIsInYiOjF9"
    ".MA.wT-RZI9YU06R919VBdAfTLn82_iIQD70J_j-3F4z_aM"
)
(0, {'alg': 'HS256', 'v': 1})
```

ItsDangerous only provides HMAC SHA derivatives and the none algorithm at the moment and does not support the ECC based ones. The algorithm in the header is checked against the one of the serializer and on a mismatch a *BadSignature* exception is raised.

class itsdangerous.jws.**JSONWebSignatureSerializer**(*secret_key*, *salt=None*, *serializer=None*, *serializer_kwargs=None*, *signer=None*, *signer_kwargs=None*, *algorithm_name=None*)

This serializer implements JSON Web Signature (JWS) support. Only supports the JWS Compact Serialization.

default_algorithm = 'HS512'

The default algorithm to use for signature generation

default_serializer

alias of `itsdangerous._json._CompactJSON`

dump_payload(*header*, *obj*)

Dumps the encoded object. The return value is always bytes. If the internal serializer returns text, the value will be encoded as UTF-8.

dumps(*obj*, *salt=None*, *header_fields=None*)

Like *Serializer.dumps()* but creates a JSON Web Signature. It also allows for specifying additional

fields to be included in the JWS header.

load_payload(*payload*, *serializer=None*, *return_header=False*)
Loads the encoded object. This function raises *BadPayload* if the payload is not valid. The `serializer` parameter can be used to override the serializer stored on the class. The encoded `payload` should always be bytes.

loads(*s*, *salt=None*, *return_header=False*)
Reverse of *dumps()*. If requested via `return_header` it will return a tuple of payload and header.

loads_unsafe(*s*, *salt=None*, *return_header=False*)
Like *loads()* but without verifying the signature. This is potentially very dangerous to use depending on how your serializer works. The return value is `(signature_valid, payload)` instead of just the payload. The first item will be a boolean that indicates if the signature is valid. This function never fails.

Use it for debugging only and if you know that your serializer module is not exploitable (for example, do not use it with a pickle serializer).

New in version 0.15.

make_signer(*salt=None*, *algorithm=None*)
Creates a new instance of the signer to be used. The default implementation uses the *Signer* base class.

class itsdangerous.jws.**TimedJSONWebSignatureSerializer**(*secret_key*, *expires_in=None*, ***kwargs*)
Works like the regular *JSONWebSignatureSerializer* but also records the time of the signing and can be used to expire signatures.

JWS currently does not specify this behavior but it mentions a possible extension like this in the spec. Expiry date is encoded into the header similar to what's specified in draft-ietf-oauth -json-web-token.

loads(*s*, *salt=None*, *return_header=False*)
Reverse of `dumps()`. If requested via `return_header` it will return a tuple of payload and header.

3.7 Encoding Utilities

itsdangerous.encoding.**base64_encode**(*string*)
Base64 encode a string of bytes or text. The resulting bytes are safe to use in URLs.

itsdangerous.encoding.**base64_decode**(*string*)
Base64 decode a URL-safe string of bytes or text. The result is bytes.

3.8 License

- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

We kindly ask you to use these themes in an unmodified manner only with Pallets and Pallets-related projects, not for unrelated projects. If you like the visual style and want to use it for your own projects, please consider making some larger changes to the themes (such as changing font faces, sizes, colors or margins).

THIS SOFTWARE AND DOCUMENTATION IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE AND DOCUMENTATION, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The initial implementation of itsdangerous was inspired by Django's signing module.

3.9 Changes

3.9.1 Version 1.1.0

Released 2018-10-26

- Change default signing algorithm back to SHA-1. #113
- Added a default SHA-512 fallback for users who used the yanked 1.0.0 release which defaulted to SHA-512. #114
- Add support for fallback algorithms during deserialization to support changing the default in the future without breaking existing signatures. #113
- Changed capitalization of packages back to lowercase as the change in capitalization broke some tooling. #113

3.9.2 Version 1.0.0

Released 2018-10-18

YANKED

Note: This release was yanked from PyPI because it changed the default algorithm to SHA-512. This decision was reverted in 1.1.0 and it remains at SHA1.

- Drop support for Python 2.6 and 3.3.
- Refactor code from a single module to a package. Any object in the API docs is still importable from the top-level `itsdangerous` name, but other imports will need to be changed. A future release will remove many of these compatibility imports. #107
- Optimize how timestamps are serialized and deserialized. #13

- `base64_decode` raises `BadData` when it is passed invalid data. #27
- Ensure value is bytes when signing to avoid a `TypeError` on Python 3. #29
- Add a `serializer_kwargs` argument to `Serializer`, which is passed to `dumps` during `dump_payload`. #36
- More compact JSON dumps for unicode strings. #38
- Use the full timestamp rather than an offset, allowing dates before 2011. #46

  To retain compatibility with signers from previous versions, consider using this shim when unsigning.
- Detect a `sep` character that may show up in the signature itself and raise a `ValueError`. #62
- Use a consistent signature for keyword arguments for `Serializer.load_payload` in subclasses. #74, #75
- Change default intermediate hash from SHA-1 to SHA-512. #80
- Convert JWS exp header to an int when loading. #99

3.9.3 Version 0.24

Released 2014-03-28

- Added a `BadHeader` exception that is used for bad headers that replaces the old `BadPayload` exception that was reused in those cases.

3.9.4 Version 0.23

Released 2013-08-08

- Fixed a packaging mistake that caused the tests and license files to not be included.

3.9.5 Version 0.22

Released 2013-07-03

- Added support for `TimedJSONWebSignatureSerializer`.
- Made it possible to override the signature verification function to allow implementing asymmetrical algorithms.

3.9.6 Version 0.21

Released 2013-05-26

- Fixed an issue on Python 3 which caused invalid errors to be generated.

3.9.7 Version 0.20

Released 2013-05-23

- Fixed an incorrect call into `want_bytes` that broke some uses of ItsDangerous on Python 2.6.

3.9.8 Version 0.19

Released 2013-05-21

- Dropped support for 2.5 and added support for 3.3.

3.9.9 Version 0.18

Released 2013-05-03

- Added support for JSON Web Signatures (JWS).

3.9.10 Version 0.17

Released 2012-08-10

- Fixed a name error when overriding the digest method.

3.9.11 Version 0.16

Released 2012-07-11

- Made it possible to pass unicode values to `load_payload` to make it easier to debug certain things.

3.9.12 Version 0.15

Released 2012-07-11

- Made standalone `load_payload` more robust by raising one specific error if something goes wrong.
- Refactored exceptions to catch more cases individually, added more attributes.
- Fixed an issue that caused `load_payload` not work in some situations with timestamp based serializers
- Added an `loads_unsafe` method.

3.9.13 Version 0.14

Released 2012-06-29

- API refactoring to support different key derivations.
- Added attributes to exceptions so that you can inspect the data even if the signature check failed.

3.9.14 Version 0.13

Released 2012-06-10

- Small API change that enables customization of the digest module.

3.9.15 Version 0.12

Released 2012-02-22

- Fixed a problem with the local timezone being used for the epoch calculation. This might invalidate some of your signatures if you were not running in UTC timezone. You can revert to the old behavior by monkey patching `itsdangerous.EPOCH`.

3.9.16 Version 0.11

Released 2011-07-07

- Fixed an uncaught value error.

3.9.17 Version 0.10

Released 2011-06-25

- Refactored interface that the underlying serializers can be swapped by passing in a module instead of having to override the payload loaders and dumpers. This makes the interface more compatible with Django's recent changes.

PYTHON MODULE INDEX

i

INDEX

M

N

O

P

S

T

U

V