Cryptographic signing

The golden rule of web application security is to never trust data from untrusted sources. Sometimes it can be useful to pass data through an untrusted medium. Cryptographically signed values can be passed through an untrusted channel safe in the knowledge that any tampering will be detected.

Django provides both a low-level API for signing values and a high-level API for setting and reading signed cookies, one of the most common uses of signing in web applications.

You may also find signing useful for the following:

- Generating "recover my account" URLs for sending to users who have lost their password.
- Ensuring data stored in hidden form fields has not been tampered with.
- Generating one-time secret URLs for allowing temporary access to a protected resource, for example a downloadable file that a user has paid for.

Protecting SECRET_KEY and SECRET_KEY_FALLBACKS

When you create a new Django project using `startproject`, the `settings.py` file is generated automatically and gets a random `SECRET_KEY` value. This value is the key to securing signed data – it is vital you keep this secure, or attackers could use it to generate their own signed values.

`SECRET_KEY_FALLBACKS` can be used to rotate secret keys. The values will not be used to sign data, but if specified, they will be used to validate signed data and must be kept secure.

Using the low-level API

Django's signing methods live in the `django.core.signing` module. To sign a value, first instantiate a `Signer` instance:

```
>>> from django.core.signing import Signer
>>> signer = Signer()
>>> value = signer.sign("My string")
>>> value
'My string:GdMGD6HNQ_qdgxYP8yBZAdAIV1w'
```

The signature is appended to the end of the string, following the colon. You can retrieve the original value using the `unsign` method:

```
>>> original = signer.unsign(value)
>>> original
'My string'
```

If you pass a non-string value to `sign`, the value will be forced to string before being signed, and the `unsign` result will give you that string value:

```
>>> signed = signer.sign(2.5)
>>> original = signer.unsign(signed)
>>> original
'2.5'
```

If you wish to protect a list, tuple, or dictionary you can do so using the `sign_object()` and `unsign_object()` methods:

```
>>> signed_obj = signer.sign_object({"message": "Hello!"})
>>> signed_obj
'eyJtZXNzYWdlIjoiSGVsbG8hIn0:Xdc-mOFDjs22KsQAqfVfi8PQSPdo3ckWJxPWwQOFhR4'
>>> obj = signer.unsign_object(signed_obj)
>>> obj
{'message': 'Hello!'}
```

See Protecting complex data structures for more details.

If the signature or value have been altered in any way, a `django.core.signing.BadSignature` exception will be raised:

```
>>> from django.core import signing
>>> value += "m"
>>> try:
...     original = signer.unsign(value)
... except signing.BadSignature:
...     print("Tampering detected!")
...
```

By default, the `Signer` class uses the `SECRET_KEY` setting to generate signatures. You can use a different secret by passing it to the `Signer` constructor:

```
>>> signer = Signer(key="my-other-secret")
>>> value = signer.sign("My string")
>>> value
'My string:EkfQJafvGyiofrdGnuthdxImIJw'
```

`class Signer(*, key=None, sep=':', salt=None, algorithm=None, fallback_keys=None)` [source]

Returns a signer which uses `key` to generate signatures and `sep` to separate values. `sep` cannot be in the **URL safe base64 alphabet**. This alphabet contains alphanumeric characters, hyphens, and underscores. `algorithm` must be an algorithm supported by `hashlib`, it defaults to `'sha256'`. `fallback_keys` is a list of additional values used to validate signed data, defaults to `SECRET_KEY_FALLBACKS`.

Using the `salt` argument

If you do not wish for every occurrence of a particular string to have the same signature hash, you can use the optional `salt` argument to the `Signer` class. Using a salt will seed the signing hash function with both the salt and your `SECRET_KEY`:

```
>>> signer = Signer()
>>> signer.sign("My string")
'My string:GdMGD6HNQ_qdgxYP8yBZAdAIV1w'
>>> signer.sign_object({"message": "Hello!"})
'eyJtZXNzYWdlIjoiSGVsbG8hIn0:Xdc-mOFDjs22KsQAqfVfi8PQSPdo3ckWJxPWwQOFhR4'
>>> signer = Signer(salt="extra")
>>> signer.sign("My string")
'My string:Ee7vGi-ING6n02gkcJ-QLHg6vFw'
>>> signer.unsign("My string:Ee7vGi-ING6n02gkcJ-QLHg6vFw")
'My string'
>>> signer.sign_object({"message": "Hello!"})
'eyJtZXNzYWdlIjoiSGVsbG8hIn0:-UWSLCE-oUAHzhkHviYz3SOZYBjFKllEOyVZNuUtM-I'
>>> signer.unsign_object(
...     "eyJtZXNzYWdlIjoiSGVsbG8hIn0:-UWSLCE-oUAHzhkHviYz3SOZYBjFKllEOyVZNuUtM-I"
... )
{'message': 'Hello!'}
```

Using salt in this way puts the different signatures into different namespaces. A signature that comes from one namespace (a particular salt value) cannot be used to validate the same plaintext string in a different namespace that is using a different salt setting. The result is to prevent an attacker from using a signed string generated in one place in the code as input to another piece of code that is generating (and verifying) signatures using a different salt.

Unlike your `SECRET_KEY`, your salt argument does not need to stay secret.

Verifying timestamped values

`TimestampSigner` is a subclass of `Signer` that appends a signed timestamp to the value. This allows you to confirm that a signed value was created within a specified period of time:

```
>>> from datetime import timedelta
>>> from django.core.signing import TimestampSigner
>>> signer = TimestampSigner()
>>> value = signer.sign("hello")
>>> value
'hello:1NMg5H:oPVuCqlJWmChm1rA2lyTUtelC-c'
>>> signer.unsign(value)
'hello'
>>> signer.unsign(value, max_age=10)
SignatureExpired: Signature age 15.5289158821 > 10 seconds
>>> signer.unsign(value, max_age=20)
'hello'
>>> signer.unsign(value, max_age=timedelta(seconds=20))
'hello'
```

`class TimestampSigner(*, key=None, sep=':', salt=None, algorithm='sha256')` [source]

`sign(value)` [source]

Sign `value` and append current timestamp to it.

`unsign(value, max_age=None)` [source]

Checks if `value` was signed less than `max_age` seconds ago, otherwise raises `SignatureExpired`. The `max_age` parameter can accept an integer or a `datetime.timedelta` object.

```
sign_object(obj, serializer=JSONSerializer, compress=False)
```

Encode, optionally compress, append current timestamp, and sign complex data structure (e.g. list, tuple, or dictionary).

```
unsign_object(signed_obj, serializer=JSONSerializer, max_age=None)
```

Checks if `signed_obj` was signed less than `max_age` seconds ago, otherwise raises `SignatureExpired`. The `max_age` parameter can accept an integer or a `datetime.timedelta` object.

Protecting complex data structures

If you wish to protect a list, tuple or dictionary you can do so using the `Signer.sign_object()` and `unsign_object()` methods, or signing module's `dumps()` or `loads()` functions (which are shortcuts for `TimestampSigner(salt='django.core.signing').sign_object()/unsign_object()`). These use JSON serialization under the hood. JSON ensures that even if your `SECRET_KEY` is stolen an attacker will not be able to execute arbitrary commands by exploiting the pickle format:

```
>>> from django.core import signing
>>> signer = signing.TimestampSigner()
>>> value = signer.sign_object({"foo": "bar"})
>>> value
'eyJmb28iOiJiYXIifQ:1kx6R3:D4qGKiptAqo5QW9iv4eNLc6xl4RwiFfes6oOcYhkYnc'
>>> signer.unsign_object(value)
{'foo': 'bar'}
>>> value = signing.dumps({"foo": "bar"})
>>> value
'eyJmb28iOiJiYXIifQ:1kx6Rf:LBB39RQmME-SRvilheUe5EmPYRbuDBgQp2tCAi7KGLk'
>>> signing.loads(value)
{'foo': 'bar'}
```

Because of the nature of JSON (there is no native distinction between lists and tuples) if you pass in a tuple, you will get a list from `signing.loads(object)`:

```
>>> from django.core import signing
>>> value = signing.dumps(("a", "b", "c"))
>>> signing.loads(value)
['a', 'b', 'c']
```

```
dumps(obj, key=None, salt='django.core.signing', serializer=JSONSerializer, compress=False)  [source]
```

Returns URL-safe, signed base64 compressed JSON string. Serialized object is signed using `TimestampSigner`.

```
loads(string, key=None, salt='django.core.signing', serializer=JSONSerializer, max_age=None, fallback_keys=None)  [source]
```

Reverse of `dumps()`, raises `BadSignature` if signature fails. Checks `max_age` (in seconds) if given.