Authorization token

Category: Crypto
Difficulty: Easy
Author: Aardvarksoep

Description

> Somehow you ended up at the website of the "Vexillophiles Society", which seems to use some FIPS-certified quantum-resistent military-grade cryptography to protect their users' sessions.
>
> Even so, when you look closely you may find a big mistake, and you don't need any math to exploit it.

Solution

We get a link to a website containing a sign up page, which also tells us in the bottom right, that it is "Secured with open source software" together with a link where we can download a python script containing the authentication logic:

```
from base64 import b64encode, b64decode
from binascii import hexlify, unhexlify
from urllib.parse import unquote
import json, hmac, os

_TOKEN_FIELDS = ['email', 'fullname', 'role', 'signature']
_AUTH_COOKIE = 'auth'

def get_user_name(request):
        """Fetch the full name of the currently logged in user."""
        return _get_user_token(request)['fullname']

def start_session(response, email, fullname, role='Guest'):
        """Add cookie to response to start an authorized session with given user info."""

        token = {
        'email': email,
        'fullname': fullname,
        'role': role
        }

        token['signature'] = _sign_token(token)
        response.set_cookie(_AUTH_COOKIE, b64encode(json.dumps(token).encode()), httponly=True)

def admin_check(request):
        """This will throw an exception when the current user is not an admin.
        Recommended to put at the start of your /admin page implementation."""

        token = _get_user_token(request)

        if not token or token['role'] != 'Admin':
                raise Exception('Authorization error: user has no admin privileges.')

def is_logged_in(request):
        return _get_user_token(request) != None

def logout(response):
        response.set_cookie(_AUTH_COOKIE, '', expires=0)

def _get_user_token(request):
```

```
        if _AUTH_COOKIE in request.cookies:
                token = json.loads(b64decode(unquote(request.cookies[_AUTH_COOKIE])))
                _verify_token(token)
                return token
        else:
                return None

def _sign_token(token):
        # Fetch secret key from temp file. Generate it the first time.
        keypath = '/tmp/secure-secret.key'
        if not os.path.exists(keypath):
                with open(keypath, 'w') as keyfile:
                        keyfile.write(hexlify(os.urandom(16)).decode())

        # Compute HMAC over token fields.
        sign_data = token['email'] + token['role'] + token['fullname']
        with open(keypath, 'r') as keyfile:
                key = unhexlify(keyfile.read().strip())
                return hmac.new(key, sign_data.encode(), 'sha256').hexdigest()

def _verify_token(token):
        # Make sure all fields are there.
        if set(token.keys()) != set(_TOKEN_FIELDS):
                raise Exception('Invalid token fields.')

        # Validate the signature.
        if token['signature'] != _sign_token(token):
                raise Exception('Invalid token signature.')
```

We can see that the server is creating a signed authentication token for us, containing our email, fullname and our role, and stores it as a cookie. If we take a look at `admin_check`, we can see that there is a hidden page under `/admin`, which we can only access if we have a signed token with the role set to `Admin`. Looking at `_sign_token`, we can see that it calculates the hmac by just concatenating our email, role and fullname, without any differentiation between these variables. If we take a look at an example we can see that this is easily exploitable:

field	value
email	a@a.Admin
role	Guest
fullname	a a

`sign_data = a@a.AdminGuesta a`

field	value
email	a@a.
role	Admin
fullname	Guesta a

`sign_data = a@a.AdminGuesta a`

While both tokens share the same hmac, one has the role `Admin`, while the other one is a normal `Guest` account. With that knowledge we can create a token with the fields from the first example and use the hmac we get from that token to craft our own token, authenticating us as an admin. If we use this token to access the `/admin` page, we get the admin panel, containing the flag.

Mitigation

To mitigate such an attack, one can just use a delimiter between the variables the user cannot use in his input (a null byte for example) to create ones `sign_data`.