Security

CITS3403 and CITS5505 - Agile Web Development

Reading: The Flask Mega-Tutorial
Miguel Grinberg
Chapter 5

Semester 1, 2024

Security basics

Security on the web

- Security is a primary concern for anyone developing web applications.
- The key assumption is that the network itself cannot be trusted!
- Data access must be controlled, passwords must be validated securely, and users just be able to trust the information presented to them.
- Complete security is very hard to achieve and beyond the scope of this unit!

Breakdown

- Security on the web depends on trust. There are several elements to this:
  1. The web server needs to be confident that someone accessing data is authorised.
     - Achieved via session/token-based authentication.
  2. The user needs to know that the site they are visiting is the one they intend to.
     - Achieved via SSL Certificates.
  3. Both the server and the client need to be confident that no one in the middle is accessing unauthorised data.
     - Achieved via HTTPS (HTTP Secure).
- Underlying all these techniques are public-key encryption and cryptographic hashing.

Public Key Encryption

- Secure communication is based on Public Key Encryption (e.g. RSA). There are two functions, $E_k(\bullet)$ - encrypts with key k, and $D_k(\bullet)$ - decrypts with key k.

- A public-private key is a pair of keys *pub* and *priv* such that
  $$x = D_{priv}(E_{pub}(x)) \quad \text{and} \quad x = D_{pub}(E_{priv}(x))$$
  and, crucially, you cannot work out what *priv* is even if you know *pub*.

- There are various uses of Public Key Encryption:
  - Authentication. The value of $E_{pub}(x)$ can be published and only someone who knows *priv* can work out what x is by computing $D_{priv}(E_{pub}(x))$.
  - Digital signatures. The pair $(x, E_{priv}(x))$ can be verified by anyone by computing $D_{pub}(E_{priv}(x))$ and comparing it to x, but only created by someone who knows *priv.*
  - Key distribution. A new random key x can be generated, and $E_{pub}(x)$ can be sent to someone who knows *priv*. The pair then knows x, but nobody else does.

Hashing

- Secure data storage is based on Cryptographic Hashing (e.g. MD5)
  - A cryptographic hash function is a function *hash* that takes a string or arbitrary length and returns a fixed length integer such that:
    - Given a value h it is difficult to find an x such that $hash(x) = h$
    - Given a value x, it is difficult to find a y such that $hash(x) = hash(y)$
  - Essentially it computes a number from a stream of data, in such a way that it's very difficult to fabricate data with a certain hash.
  - Related to hashing for hash tables, but the *hash* function must satisfy the much stronger properties above to be cryptographically secure.

- Useful for:
  - Storing passwords securely – given password p, you can store $h = hash(p)$ in your database instead of p and when the user tries to login with password q, you can test whether $hash(q) = h$.

Session-based user authentication

Overview of session authentication

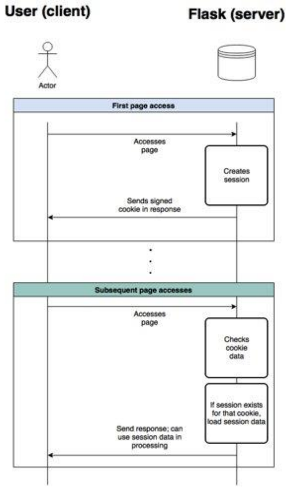
- Although HTTP is stateless, the application is not!

- To track a user's session, we need to have them register so we can associate a username and password with them. The password is salted and hashed, and the result is stored in the database.

- When someone logs in, they provide the same password. This again is salted and hashed to provide a digest which can compared to the hash in a database.

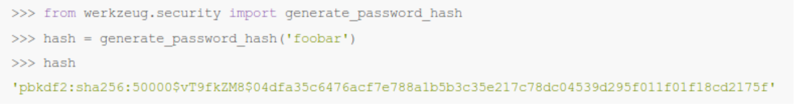
- If successful, the server's response contains a cookie with the session ID and *hash(ID + secret).*

- Whenever a future HTTP request is made, the server checks the session ID in the cookie and recomputes *hash(ID + secret*) to check the ID hasn't been altered.

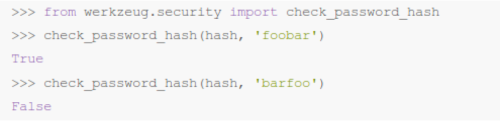

User (client)
Flask (server)
Actor
First page access
Accesses page
Creates session
Sends signed cookie in response
Subsequent page accesses
Accesses page
Checks cookie data
If session exists for that cookie, load session data
Send response; can use session data in processing

Using Hashing in Flask applications

- In previous lectures we had a simple database which contained a table of users.
- To add in authentication, we need every user to have a unique user ID and a password, but we only want to store password hashes.
- The Python package werkzeug (a part of Flask) can handle the hashing.



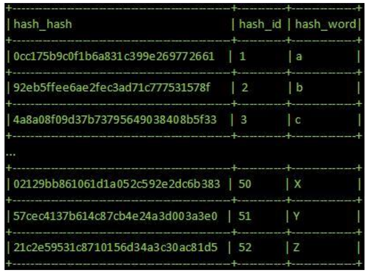
```
>>> from werkzeug.security import generate_password_hash
>>> hash = generate_password_hash('foobar')
>>> hash
'pbkdf2:sha256:50000$vT9fkZM8$04dfa35c6476acf7e788a1b5b3c35e217c78dc04539d295f011f01f18cd2175f'
```



```
>>> from werkzeug.security import check_password_hash
>>> check_password_hash(hash, 'foobar')
True
>>> check_password_hash(hash, 'barfoo')
False
```

Salting your hashes

- Hashing a password on its own is not sufficient for security. If the attacker gets your database, then they can pre-compute the hashes of common passwords via rainbow tables.

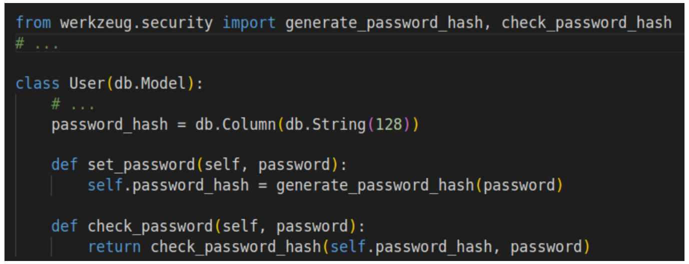


```
+----------------------------------+---------+-----------+
| hash_hash                        | hash_id | hash_word |
+----------------------------------+---------+-----------+
| 0cc175b9c0f1b6a831c399e269772661 | 1       | a         |
+----------------------------------+---------+-----------+
| 92eb5ffee6ae2fec3ad71c777531578f | 2       | b         |
+----------------------------------+---------+-----------+
| 4a8a08f09d37b73795649038408b5f33 | 3       | c         |
+----------------------------------+---------+-----------+
...
+----------------------------------+---------+-----------+
| 02129bb861061d1a052c592e2dc6b383 | 50      | X         |
+----------------------------------+---------+-----------+
| 57cec4137b614c87cb4e24a3d003a3e0 | 51      | Y         |
+----------------------------------+---------+-----------+
| 21c2e59531c8710156d34a3c30ac81d5 | 52      | Z         |
+----------------------------------+---------+-----------+
```

- The solution is to salt your password with a random salt *s* and store *(s, hash(p + s))* in your database instead of just *hash(p)*.
- As the salt is different for each user, the attacker can't pre-compute a rainbow table. Furthermore, the same passwords are hashed to different values.

Adding passwords in Flask

- The password management methods can now be added to the User model, using `werkzeug` to generate and verify hashes.

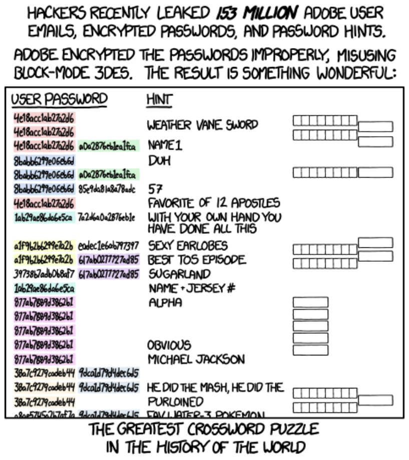


```
from werkzeug.security import generate_password_hash, check_password_hash
# ...

class User(db.Model):
    # ...
    password_hash = db.Column(db.String(128))

    def set_password(self, password):
        self.password_hash = generate_password_hash(password)

    def check_password(self, password):
        return check_password_hash(self.password_hash, password)
```

- The function `generate_password_hash` automatically handles salting for you.

Best practices for hashing

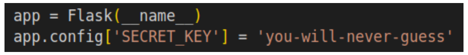
- Cryptographic hash functions are notoriously difficult to get right.
- Never try to create your own or try to mix and match different hash functions.
- Many (very expensive!) horror stories out there!


HACKERS RECENTLY LEAKED 153 MILLION ADOBE USER EMAILS, ENCRYPTED PASSWORDS, AND PASSWORD HINTS.
ADOBE ENCRYPTED THE PASSWORDS IMPROPERLY, MISUSING BLOCK-MODE 3DES. THE RESULT IS SOMETHING WONDERFUL:
USER PASSWORD
HINT
WEATHER VANE SWORD
NAME1
DUH
57
FAVORITE OF 12 APOSTLES
WITH YOUR OWN HAND YOU HAVE DONE ALL THIS
SEXY EARLOBES
BEST TOS EPISODE
SUGARLAND
NAME + JERSEY #
ALPHA
OBVIOUS
MICHAEL JACKSON
HE DID THE MASH, HE DID THE
PURLOINED
FAV WATER-3 POKEMON
THE GREATEST CROSSWORD PUZZLE IN THE HISTORY OF THE WORLD

https://xkcd.com/1286/
(see https://www.explainxkcd.com/wiki/index.php/1286:_Encryptic for the solution!)

Sessions in Flask

- Flask comes with the notion of a general `session`.
- When Flask is configured with a "secret key" it will automatically include a signed cookie with any response to a client that has been "added to" the session.
- We have already seen how to set the secret key in Flask when looking at creating forms with Jinja.

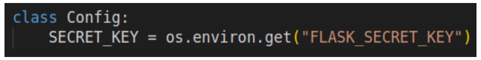


```
app = Flask(__name__)
app.config['SECRET_KEY'] = 'you-will-never-guess'
```

- However, storing it explicitly in the source file is a bad idea and we will now look at a better way....

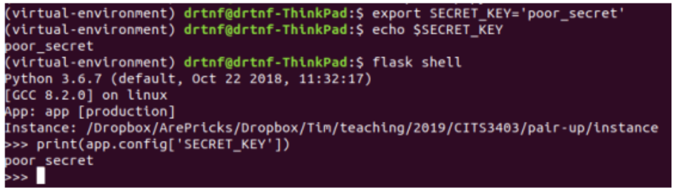
Configuring secret keys

- Secret keys and credentials for third party services should always be manually configured, and never stored under version control in the Git repository.
- Therefore, rather than keeping them in source code, a much better approach is to set them via system variables.



```
class Config:
    SECRET_KEY = os.environ.get("FLASK_SECRET_KEY")
```

- Create a configuration file to store all configuration variables. This can then be loaded when the app runs.



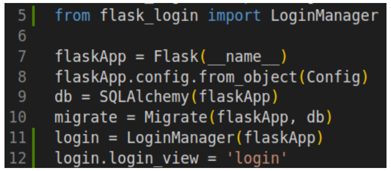
```
(virtual-environment) drtnf@drtnf-ThinkPad:$ export SECRET_KEY='poor_secret'
(virtual-environment) drtnf@drtnf-ThinkPad:$ echo $SECRET_KEY
poor_secret
(virtual-environment) drtnf@drtnf-ThinkPad:$ flask shell
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
App: app [production]
Instance: /Dropbox/ArePricks/Dropbox/Tim/teaching/2019/CITS3403/pair-up/instance
>>> print(app.config['SECRET_KEY'])
poor_secret
>>> 
```

Setting up Flask-Login

- Flask sessions are general and support the storing of arbitrary state about the current user.
- The package Flask-Login provides an interface for implementing logging-in using the provided session infrastructure, and is installed via:

  `pip install flask-login`

- First, an instance of LoginManager is created in `__init__.py`.

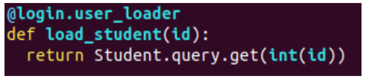


```
from flask_login import LoginManager

flaskApp = Flask(__name__)
flaskApp.config.from_object(Config)
db = SQLAlchemy(flaskApp)
migrate = Migrate(flaskApp, db)
login = LoginManager(flaskApp)
login.login_view = 'login'
```

- The `login_view` should be set to the name of the route that loads the login screen.

Loading users

- As the package is agnostic to the database or ORM being used, we need to tell Flask-Login how to load a "user" object.
- Concretely, the decorator @login.user_loader can be added to the method that maps an ID to a user.



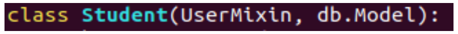
```
@login.user_loader
def load_student(id):
  return Student.query.get(int(id))
```

- This method usually lives in `models.py`

Adding login methods for users

- The Flask-Login package also requires the user model to implement several methods and properties:
  1. `is_authenticated` – does the user have valid login credentials?
  2. `is_active` – is the user allowed to login?
  3. `is_anonymous` – is the user anonymous? (only true for at most one!)
  4. `get_id()` - returns the unique ID for the user

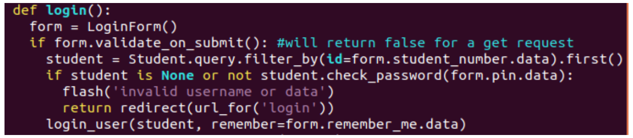
- This functionality can be achieved by using the UserMixin, which implements those methods for you with sensible defaults.



```
class Student(UserMixin, db.Model):
```

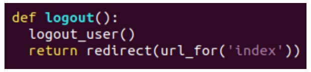
Utilities provided by Flask-Login

- Flask-Login provides various properties and methods
  - `current_user` – the current user model that is logged in. If no user is logged in than returns a special anonymous user object.
  - `login_user()` - sets `current_user` to the specified user model



```
def login():
  form = LoginForm()
  if form.validate_on_submit(): #will return false for a get request
    student = Student.query.filter_by(id=form.student_number.data).first()
    if student is None or not student.check_password(form.pin.data):
      flash('invalid username or data')
      return redirect(url_for('login'))
    login_user(student, remember=form.remember_me.data)
```

  - `logout_user()` - sets `current_user` back to the anonymous user.

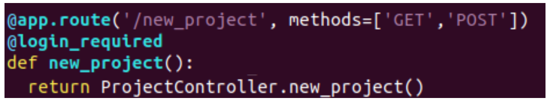


```
def logout():
  logout_user()
  return redirect(url_for('index'))
```

Using Flask-Login

- We can use `current_user.is_authenticated` to check if the current user is logged in.

- We also use a decorator @login_required from Flask-Login to label the routes that require a login.



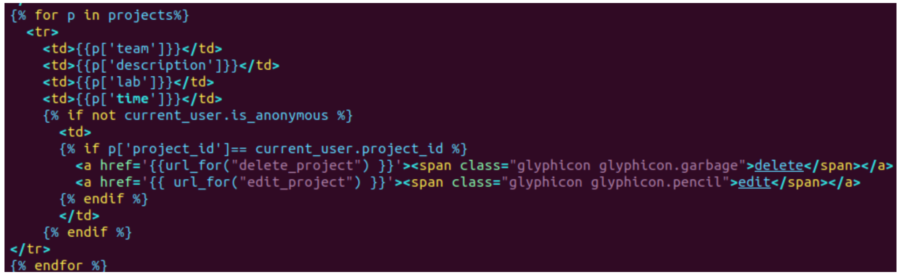
```
@app.route('/new_project', methods=['GET','POST'])
@login_required
def new_project():
  return ProjectController.new_project()
```

- If someone who is not logged in attempts to access this route, they will automatically receive a response that indicates that they are not authorised to access that page.

Updating the view

- We can also use the current_user variable in Jinja templates.

- Its properties can be used to guard components of the web page that you only want logged in users to see, or to personalise the web page.

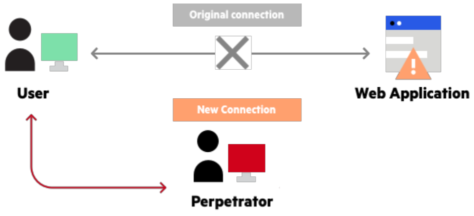


```
{% for p in projects%}
  <tr>
    <td>{{p['team']}}</td>
    <td>{{p['description']}}</td>
    <td>{{p['lab']}}</td>
    <td>{{p['time']}}</td>
    {% if not current_user.is_anonymous %}
      <td>
      {% if p['project_id']== current_user.project_id %}
        <a href='{{url_for("delete_project") }}'><span class="glyphicon glyphicon.garbage">delete</span></a>
        <a href='{{ url_for("edit_project") }}'><span class="glyphicon glyphicon.pencil">edit</span></a>
      {% endif %}
      </td>
    {% endif %}
</tr>
{% endfor %}
```

Attacks on session-based protocols

Impersonation attack

- Sessions allow the server to authenticate the user. However, remember the network is still fundamentally insecure!
- Without extra security, the easiest attack is an impersonation attack.
  - The network redirect the user's request to a fake server.
  - The fake server acts just like the real server.
  - User submits passwords, private data etc. to the fake server.

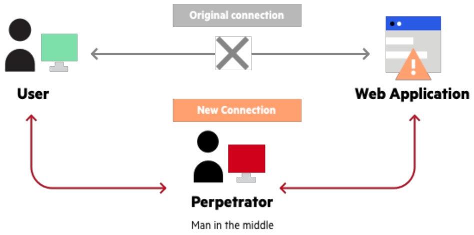


SSL certificates

- An impersonation attack can be prevented by the server a certificate that it's real via the Secure Socket Layer (SSL) protocol. Certificates are provided by a trusted third party known as a Certificate Authority (CA).
- When a server requests a certificate it sends their public key *s_pub* to the CA, who, after various checks, returns (s_pub, $E_{c\_priv}(s\_pub)$) where *c_priv* is the private key of the CA.
- When a user contacts the server, the server sends them (s_pub, $E_{c\_priv}(s\_pub)$), and the user can then use *c_pub*, the CA's public key, to compare *s_pub* to $D_{c\_pub}(E_{c\_priv}(s\_pub))$.
- If it matches, then the user knows the CA has vouched that the server has the public key *s_pub*, and only the real server knows the private key *s_priv* and therefore can respond to messages encrypted with *s_pub*.
- What's not solved: how do you trust the CA is real? Answer: root certificates.

Man-in-the-middle attack

- Even when using certificates, the network can still perform man-in-the-middle attack.
  - All traffic is first redirected to the attacker and then forwarded on to the real server.
  - Because the user is interacting with the real server, the certificate check passes.
  - However, the attacker still gets access to all the information sent back and forth, including the signed cookies.



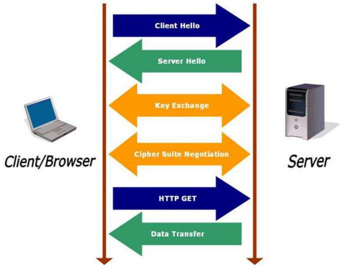
Encrypted sessions with SSL

- We cannot prevent the network from redirecting the messages via a third party, so we must encrypt our messages.

- SSL also includes a public key encryption process to enable secure HTTP requests.

- After the initial certificate handshake establishing that the server has the public key, both parties use a key distribution protocol to generate a shared set of private session keys.

- They then use these keys to encrypt all future traffic between them during that session.

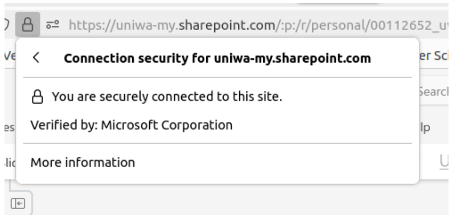


HTTPS

- In theory SSL can be used to encrypt any data between two end points.

- In practice, the most common type of traffic is simply HTTP requests. Using HTTP over SSL is known as HTTPS (HTTP Secure).

- Modern browsers highlight websites that don't use HTTPS, in order to motivate adoption.



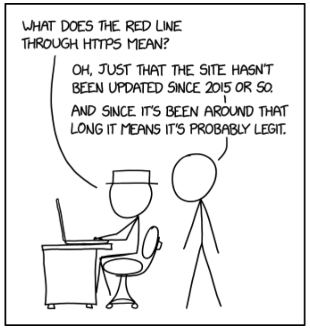
https://uniwa-my.sharepoint.com/:p:/r/personal/00112652_u

Connection security for uniwa-my.sharepoint.com

You are securely connected to this site.

Verified by: Microsoft Corporation

More information


WHAT DOES THE RED LINE THROUGH HTTPS MEAN?
OH, JUST THAT THE SITE HASN'T BEEN UPDATED SINCE 2015 OR SO.
AND SINCE IT'S BEEN AROUND THAT LONG IT MEANS IT'S PROBABLY LEGIT.

https://xkcd.com/2634/

Cross-site request forgery (CRSF) attack

- However, even using certificates and HTTPS doesn't make your application safe (nothing ever does!).

- A common attack that exploits the facts that users may already be authenticated to a server is a cross-site forgery attack.

- Suppose you've recently logged in to your bank at www.mybank.com and therefore have a signed cookie from their server sitting in your browser.

- You then receive an email saying:

> Hi Taylor,
> There are some urgent new instructions for submitting your Agile Web Dev group project! You can find them here.
> Best,
> Matthew

Cross-site request forgery (CRSF) attack

- The actual destination of the URL of the link is something like:

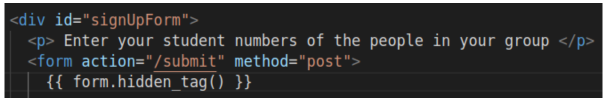
```
www.mybank.com/transfer?from=savings&destBSB=112043&destNo=1230
111&amount=500.00
```

- Remember that cookies are transferred automatically when making the request to the domain they are associated with, and therefore the signed cookie is sent to the server.

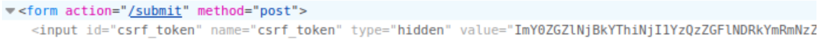
- The server checks the cookie, which passes as it is real, and therefore thinks the request is from the user, and the transfer goes through!

Defending against CRSF

- We have already seen how to solve this problem. When the server renders the form, it generates a secret key and includes it in the form.

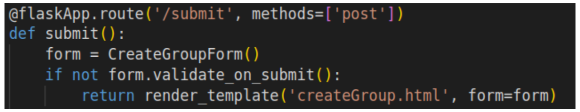


```
<div id="signUpForm">
  <p> Enter your student numbers of the people in your group </p>
  <form action="/submit" method="post">
    {{ form.hidden_tag() }}
```



```
▼<form action="/submit" method="post">
   <input id="csrf_token" name="csrf_token" type="hidden" value="ImY0ZGZlNjBkYThiNjI1YzQzZGFlNDRkYmRmNzZ
```

- When the form is submitted, the secret key is checked during `validate_on_submit`.



```
@flaskApp.route('/submit', methods=['post'])
def submit():
    form = CreateGroupForm()
    if not form.validate_on_submit():
        return render_template('createGroup.html', form=form)
```

- In this way, the server will only respond to requests generated via an official form.

- Additionally, safety-critical apps will often time-out signed cookies out after ~5 minutes.

Token-based user authentication

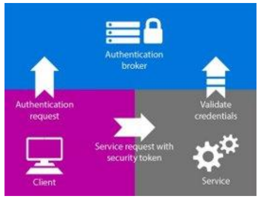
Downsides of session-based authentication

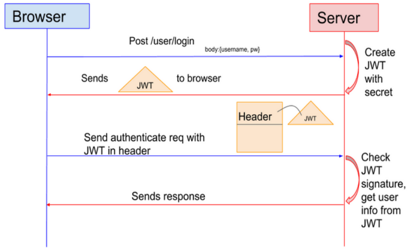
- Session-based authentication works for web-based applications but has several drawbacks.
  1. It requires the application to track all user sessions which may not scale well.
  2. HTTP requests are sent in plain text, and passwords should never be transmitted or stored in plain text.
  3. The user may be accessing the application via a mobile application where there is no such thing as a cookie.
  4. It is inherently stateful, which may not be necessary (e.g. for API access).
  5. It requires users to have registered their information with the server.

- What if we only want to check authorisation rather than authentication?

Tokens and JWT

- From a cryptographic perspective, tokens are the same as session cookies, but unlike session cookies, tokens are not usually used to authenticate a user and retrieve the state on the server.
- Instead, they authorise the holder to perform a particular action, e.g. retrieve data from a public API.
- Crucially, it is not necessary for the server to know who the requester is.





Advantages of tokens

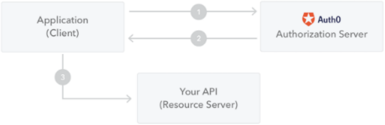
- By sacrificing the ability to store state, tokens have many advantages over sessions:
  1. Validating a token doesn't require querying the database.
  2. Tokens aren't vulnerable to CRSF attacks (as long as they are not stored in cookies!).
  3. They don't require cookies and therefore work on mobile apps.
  4. Tokens can store a set of privileges, rather than just one.

- Tokens are also often issued by trusted third parties (e.g. Twitter, Google) instead of the server itself. Therefore, the server doesn't have to store sensitive user information!

- Note that *obtaining* the token still usually requires authentication via a password-protected user account! Therefore, state is still required, but only at the beginning.

OAuth

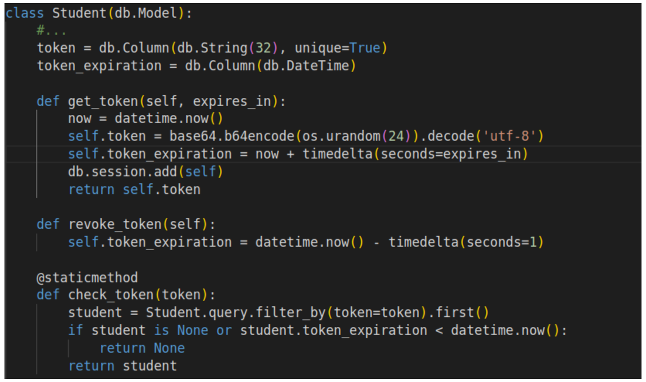
- OAuth was developed by Twitter to allow applications to authenticate and interact with Twitter, without requiring repeated logins.
- OAuth verifies a user's identity and provides a JSON Web Token (JWT). This contains a header, a payload and a signature in compressed JSON.
- This header describes the encryption type, the payload typically provides some user identifier, an expiry and issuer information, and the signature is a secure hash, proving the token wasn't tampered with.



1. The application or client requests authorization to the authorization server. This is performed through one of the different authorization flows. For example, a typical OpenID Connect compliant web application will go through the `/oauth/authorize` endpoint using the authorization code flow.
2. When the authorization is granted, the authorization server returns an access token to the application.
3. The application uses the access token to access a protected resource (like an API).

Creating tokens in Flask

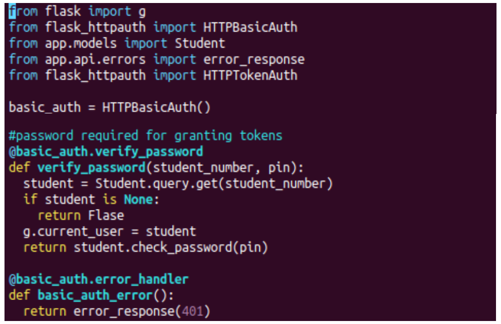
- You can configure Flask to both serve OAuth tokens to clients, and verify those tokens, with the `flask_oauth` module.
- We need to first update our user models so that the temporary token is kept in the database, as well as the password hash.



```
class Student(db.Model):
    #...
    token = db.Column(db.String(32), unique=True)
    token_expiration = db.Column(db.DateTime)

    def get_token(self, expires_in):
        now = datetime.now()
        self.token = base64.b64encode(os.urandom(24)).decode('utf-8')
        self.token_expiration = now + timedelta(seconds=expires_in)
        db.session.add(self)
        return self.token

    def revoke_token(self):
        self.token_expiration = datetime.now() - timedelta(seconds=1)

    @staticmethod
    def check_token(token):
        student = Student.query.filter_by(token=token).first()
        if student is None or student.token_expiration < datetime.now():
            return None
        return student
```

Checking passwords in Flask

- The `HTTPBasicAuth` module is used for verifying passwords in a request without using the Flask-Login package.

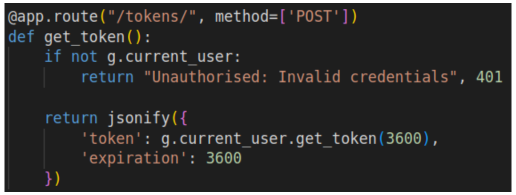


```
from flask import g
from flask_httpauth import HTTPBasicAuth
from app.models import Student
from app.api.errors import error_response
from flask_httpauth import HTTPTokenAuth

basic_auth = HTTPBasicAuth()

#password required for granting tokens
@basic_auth.verify_password
def verify_password(student_number, pin):
  student = Student.query.get(student_number)
  if student is None:
    return Flase
  g.current_user = student
  return student.check_password(pin)

@basic_auth.error_handler
def basic_auth_error():
  return error_response(401)
```

- The field `g` is a Flask context object that comes with each HTTP request

Serving tokens in Flask

- We can then construct a POST request that authenticated users can use to obtain a valid token.

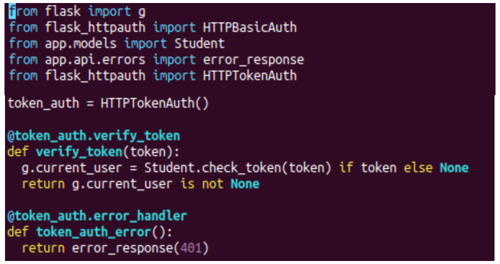


```
@app.route("/tokens/", method=['POST'])
def get_token():
    if not g.current_user:
        return "Unauthorised: Invalid credentials", 401

    return jsonify({
        'token': g.current_user.get_token(3600),
        'expiration': 3600
    })
```

- We set a timeout on the token to 1 hour, so the user only has permissions for a short period of time and must reauthenticate after that.

Verifying tokens in Flask

- Finally, the `HTTPTokenAuth` class enables us to do token-based authentication.

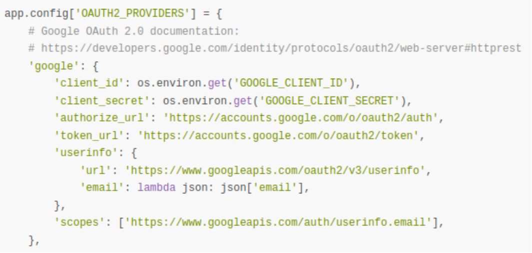


```
from flask import g
from flask_httpauth import HTTPBasicAuth
from app.models import Student
from app.api.errors import error_response
from flask_httpauth import HTTPTokenAuth

token_auth = HTTPTokenAuth()

@token_auth.verify_token
def verify_token(token):
  g.current_user = Student.check_token(token) if token else None
  return g.current_user is not None

@token_auth.error_handler
def token_auth_error():
  return error_response(401)
```

- In this case, requests augmented with that token user are assumed to come from the user.

Interacting with a REST API in Python

- We won't go into the details of how you implement OAuth with third party tokens.



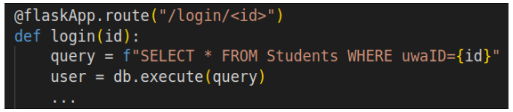
```
app.config['OAUTH2_PROVIDERS'] = {
    # Google OAuth 2.0 documentation:
    # https://developers.google.com/identity/protocols/oauth2/web-server#httprest
    'google': {
        'client_id': os.environ.get('GOOGLE_CLIENT_ID'),
        'client_secret': os.environ.get('GOOGLE_CLIENT_SECRET'),
        'authorize_url': 'https://accounts.google.com/o/oauth2/auth',
        'token_url': 'https://accounts.google.com/o/oauth2/token',
        'userinfo': {
            'url': 'https://www.googleapis.com/oauth2/v3/userinfo',
            'email': lambda json: json['email'],
        },
        'scopes': ['https://www.googleapis.com/auth/userinfo.email'],
    },
```

- Crucially, though it relies on the server being able to send HTTP requests to the third party and receive responses.
- There are various Python packages that allow you to do this such requests and HTTPie.

Other

Cross-site scripting

- You should never directly include unvalidated user input into executable code!
- If you do, then a malicious user can perform a cross-site scripting attack by using unexpected inputs to escape the current statement and begin executing arbitrary code.
- There are many types of CSS attacks, but SQL injection is a common one.
- Suppose a web server is directly constructing SQL queries as follows:

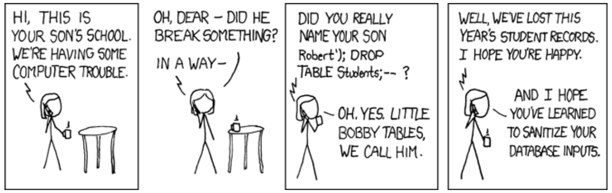


```
@flaskApp.route("/login/<id>")
def login(id):
    query = f"SELECT * FROM Students WHERE uwaID={id}"
    user = db.execute(query)
    ...
```

- Then the user can provide an ID of the form "name; X" where "X" is an arbitrary SQL statement.

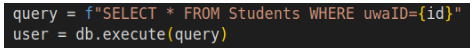
Cross-site scripting example

- Then the following problem occurs:


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

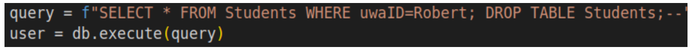
https://xkcd.com/327/

- In particular, the following line



```
query = f"SELECT * FROM Students WHERE uwaID={id}"
user = db.execute(query)
```

becomes



```
query = f"SELECT * FROM Students WHERE uwaID=Robert; DROP TABLE Students;--
user = db.execute(query)
```

Sanitising user inputs

- One of the easiest approaches is to automatically escape any control characters present in the string.
- In SQL this involves inserting a backslash before a character.
- One of the advantages of the SQLAlchemy `query` API is that it automatically escapes any string passed in
  - e.g. `Student.query.get(id)` will automatically escape everything in id.
- Likewise, Jinja automatically escapes any strings substituted into HTML templates.
- Other approaches include input validation, prepared queries etc.

Case study

- The case of Anonymous's attack on the security company HBGary is a fascinating exploration of how what seems to be small security problems in isolation can lead to an entire company be compromised:

arstechnica.com/tech-policy/2011/02/anonymous-speaks-the-inside-story-of-the-hbgary-hack/

Security in the group project

- In the group project you won't be expected to implement everything!

- You will be expected to:
  - Implement user authentication using Flask-Login.
  - Use SQLAlchemy to avoid CSS attacks.
  - Use WTForms tokens to avoid CRSF attacks.

- You will not be expected to:
  - Obtain an SSL certificate or setup HTTPS
  - Implement token-based authorisation