Hashing and Login

Download Demo Code <../flask-hashing-login-demo.zip>

Goals

- Define hashing
- Implement authentication and authorization in flask with Bcrypt

Registering and Logging In

User Class

demo/badpassword/models.py

```
class BadUser(db.Model):
    "Site user."

    __tablename__ = "bad_users"

    id = db.Column(db.Integer,
                   primary_key=True,
                   autoincrement=True)

    username = db.Column(db.Text,
                         nullable=False,
                         unique=True)

    password = db.Column(db.Text,
                         nullable=False)
```

Registration

demo/badpassword/app.py

```
@app.route("/register", methods=["GET", "POST"])
def register():
    """Register user: produce form & handle form submission."""

    form = RegisterForm()

    if form.validate_on_submit():
        name = form.username.data
        pwd = form.password.data

        user = BadUser(username=name, password=pwd)
        db.session.add(user)
        db.session.commit()
```

```
        # on successful login, redirect to secret page
        return redirect("/secret")

    else:
        return render_template("register.html", form=form)
```

Login

demo/badpassword/app.py

```
@app.route("/login", methods=["GET", "POST"])
def login():
    """Produce login form or handle login."""

    form = LoginForm()

    if form.validate_on_submit():
        name = form.username.data
        pwd = form.password.data

        user = BadUser.query.filter_by(username=name).first()

        if user and user.password == pwd:
            # on successful login, redirect to secret page
            return redirect("/secret")

        else:
            # re-render the login page with an error
            form.username.errors = ["Bad name/password"]

    return render_template("login.html", form=form)
```

Database

```
SELECT * FROM bad_users;
```

username	password
rita	squid-13
roger	meeples4ever

Ut oh.

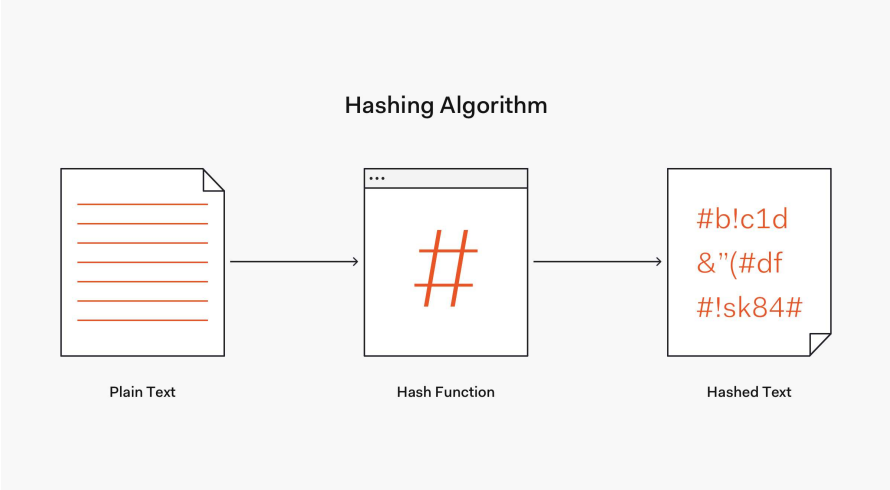
Plaintext Passwords

- Access to database allows access to all passwords!
- People use same passwords for multiple sites
- Don't ever do this! Useful discussion <https://stackoverflow.com/questions/2283937/how-should-i-ethically-approach-user-password-storage-for-later-plaintext-retrie> about this

Hashing

Define Hashing

Hashing performs a one-way transformation on a password.



<_images/hash-flow.png>

"One-way" means it is virtually impossible to reverse.

A Basic Hash

"One-way encryption"

demo/badhash.py

```
def awful_hash(phrase):
    """Truly terrible hash:
        simply shifts each letter (a->b, etc).

        >>> awful_hash('yay')
        'zbz'
    """

    return ''.join(next_char(c) for c in phrase)
```

But is that really one-way?

One-Way Encryption

demo/badhash.py

```
def slightly_better_hash(phrase):
    """Better hash: returns every other letter, shifted, max 4.

        >>> slightly_better_hash('penguin1')
        'qovo'

    Since this is "lossy", multiple inputs return same output:
```

```
        >>> slightly_better_hash('penguin1~pretzel7')
        'qovo'

        >>> slightly_better_hash('p?nguinZ')
        'qovo'
    """

    return ''.join(next_char(c) for c in phrase[0:8:2])
```

- Now is one-way (non-reversible)
- Same input always equal same output

Python has this kind of hash built-in:

```
>>> hash('penguin1')
6678229702981429425
```

(Python's built in ***hash*** seeds itself randomly on startup, so the same input only returns the same output for any individual Python process. As such, it's not suitable for storing in a database, even if it were designed to be cryptographically secure.)

Salt

Salt: a random string introduced before hashing.

password	salt	hashed result
penguin1	xab17	qovoyc8|xab17
penguin1	meeps	qovonft|meeps

Salt is usually concatenated to the password, then hashed.

demo/badhash.py

```
def salting_hash(phrase, salt=None):
    """Adds random salt; returns "salt|hash(phrase+salt)

        >>> salting_hash('hey', salt='abc')
        'izbd|abc'

        >>> salting_hash('hey', salt='def')
        'izeg|def'
    """

    if salt is None:
        salt = str(randint(1000, 9999))

    hashed = slightly_better_hash(f"{phrase}|{salt}")
    return f"{hashed}|{salt}"
```

Cryptographic Hash

- Non-reversible
- Change in input changes output unpredictably

password	salt	hashed result
penguin1	xab17	dsfdsfj33gw|xab17
penguin2	xab17	ewruoi3kl1z|xab17
penguin2	meeps	kj34kjkf28z|meeps

The same password will generate a different hash with a different salt.

> **Note: Storing Salt**
>
> You may have noticed that the salt is clearly visible in the hashed result.
>
> This is totally fine; the application needs the salt value in order to compare passwords properly. Even if an attacker gained access to the database and saw all the salts, they would still have to reverse the hashing algorithms to get the original password, which is extremely difficult and slow.
>
> You can read more about storing salts <https://security.stackexchange.com/questions/17421/how-to-store-salt>.

Popular Algorithms

Cryptographic Hashes:

- MD5
- SHA *(family)*

(Fast, non-reversible, output very different)

Password Hashes:

- Argon2
- Bcrypt
- Scrypt

(Same but slow and hard to optimize)

Bcrypt

```
>>> import bcrypt   # pip install bcrypt

>>> salt = bcrypt.gensalt()
>>> salt
b'$2b$12$uYNRTDE7RrMvwDcF9f1Yyu'

>>> bcrypt.hashpw(b'secret', salt)
b'$2b$12$uYNRTDE7RrMvwDcF9f1Yyuvuu48PzANrWy88Iz3z1tRTfdXi6DlNW'
```

Work Factor

- Bcrypt algorithm is designed to be slow
  - But computers get faster all the time!
- So, you can specify how many rounds of encryption it should use
  - And, over time, increase this “work factor” of work

```
>>> salt = bcrypt.gensalt(rounds=14)  # default (in 2018) is 12
```

This is encoded in the result:

```
b'$2b$12$3cy0jD1AfgcT0ipGL1UhquBZXvAxUwRrdG90Gi951AcxIXm2F2gMK'
```

- **Prefix:** `2b` *(identifies as Bcrypt algorithm)*
- **Work Factor:** `12` rounds
- **Salt:** `3cy0jD1AfgcT0ipGL1Uhqu`
- **Hash:** `BZXvAxUwRrdG90Gi951AcxIXm2F2gMK`

Flask Password Hashing

Flask-Bcrypt

A nicer API for Bcrypt:

```
>>> from flask_bcrypt import Bcrypt

>>> bcrypt = Bcrypt()

>>> hash = bcrypt.generate_password_hash("secret")
>>> hash
b'$2b$12$s.tjeALK2I7rfI2gV27me.mkZu5IQd1Y1EBAXsbTvNExIEQcID/te'

>>> bcrypt.check_password_hash(hash, "secret")
True
```

Class Methods

It’s good to move logic out of views

Let’s make convenient class methods for registering & validating

Registering

demo/goodpassword/models.py

```
class User(db.Model): # ...
    @classmethod
```

```
    def register(cls, username, pwd):
        """Register user w/hashed password & return user."""

        hashed = bcrypt.generate_password_hash(pwd)
        # turn bytestring into normal (unicode utf8) string
        hashed_utf8 = hashed.decode("utf8")

        # return instance of user w/username and hashed pwd
        return cls(username=username, password=hashed_utf8)
```

Authenticating

demo/goodpassword/models.py

```
class User(db.Model): # ...
    @classmethod
    def authenticate(cls, username, pwd):
        """Validate that user exists & password is correct.

        Return user if valid; else return False.
        """

        u = User.query.filter_by(username=username).first()

        if u and bcrypt.check_password_hash(u.password, pwd):
            # return user instance
            return u
        else:
            return False
```

Using Class Methods

```
>>> roger = User.register("roger", "cupcakes")
>>> db.session.add(roger)
>>> db.session.commit()
```

```
>>> User.authenticate("roger", "cupcakes")
<User 3>
```

Checking Our Database

```
SELECT * FROM users;
```

username	password
rita	$2b$12$KD6YjzB6jyDUYxS3E/QDMeaLosFsnG/G6UVv6Ls3rWolypPXmU4LO
roger	$2b$12$/GIp9nJDuoEinr4b1IbUKOXKfTANIABT47jJhFDX.jIhHft9taePi

Encrypted! 🎉

User Sessions

How Do We Remember a Logged In User?

When they sign up or authenticate, store their ***user_id*** in the session:

demo/goodpassword/app.py

```
@app.route("/login", methods=["GET", "POST"])
def login():
    """Produce login form or handle login."""

    form = LoginForm()

    if form.validate_on_submit():
        name = form.username.data
        pwd = form.password.data

        # authenticate will return a user or False
        user = User.authenticate(name, pwd)

        if user:
            session["user_id"] = user.id  # keep logged in
            return redirect("/secret")

        else:
            form.username.errors = ["Bad name/password"]

    return render_template("login.html", form=form)
```

Keeping a User Logged In

Anywhere: you can check if ***user_id*** is in session:

demo/goodpassword/templates/index.html

```
  {% if 'user_id' in session %}
    <li><a href="/logout">Logout</a></li>
    <li><a href="/secret">Secret</a></li>
  {% endif %}
```

Ensuring That User Is Authenticated

On any “protected” or “secret” route…

demo/goodpassword/app.py

```
@app.route("/secret")
def secret():
    """Example hidden page for logged-in users only."""
```

```
    if "user_id" not in session:
        flash("You must be logged in to view!")
        return redirect("/")

        # alternatively, can return HTTP Unauthorized status:
        #
        # from werkzeug.exceptions import Unauthorized
        # raise Unauthorized()

    else:
        return render_template("secret.html")
```

Logging Out

Just remove ***user_id*** from the session!

demo/goodpassword/app.py

```
@app.route("/logout")
def logout():
    """Logs user out and redirects to homepage."""

    session.pop("user_id")

    return redirect("/")
```