

Assignment 1

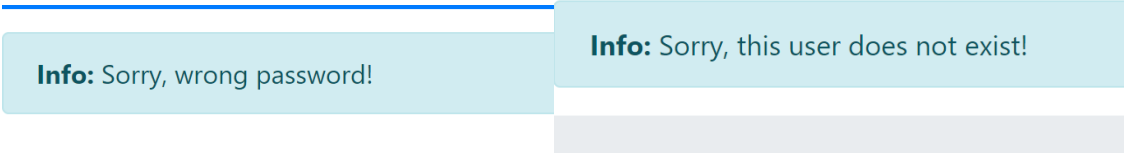
*by: William Dahal, Eivind Gederø,
Harykaran Lambotharan, Torjus Tuft and
Truls Øvrebø*

The Vulnerability List:

- **Login Response**
- **Register Response**
- **Existing User Response**
- **Hash Password**
- **Safe URL**
- **Flask-Login**
- **Cookies**
- **CSRF Token**
- **CSP Header**
- **File Upload**
- **SQL Injection**

Login Response

Problem



Info: Sorry, wrong password!

Info: Sorry, this user does not exist!

As we can see from the attachment above, attempting to log in with the wrong username or password tells you whichever one you had wrong. This flaw is essentially a helpful tool for any attacker trying to log into an account that is not theirs by simply guessing the username and password.

Solution



Info: Sorry, wrong username or password!

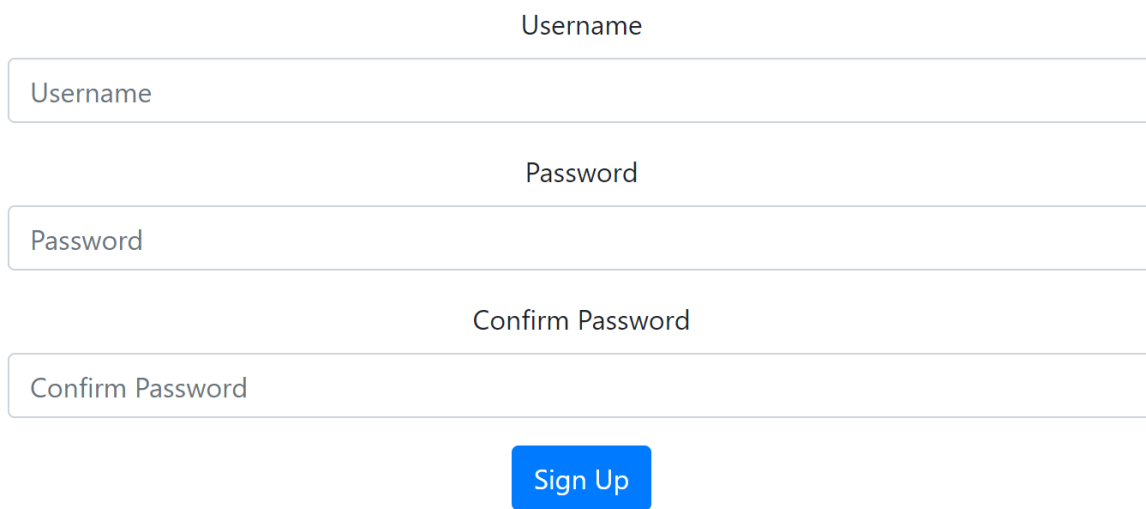
This was changed to “wrong username or password”.

Register Response

Problem

Initially the application had no password requirements, the passwords did not even have to match each other. Therefore many people could be tempted to create easily guessed passwords such as 1234.

Before:



Username

Password

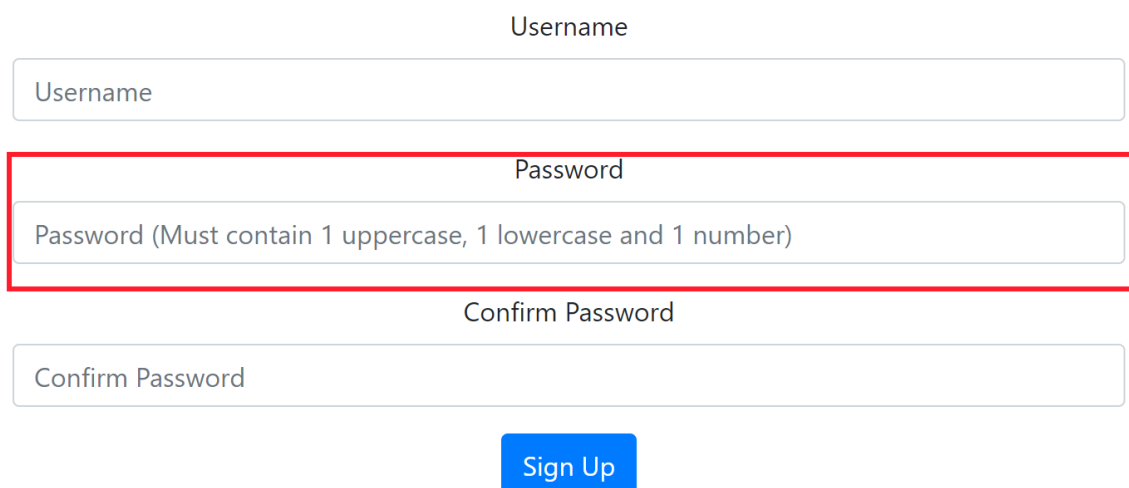
Confirm Password

Sign Up

Solution

Our solution to this was to force users into creating passwords that are difficult to guess. Therefore we got the app to check if the passwords were of at least eight characters in length, contained at least one upper- and lowercase letter, and one or more numbers. A password should be hard to guess, but easy for a user to remember. An example of a good password is “10DaystoRain”.

After:



Username

Password

Confirm Password

Sign Up

Existing User Response

If a username already exists in the database, it has to show an alert that “username is already taken”. By doing this we do not risk to clash two or more same usernames in a database.

Info: Username already taken!

Hash password

Problem

The website has a create account and login function which will save information about the user in (database.db). If any chance a hacker or group of hackers, hacked into the database, they would have access to the usernames and passwords of anyone who has created an account on this website. Considering that most people do not use unique usernames and passwords for different websites, this information could be used to gain even more access to that user's accounts. If someone were to gain access to the database they would have access to all information on it. Therefore we can use a hash password generator to hash the passwords, this will add another obstacle if someone were to access the database.

Solution

Before:

```
elif form.register.is_submitted() and form.register.submit.data:
    query_db('INSERT INTO Users (username, first_name, last_name, password) VALUES("{}","{}","{}","{}");'
              .format(form.register.username.data, form.register.first_name.data,
                      form.register.last_name.data, form.register.password.data))
    return redirect(url_for('index'))
return render_template('index.html', title='Welcome', form=form)
```

Source: picture of old code(register) from routes.py

```
if form.login.is_submitted() and form.login.submit.data:
    user = query_db('SELECT * FROM Users WHERE username="{}";'.format(form.login.username.data), one=True)
    if user == None:
        flash('Sorry, this user does not exist!')
    elif user['password'] == form.login.password.data:
        return redirect(url_for('stream', username=form.login.username.data))
    else:
        flash('Sorry, wrong password!')
```

Source: picture of old code(login) from routes.py

Hashing is used to change the password to a long string of letters and/or numbers using an encryption algorithm. That means hackers have a harder time gaining access to the password, they only get access to the encrypted "hash" code and will not know the real password because of the encryption algorithm. This can be used to generate passwords and also can be used to check logged-in from input.

After:

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

Source: import code from routes.py

Before:

```
elif form.register.is_submitted() and form.register.submit.data and form.register.validate_on_submit():
    # -- |HASHING|
    hashed_password = generate_password_hash(form.register.password.data, method='sha256')

    registration_info = (form.register.username.data, form.register.first_name.data,
                        form.register.last_name.data, hashed_password)
    if register_query(registration_info) == True:
        session["username"] = form.register.username.data
        flash(f"Welcome {form.register.username.data}")
        return redirect(url_for('stream'))
    else:
        flash("Username already taken!")
```

Source: code(register) from routes.py

After:

```
if form.login.is_submitted() and form.login.submit.data:
    user_find = query_db('SELECT * FROM Users WHERE username="{0}"'.format(form.login.username.data), one=True)
    user = load_user(form.login.username.data)
    if form.login.validate_on_submit():
        # -- |HASHING|
        if check_password_hash(user_find['password'], form.login.password.data) == True:
            session["username"] = form.login.username.data
            login_user(user, remember=True)
            flash(f"Welcome {form.login.username.data}!")
            return redirect(url_for('stream'))
        else:
            flash('Sorry, wrong username or password!')
```

Source: new code(login) from routes.py

Safe URL

Without session or flask_login, it is possible to access the user's (stream, friend, profile) by searching the url with the username as the last variable. The original app route ('/stream/<username>') is needed to get the information from (index), and this will be used to get any user information from the database.

Before:



Source: picture of before url for stream

```
@app.route('/stream/<username>', methods=['GET', 'POST'])
def stream(username):
    form = PostForm()
    user = query_db('SELECT * FROM Users WHERE username="{0}"'.format(username), one=True)

    if form.is_submitted():
        if form.image.data:
            path = os.path.join(app.config['UPLOAD_PATH'], form.image.data.filename)
            form.image.data.save(path)
```

Source: old code(stream) from routes.py

Session object lets certain information parameters go across requests, so it can get the information for next url route, and because information is saved to (session["username"]) it will always be saved until value is set to (None) or deleted.

After:



Source: after url for stream

```
@app.route('/stream', methods=['GET', 'POST'])
@login_required
def stream():
    # -- |SESSION|
    session["username"] = current_user.get_id()
    username = session["username"]

    if session["username"] == None:
        return redirect("/index")
    if username != session.get("username"):
        flash("Access denied: Not logged in!")
        return redirect(url_for('index'))
```

Source: new code(stream) from routes.py

Flask-Login

What is flask_login?

Flask_login provides user session management for flask. It handles the common tasks of logging in, logging out, and remembering the users' sessions over extended periods of time.

Flask_login strong sides.

- store the active user's ID in the flask session, and let users easily log in and out.
- let users restrict views to logged-in and logged out.
- Handling the "remember me" function.
- Help to protect users' sessions from being stolen cookie thieves.

Flask_login weak sides

- It does not impose a particular database or other storage method, developers are entirely in charge of how the user is loaded.
- It does not restrict to using usernames and passwords, OpenIDs or any other method of authenticating.
- it does not handle logged in or out beyond permissions. It also can not handle user registration or account recovery

Flask login has different functions that will be used with class(user). User has self variables for username, password and id that will be used to interact with the database. It also has functions(get_id, get_password, get_name_user_query) that will be used afterward.

UserMixin provides properties like metode to check if the information is correct or not, instead of having to write the function yourself. The function(load_user) return the function(get_name_user_query) when the load_user definite in index(login)

```
from flask_login import LoginManager, UserMixin, login_user, login_required, logout_user, current_user
```

Source: import code from routes.py

```
# [NB!] FLASK LOGIN | -----
class User(UserMixin):
    def __init__(self, username, password, id=0):
        self.__id = id
        self.__username = username
        self.__password = password

    def get_id(self):
        return self.__username
    def get_password(self):
        return self.__password
    def get_name_User_query(username):
        query = query_db('SELECT * FROM Users WHERE username="{}";'.format(username), one=True)
        user = User(username=query['username'], password=query['password'], id=query['id'])
        return user

@login_manager.user_loader
def load_user(username):
    return User.get_name_User_query(username)
```

Source: new code of class(User) from routes.py

The website has session and flask login which fixes when a user accidentally closes the browser, it will return to the account if the user opens the website again.

```
user = load_user(form.login.username.data)
if form.login.validate_on_submit():
    # -- |HASHING|
    if check_password_hash(user_find['password'], form.login.password.data) == True:
        session["username"] = form.login.username.data
        login_user(user, remember=True)
        flash(f"Welcome {form.login.username.data}!")
        return redirect(url_for('stream'))
```

Source: new code(login) from routes.py

```
session["username"] = current_user.get_id()
```

Source: new code(stream, friend, profile) from routes.py

Cookies

What is cookies?

cookies are small data files with text information that are used to identify computers. Specific cookies as HTTP cookies are used to identify specific users and improve users' web browsing experience.

Session vs Cookies.

Session is used to temporarily store the information on the server to be used across multiple pages of the website. It ends when the user logs out from the application or closes the web browser. It can also store an unlimited amount of data.

Problem

Without cookies, it will be time consuming for web users. everytime a user has to log in or save data for the web browser to remember.

Solution

Cookies are stored on the user's computer as a text file, and end on the lifetime set by users. It can only store limited amounts of data. Cookies are not secure, as data is stored in a text file, and if any unauthorized users get access to our system, they can temper the data.

How to apply cookies

It has applied the same functions in all sites. Code should automatically make cookies to count every time a user visits that website. it should count differently on all different sides. for example one count for index and another one for friend side.

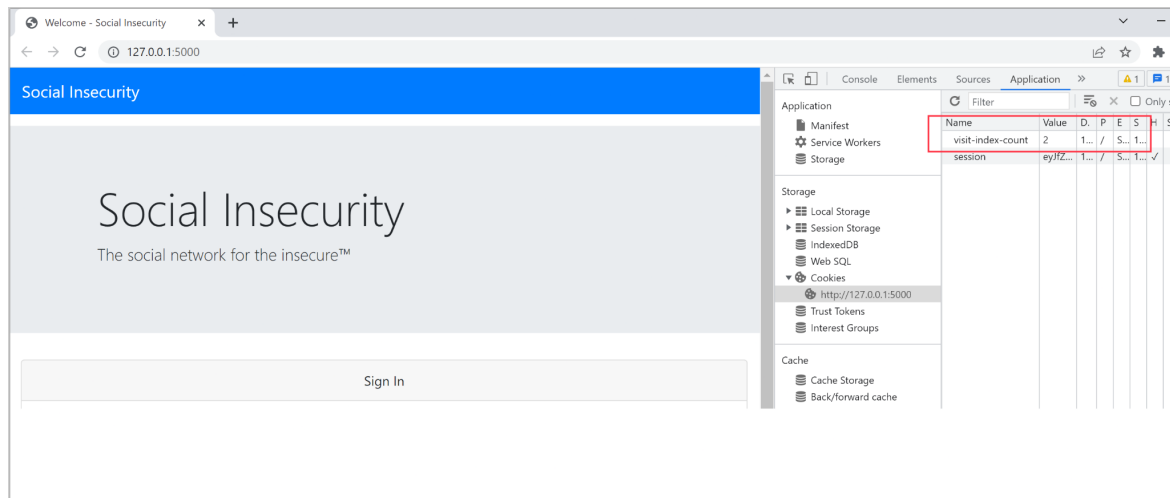
```
# Cookies for index -----
res = make_response(render_template(
    'index.html', title='Welcome', form=form))
count = int(request.cookies.get('visit-index-count', 0))
count += 1
res.set_cookie('visit-index-count', str(count))

return res
```

Source: routes.py

This attachment() is cookies for index. This code has to apply for all index, stream, friends and profile. In the attachment below, on the right corner we can see the visit count increase, everytime someone visits this site, by doing this we can have control over user traffic on the website.

Result



Source(<http://127.0.0.1:5000/>)

CSRF Token

Problem

Cross-Site Request Forgery(CSRF) is an attack where an attacker can perform actions on the behalf of the victim. Browser requests automatically include users credentials therefore if the user is authenticated, the site can't differentiate between a forged request and a legitimate request.

Solution

To determine if the http request is generated by the user we need to use a CSRF token.

```
csrf = CSRFProtect(app)

class Meta:
    csrf = True
    csrf_class = SessionCSRF
    csrf_secret = app.config['SECRET_KEY']
    csrf_time_limit = timedelta(minutes=30)
```

Source: new code of Meta from forms.py

This will be implemented to every class function in forms.py

```
class LoginForm(FlaskForm):
    Meta
    username = StringField('Username', [DataRequired(message="Required field")], render_kw={'placeholder': 'Username'})
    password = PasswordField('Password', [DataRequired(message="Required field")], render_kw={'placeholder': 'Password'})
```

Source: example of Meta code that is used in LoginForm from forms.py

CSP Header

Problem

Web security is based on same-origin policy (SOP), which blocks a website from accessing data outside its origin. This should be enough to ensure security, but the modern web demands sites to incorporate many assets from outside sources like scripts, fonts, styles and other resources from content delivery networks etc. By only using the SOP many functions stopped working in our application.

This vulnerability can be exploited using cross-site scripting (XSS), which is an attack designed to deceive websites trusted by the user into delivering malicious code. With no extra security measures the browser will execute all code from any origin and will not be able to determine which code is authorized.

Solution

Setting up proper CSP directives in HTTP response headers allows us to selectively define which data sources should be allowed in the web application. Too many restrictions can also prevent the intended functionality of the application from working, we had to whitelist scripts and fonts that the application was calling for from outside sources.

File Upload

Problem

In the stream tab, the user has the possibility to post images and text, which is visible to the user and its friends. The users had the ability to upload a file without restrictions, which can be dangerous. letting none image files to get posted can give an attacker the possibility to post some code and try to run it, also no restrictions on file names can be an opening for attackers. No limit on the size of the file will allow for anyone to take up as much storage space as they want, taking up server storage meant for pictures. So the vulnerabilities that need to be solved are the file type, the name of the file and the size.

Solution

By checking for “.” in the file name and splitting at the dot. We can then check if the file is any of the allowed types. By showing an error message the user gets informed why the file is denied. Using “secure_filename” from “werkzeug.utils” which checks the name of the file and changes it to a safe name if needed. By setting a limit to the size of the files the users are no longer able to post bigger files.

Before:

```
# content stream page
@app.route('/stream/<username>', methods=['GET', 'POST'])
def stream(username):
    form = PostForm()
    user = query_db('SELECT * FROM Users WHERE username="{0}"'.format(username), one=True)
    if form.is_submitted():
        if form.image.data:
            path = os.path.join(app.config['UPLOAD_PATH'], form.image.data.filename)
            form.image.data.save(path)

            query_db('INSERT INTO Posts (u_id, content, image, creation_time) VALUES({}, "{0}", "{0}", \'{0}\')'.format(user.id, form.content.data, path, form.image.data.filename))
            return redirect(url_for('stream', username=username))

    posts = query_db('SELECT p.*, u.*, (SELECT COUNT(*) FROM Comments WHERE p_id=p.id) AS cc FROM Posts p JOIN Users u ON p.u_id=u.id')
    return render_template('stream.html', title='Stream', username=username, form=form, posts=posts)
```

Source: routes.py

After:

```
app.config["ALLOWED_IMAGE_EXTENSION"] = ["png", "jpg", "jpeg", "gif"]
@app.route('/stream', methods=['GET', 'POST'])
@login_required
def stream():
    # -- |SESSION|
    session["username"] = current_user.get_id()
    username = session["username"]

    if session["username"] == None:
        return redirect("/index")
    if username != session.get("username"):
        flash("Access denied: Not logged in!")
        return redirect(url_for('index'))

    form = PostForm()
    user = query_db('SELECT * FROM Users WHERE username="{0}"'.format(username), one=True)
    if form.is_submitted():
        if form.image.data:
            if form.image.data.filename == "":
                flash("Image needs name")
                return redirect(url_for('stream', username=username))

            if not "." in form.image.data.filename:
                flash("File type needed")
                return redirect(url_for('stream', username=username))
            else:
                filename = secure_filename(form.image.data.filename)

            ext = filename.rsplit(".", 1)[1]

            if ext.lower() not in app.config["ALLOWED_IMAGE_EXTENSION"]:
                flash("File type is not valid")
                return redirect(url_for('stream', username=username))

            path = os.path.join(
                app.config['UPLOAD_PATH'], filename)
            form.image.data.save(path)
            post_query(user['id'], form.content.data, filename, datetime.now())
            return redirect(url_for('stream'))
        posts = query_db('SELECT p.*, u.*, (SELECT COUNT(*) FROM Comments WHERE p_id=p.id) AS cc FROM Posts AS p JOIN Users AS u ON u.id=p.u_id')

    # -- |COOKIES|
    res = make_response(render_template('stream.html', title='Stream', username=username, form=form, posts=posts))
    count = int(request.cookies.get('visit-stream-count', 0))
    count += 1
    res.set_cookie('visit-stream-count', str(count))
    return res
```

Source: (routes.py)

Social Insecurity Stream Friends

Info: File type is not valid

Error message when trying to upload non-image files

Image (max: 2MB) Choose File No file chosen

```
app.config['MAX_CONTENT_LENGTH'] = 2 * 1024 * 1024
```

Source: <http://127.0.0.1:5000/stream> and `__init__.py`

SQL Injection

Problem

SQL injection is a web security vulnerability that allows an attacker to interfere with the queries that an application makes to its database. It generally allows an attacker to view data that they are not normally able to retrieve. This might include data belonging to other users, or any other data that the application itself is able to access. In many cases, an attacker can modify or delete this data, causing persistent changes to the application's content or behavior.

A successful SQL injection attack can result in unauthorized access to sensitive data, such as passwords, credit card details, or personal user information. Many high-profile data breaches in recent years have been the result of SQL injection attacks, leading to reputational damage and regulatory fines. In some cases, an attacker can obtain a persistent backdoor into an organization's systems, leading to a long-term compromise that can go unnoticed for an extended period.

Solution

A parameterized query is a query in which placeholders are used for parameters and the parameter values are supplied at execution time. The most important reason to use parameterized queries is to avoid SQL injection attacks. Essentially what it does is that instead of concatenating user input with SQL syntax, the query plan gets constructed on the server before the query is executed with parameter values. The only query that gets executed is the insert query. The server accepts the user input and inserts the entire value into the field. It also allows us to handle less malicious scenarios, such as if the user supplies a value like "O'Connor" we no longer need to replace single quotes with double single quotes.

Conclusion

Can always improve the application's security. Need to focus on the more serious issues and work from there. You can never fully remove all flaws, but it can be made adequate for the application's purpose.

Sources

Cookies:

<https://www.javatpoint.com/session-vs-cookies>

Flask_login:

<https://flask-login.readthedocs.io/en/latest/#:~:text=Flask%2DLogin%20provides%20user%20session,log%20them%20in%20and%20out.>

CSRF:

<https://owasp.org/www-community/attacks/csrf>