Security Considerations

Web applications usually face all kinds of security problems and it's very hard to get everything right. Flask tries to solve a few of these things for you, but there are a couple more you have to take care of yourself.

Cross-Site Scripting (XSS)

Cross site scripting is the concept of injecting arbitrary HTML (and with it JavaScript) into the context of a website. To remedy this, developers have to properly escape text so that it cannot include arbitrary HTML tags. For more information on that have a look at the Wikipedia article on Cross-Site Scripting.

Flask configures Jinja2 to automatically escape all values unless explicitly told otherwise. This should rule out all XSS problems caused in templates, but there are still other places where you have to be careful:

- generating HTML without the help of Jinja2
- calling **`Markup`** on data submitted by users
- sending out HTML from uploaded files, never do that, use the `Content-Disposition: attachment` header to prevent that problem.
- sending out textfiles from uploaded files. Some browsers are using content-type guessing based on the first few bytes so users could trick a browser to execute HTML.

Another thing that is very important are unquoted attributes. While Jinja2 can protect you from XSS issues by escaping HTML, there is one thing it cannot protect you from: XSS by attribute injection. To counter this possible attack vector, be sure to always quote your attributes with either double or single quotes when using Jinja expressions in them:

```
<input value="{{ value }}">
```

Why is this necessary? Because if you would not be doing that, an attacker could easily inject custom JavaScript handlers. For example an attacker could inject this piece of HTML+JavaScript:

```
onmouseover=alert(document.cookie)
```

When the user would then move with the mouse over the input, the cookie would be presented to the user in an alert window. But instead of showing the cookie to the user, an attacker might also execute any other JavaScript code. In combination with CSS injections

the attacker might even make the element fill out the entire page so that the user would just have to have the mouse anywhere on the page to trigger the attack.

There is one class of XSS issues that Jinja's escaping does not protect against. The `a` tag's `href` attribute can contain a *javascript:* URI, which the browser will execute when clicked if not secured properly.

```
<a href="{{ value }}">click here</a>
<a href="javascript:alert('unsafe');">click here</a>
```

To prevent this, you'll need to set the Content Security Policy (CSP) response header.

Cross-Site Request Forgery (CSRF)

Another big problem is CSRF. This is a very complex topic and I won't outline it here in detail just mention what it is and how to theoretically prevent it.

If your authentication information is stored in cookies, you have implicit state management. The state of "being logged in" is controlled by a cookie, and that cookie is sent with each request to a page. Unfortunately that includes requests triggered by 3rd party sites. If you don't keep that in mind, some people might be able to trick your application's users with social engineering to do stupid things without them knowing.

Say you have a specific URL that, when you sent `POST` requests to will delete a user's profile (say `http://example.com/user/delete`). If an attacker now creates a page that sends a post request to that page with some JavaScript they just have to trick some users to load that page and their profiles will end up being deleted.

Imagine you were to run Facebook with millions of concurrent users and someone would send out links to images of little kittens. When users would go to that page, their profiles would get deleted while they are looking at images of fluffy cats.

How can you prevent that? Basically for each request that modifies content on the server you would have to either use a one-time token and store that in the cookie **and** also transmit it with the form data. After receiving the data on the server again, you would then have to compare the two tokens and ensure they are equal.

Why does Flask not do that for you? The ideal place for this to happen is the form validation framework, which does not exist in Flask.

JSON Security

In Flask 0.10 and lower, **`jsonify()`** did not serialize top-level arrays to JSON. This was because of a security vulnerability in ECMAScript 4.

ECMAScript 5 closed this vulnerability, so only extremely old browsers are still vulnerable. All of these browsers have other more serious vulnerabilities, so this behavior was changed and **`jsonify()`** now supports serializing arrays.

Security Headers

Browsers recognize various response headers in order to control security. We recommend reviewing each of the headers below for use in your application. The Flask-Talisman extension can be used to manage HTTPS and the security headers for you.

HTTP Strict Transport Security (HSTS)

Tells the browser to convert all HTTP requests to HTTPS, preventing man-in-the-middle (MITM) attacks.

```
response.headers['Strict-Transport-Security'] = 'max-age=31536000; incl
```

- https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security

Content Security Policy (CSP)

Tell the browser where it can load various types of resource from. This header should be used whenever possible, but requires some work to define the correct policy for your site. A very strict policy would be:

```
response.headers['Content-Security-Policy'] = "default-src 'self'"
```

- https://csp.withgoogle.com/docs/index.html
- https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy

X-Content-Type-Options

Forces the browser to honor the response content type instead of trying to detect it, which can be abused to generate a cross-site scripting (XSS) attack.

```
response.headers['X-Content-Type-Options'] = 'nosniff'
```

- https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Content-Type-Options

X-Frame-Options

Prevents external sites from embedding your site in an `iframe`. This prevents a class of attacks where clicks in the outer frame can be translated invisibly to clicks on your page's elements. This is also known as "clickjacking".

```
response.headers['X-Frame-Options'] = 'SAMEORIGIN'
```

- https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Frame-Options

X-XSS-Protection

The browser will try to prevent reflected XSS attacks by not loading the page if the request contains something that looks like JavaScript and the response contains the same data.

```
response.headers['X-XSS-Protection'] = '1; mode=block'
```

- https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-XSS-Protection

Set-Cookie options

These options can be added to a `Set-Cookie` header to improve their security. Flask has configuration options to set these on the session cookie. They can be set on other cookies too.

- `Secure` limits cookies to HTTPS traffic only.
- `HttpOnly` protects the contents of cookies from being read with JavaScript.
- `SameSite` restricts how cookies are sent with requests from external sites. Can be set to `'Lax'` (recommended) or `'Strict'`. `Lax` prevents sending cookies with CSRF-prone requests from external sites, such as submitting a form. `Strict` prevents sending cookies with all external requests, including following regular links.

```
app.config.update(
    SESSION_COOKIE_SECURE=True,
    SESSION_COOKIE_HTTPONLY=True,
    SESSION_COOKIE_SAMESITE='Lax',
)
```

```
response.set_cookie('username', 'flask', secure=True, httponly=True, sa
```

Specifying `Expires` or `Max-Age` options, will remove the cookie after the given time, or the current time plus the age, respectively. If neither option is set, the cookie will be removed when the browser is closed.

```
# cookie expires after 10 minutes
response.set_cookie('snakes', '3', max_age=600)
```

For the session cookie, if **`session.permanent`** is set, then **`PERMANENT_SESSION_LIFETIME`** is used to set the expiration. Flask's default cookie implementation validates that the cryptographic signature is not older than this value. Lowering this value may help mitigate replay attacks, where intercepted cookies can be sent at a later time.

```
app.config.update(
    PERMANENT_SESSION_LIFETIME=600
)

@app.route('/login', methods=['POST'])
def login():
    ...
    session.clear()
    session['user_id'] = user.id
    session.permanent = True
    ...
```

Use **`itsdangerous.TimedSerializer`** to sign and validate other cookie values (or any values that need secure signatures).

- https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies
- https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie

HTTP Public Key Pinning (HPKP)

This tells the browser to authenticate with the server using only the specific certificate key to prevent MITM attacks.

Warning:

Be careful when enabling this, as it is very difficult to undo if you set up or upgrade your key incorrectly.

- https://developer.mozilla.org/en-US/docs/Web/HTTP/Public_Key_Pinning