AppSecAssignment2.1

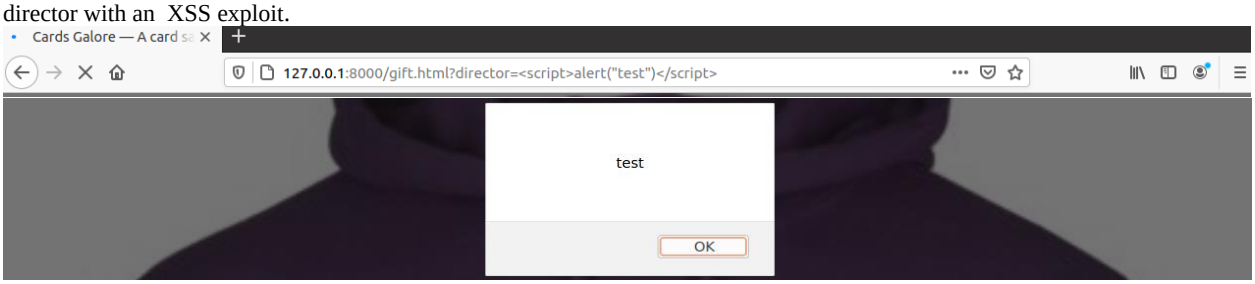
Used the following commands for part 0 to run the web application successful by reading the AppSecAssignment2.1 HW2_Instructions.md.

Attack 1: XSS Attack:

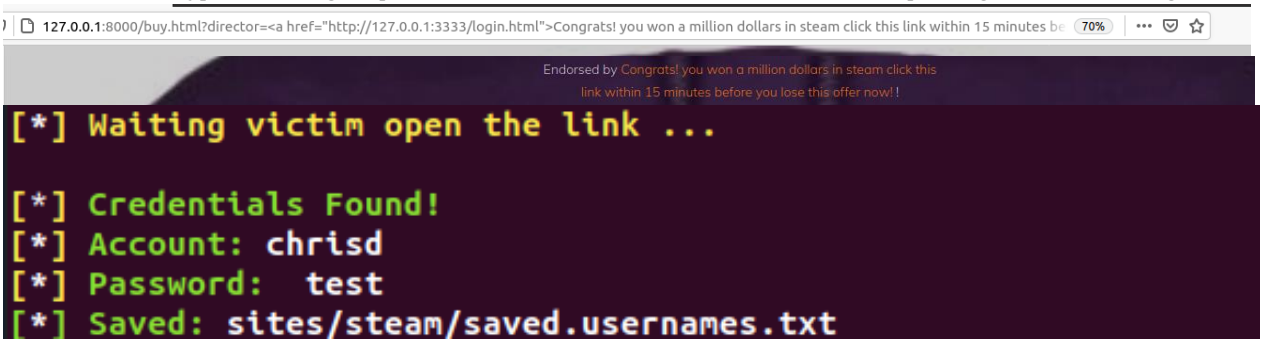
Before looking into the security assignment, I found that there were a lot of interesting cases when making my own web application using "Django Getting Started documentation" and reading "Security with Django", there were a few highlights on vulnerabilities when using Django Web framework. One XSS vulnerabilities it was mentioned that turning off auto escaping code is unsafe which cause a XSS vulnerability. By default, Django auto escapes is turned on to prevent end users to make the browsers render malicious code. To find if auto escape is turned off, look for the keyword safe within the html file within the templated folders. using the grep command was a helpful tool for finding templated code inside the templated folder by typing, grep "{{" *.

It seems that safe word is next to director and located within the file item-single.html that represents the buy URL page and gift html.

The variable director is next to the safe keyword which will be used to exploit the browser. By adding an alert tag by assigning to director with an XSS exploit.



Using this exploit, I was able to use a phishing app call social phish to generate a steam link to steal credential and use the XSS link to create text hyperlink stating the person won a million dollars which will redirect to the phishing site for users to sign in.



To Patch the XSS attack, removing the safe keyword will fix the XSS vulnerability which turns back auto escape on. Another way to fix this vulnerability stated with the documentation was wrapping the template with **{% autoescape on %} {{director}} {% endautoescape %}** for best practice**.**

Patch Test #1 XSS:

After patching the XSS attack, I notice within the html file the output of the XSS is different when it is not patch. Looks like the XSS protection convert the tag into a character entity reference which is used to display reversed characters which would be interpreted as HTML code and non-breaking spaces. The < in a tag converts to an &lt and it will be used for Test cases, reading the Django test cases example looks like you need to call the file test so when you run python manage.py test, it will looks for the test files and run the test cases. For checking the XSS code was patch, the code have to make a request to the URL with the parameters, "directors" and then search for the response contest of &alt to see if the XSS was patch. I had issues doing the method until looking at the view.py code it needed a Product object to functionally run the page. Create an object for product with test values and created a response for the gift page with the director parameter and added the XSS attack. Now to validate the test I check if there is an XSS attack was converter after the get request was initiated using the python function

assertcontains which check if the value of the XSS attack.

```
<p>
  Endorsed by  &lt;script&gt;alert(1)&lt;/script&gt; !
</p>
```

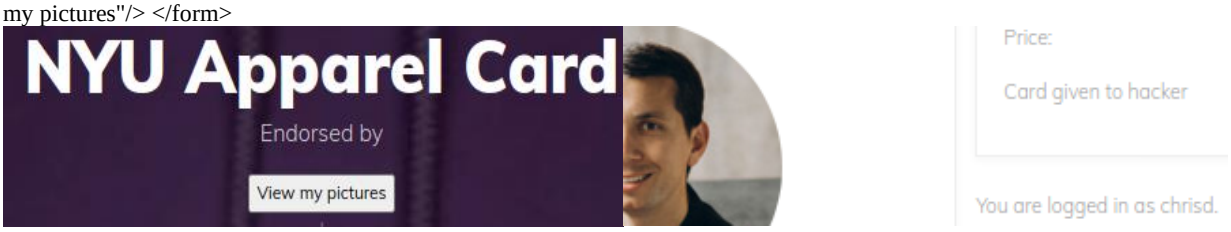
```
<p>
  Endorsed by <script>
    alert(1)
  </script>
```

Attack 2: CSRF Attack:

With this XSS attack, I can create a CSRF attack where if someone clicks a button, I can make them buy a gift card to the user hacker and set the amount of item inside that product. Looking at the OWASP CSRF examples, I copy the templated code and added the proper form action and input type for the CSRF attack to work. To get the proper form action, I had to use burp suit to capture the post request and see what URL it was going to use for the POST request which was http://127.0.0.1:8000/gift/0. Then was able to add the CSRF by setting the value for director that is exploitable to XSS attack.

```
POST /gift/0 HTTP/1.1
Host: 127.0.0.1:8000
Content-Length: 24
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
Origin: http://127.0.0.1:8000
```

http://127.0.0.1:8000/gift.html?director=<form action="http://127.0.0.1:8000/gift/0" method="POST"> <input type="hidden" name="amount" value="2000"/> <input type="hidden" name="username" value="hacker"/><input type="submit" value="View my pictures"/> </form>



The same patch one prevent creating the post request but for add on security I added a csrfmiddlewaretoken within the post request for gift and buy page to prevent hackers to implement the CSRF attack and add the 'django.middleware.csrf.CsrfViewMiddleware' within the Middlware settings.

Patch 2 Test CSRF:

To detect if the CSRF token was generated when the page was rendered, check for pages that has a form pages, since I enable CSRF cookie, the page will output an error if there are any POST request that does not have a CSRF middle token that was generated at time it was rendered. With that I sent a get request to the Gift and search for the csfmiddleware variable.

Attack 3: SQL Injection:

In the Django Security Documentation, in the SQL Section, it was mention for developers to be careful using raw queries in your code which can cause SQL injection attack if its not wrap properly. So, inside the Legacycard folder I used a grep command to search for Raw queries, found that there is two queries being used in the use a gift card page.

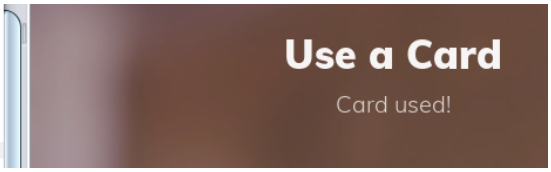
```
root@ubuntu:/home/chrisd/Desktop/ProdCGGY9163HW2.1/GiftcardSite/LegacySite# grep "select" *
grep: migrations: Is a directory
grep: __pycache__: Is a directory
views.py:        card_query = Card.objects.raw('select id from LegacySite_card where data = \'%s\'' % signature)
views.py:        user_cards = Card.objects.raw('select id, count(*) as count from LegacySite_card where LegacySite_card.user_id = %s' % str(request.user.id))
```

Tested how it was used and it spawned up an error stating I can't use a binary value with a type string so inside the file I wrap the variable with a string cast and the post request worked successfully. I capture the time when I need to use the card and send it to a repeater in burp suite to test my injection attacks.

```
Connection: close

------WebKitFormBoundaryKbBOnEfvz7jrOeiQ
Content-Disposition: form-data; name="card_data"; filename="newcard.gftcrd"
Content-Type: application/octet-stream

{"merchant_id": "NYU Apparel Card", "customer_id": "chrisd", "total_value": "101",
"records": [{"record_type": "amount_change", "amount_added": 2000, "signature": "[
insert crypto signature here ]"}]}
```



Looking at the query it wants the value for signature. So, in the signature value I did a comment to spit out a SQL error which I did and try various of SQL injection. One SQL injection that failed when I had to call in another query, the program logged the error that I cannot run more than one query. Researching other attacks, I was able to find a video demonstration on injection cases when SQLite allow one query. Using the union attack with the same columns of the original entry made the application to spit out the admin password hash.

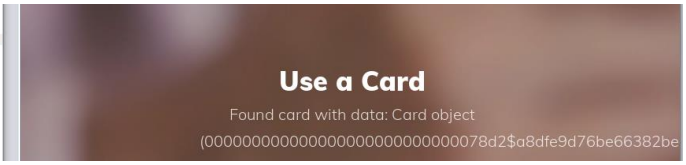
```
Content-Type: application/octet-stream

{"merchant_id": "NYU Apparel Card", "customer_id": "chrisd", "total_value": "3077", "records":
[{"record_type": "amount_change", "amount_added": 2000, "signature": " ' union SELECT password from
LegacySite_user --"}]}

------WebKitFormBoundarykaDjul072DLEcZKq
Content-Disposition: form-data; name="card_supplied"

True
------WebKitFormBoundarykaDjul072DLEcZKq
Content-Disposition: form-data; name="card_fname"

Te
```



To fix the code,

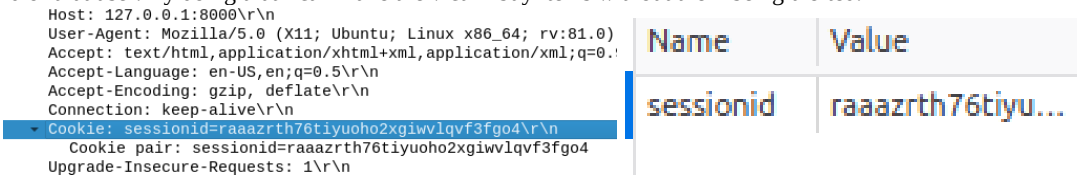
Translated card_query = Card.objects.raw('select id from LegacySite_card where data = \'%s\' % signature) to card_query = Card.objects.filter(data = signature.encode()) using the object relational model.

Attack 3 Test Cases:

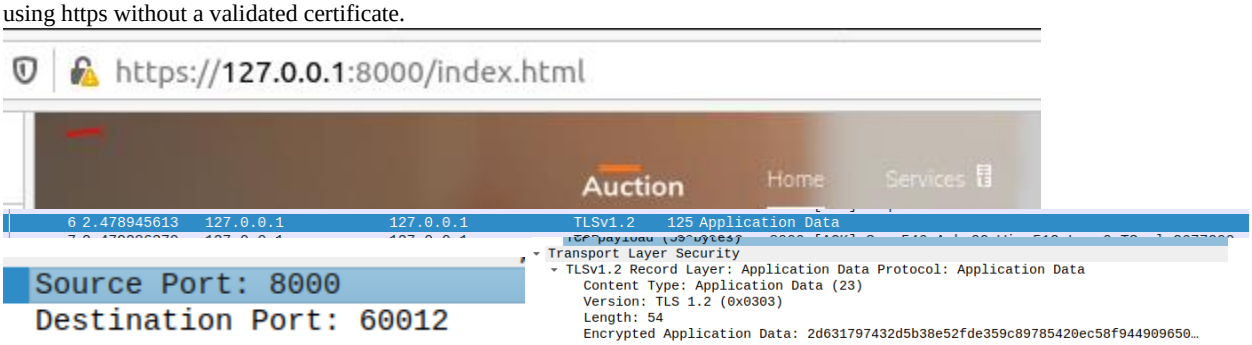
To detect a SQL injection was patch I had to mimic the same post request within the use page when a user submit a gft card. I needed the User object to create a user to the database for SQLi injection to output the database since test cases run on its own separate environment have some data to output. Use IO module to mimic the same response when the application part the gft file. After adding the parameters that needs in the post request like data, filename, card_supplied and fname. Using Burpsuite help finding what variables needed to be added within the Post request. Now test the html if there is not a password being leak within the webpage.

Attack 4: Network Attack HTTPS:

Another vulnerability that explained within the documentation was sites using http. It is possible for existing cookies to be leaked if there is a man in the middle between the client and server, the attack can capture the header information containing the session id of that user. By doing that I can make the victim buy items without them being alerted.



To fix this issue, I made my site HTTPS instead of HTTP. To do that install django-sslserver and add sslserver to the installed app setting to use the command python3 manage.py runsslserver instead of python3 manage runserver. The lock shows that its using https without a validated certificate.



Attack 4 Testcases:

Implementing HTTPS was used to detect if it redirects HTTPS. It seems that I can only choose whether to run http and https, could not find a way to detect if the.

Part 2: Encrypted Database Model

After doing researching and implement ways to encrypt the data value within the Legacy_Card table. I found a cool library that encrypts the database without causing any issue. I installed django-fernet-fields and imported that library in the modely.py. After importing the module, I replaced data = models.BinaryField(unique=True) and used the function for EncryptedTextField to encrypt the data string when giftcard file is created.

NOTE: the test_sqlunion test case fails when I use the EncryptedTextField() since it does not support lookups.

Proper Key Management

One of the methods I did to store my keys is put it in a .env file and let it call through the file as an environment variable. I believe this can be secure if we place the .env file to a private server that only makes calls to the public web application Host IP. This makes the keys that was stored in the source code can be called within separate application. To implement the process, install python-decouple 3.3. To store the keys, create a .env file put a variable name and assigned to your key. After the key is in the. env, we can go to the setting.py file under the Main Giftcard directory. import the module for decouple which is "from decouple import config" and use the config function to call the keys to the .env file. Within a production environment we can host a private key json server with certificate authentication to request the keys that are called from the web application

Travis.yml

Add .travis yml file and stated what type I have environment for that application (linux, python, Django_versions), then I created a requirement.txt to point dependency for the application like the encryption and the decouple for it to work. Lastly, I used the script tag to run the test cases within the GiftCardSite folder