2nd Project Report

Contains 3 sections:

- **Key issues and their fixes** - the *2 x Number of Students* (10 in our case) that we chose and the respective implemented fixes.
- **Implemented Software Features** - the two selected software of the list of three - we chose *password strength evaluation* and *multi-factor authentication (with TOTP)*.
- **Checklist Support Screenshots/Videos** - this section contains any screenshots/videos that are relevant to our **validity** choices while filling the ASVS checklist.

Authors

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

Table of Contents

Key Issues and Their Fixes

As we are 5 students, the 10 issues we identified as most relevant and fixed are the following:

- 2.1.5 - Change password
- 2.1.8 - Password strength meter
- 3.7.1 - Re-authentication or secondary verification before allowing account modifications or sensitive transactions
- 4.1.1 - Application enforces access control rules on a trusted service layer
- 4.1.4 - Principle of Deny by default
- 4.2.2 - Ensure application or framework enforces strong anti-CSRF measures
- 5.2.2 - Sanitize unstructured data to enforce safety measures such as allowed characters and length
- 8.3.2 - Verify users have a method to remove or export their data on demand
- 9.1.1 - Secure TLS for client connectivity
- 11.1.2 - Application will only process business logic flows with all steps being processed in realistic human time.

We've also implemented an extra requirements which didn't require much effort but fixes the following issue: 8.2.1 - Not Caching Sensitive Data.

2.1.5 - Change password

We did not support changing passwords in our original application. This is a significant security issue because password breaches can happen, and users should be able to keep their accounts secure by changing password when that occurs. Therefore, we fixed this issue, and implemented the possibility to change passwords in our web application. While implementing 2.1.5, we also complied with 2.1.6 (password change requires current and new password)

We implemented this by creating a new endpoint at `/accounts/profile/changepassword` which can be accessed by clicking the **Change password** button in the edit profile page (`/accounts/profile/edit`) or directly accessing the changepassword endpoint. The endpoint code is the following:

app_sec/accounts/views.py line 182

```
@never_cache
@ratelimit(key='ip', rate='10/m', block=True)
@verified_required
def change_password(request):
    if request.session.get("valid_otp_accounts") != True:
        request.session["redirect_to"] = "accounts:change_password"
        return redirect('accounts:reauth_2s')
    if request.method == "POST":
        form = ChangePasswordForm(request.POST)
        if form.is_valid():
            user = request.user
            data = form.cleaned_data
            if not request.user or not authenticate(request, email=user.email, password=data.get("current_password")):
                messages.error(
                    request, "Current password doesn't match.", "danger"
                )
                context = {"title": "Change Password", "form": form}
                return render(request, "change_password.html", context)

            user.set_password(data.get("new_password"))
            user.save()
            messages.success(request, "Your password has been successfully updated", "success")
            del request.session["valid_otp_accounts"]
            return redirect("accounts:change_password")
    else:
        form = ChangePasswordForm()

    context = {"title": "Change Password", "form": form}
    return render(request, "change_password.html", context)
```

If it's a *GET* request it renders a page with the following form for changing passwords:

app_sec/accounts/forms.py line 64

```
class ChangePasswordForm(forms.Form):
    current_password = forms.CharField(
        widget=forms.PasswordInput(attrs={"class": "form-control", "placeholder": "current password"})
    )
    new_password = forms.CharField(
        widget=forms.PasswordInput(
            attrs={"class": "form-control", "placeholder": "new password", "id": "passwordInput"},
        ),
        validators=[validate_password, validate_breached_password]
    )
    confirm_new_password = forms.CharField(
        widget=forms.PasswordInput(
            attrs={"class": "form-control", "placeholder": "new password", "id": "confirmPasswordInput"},
        ),
        validators=[validate_password, validate_breached_password]
    )

    def clean(self):
        cleaned_data = super().clean()

        validate_match_password(cleaned_data.get("new_password"), cleaned_data.get("confirm_new_password"))

        validate_new_password_diff(cleaned_data.get("new_password"), cleaned_data.get("current_password"))
```

First, Django validates the fields with the `validate_password` and `validate_breached_password` validators: the first one checks for password length as defined by ASVS 2.1.1 and 2.1.2 and the second one cross-checks the password against the *Have I Been Pwned* passwords API.

Second, the `clean()` function is ran, which validates if `new_password` matches `confirm_new_password` and also validates if `new_password` is different from

`current_password`.

Lastly, the view code runs, which checks if `current_password` is the user's current password by trying to authenticate him with `authenticate()`.

If all these validations pass, the user successfully changes his password.

Demonstration:

https://github.com/detiuaveiro/2nd-project-group_04/assets/97046574/2d3e7351-a459-4b0d-b1ef-b0287321b7d0

2.1.8 - Password strength meter

In the last project we defined some requirements that had to be met for the password to be considered valid (which are the same criteria that we are going to talk about next). However this go against ASVS 2.1.9 (Verify that there are no password composition rules limiting the type of characters permitted. There should be no requirement for upper or lower case or numbers or special characters.). We couldn't just remove this criteria completely as the user needs to be educated relative to what constitutes a strong password and should be encouraged to create stronger and more complex passwords. In this new app, we implemented a password strength meter and added hints so that the users can have a visual representation of how strong their password is and how to make it stronger.

The strength criteria are the following:

- includes a special character
- includes an uppercase letter
- includes a lowercase letter
- includes a number
- doesn't include user's email or full name in the password

app_sec/accounts/templates/password_meter.html line 6

```
 meter = document.getElementById("passwordMeterRange");
password = document.getElementById("passwordInput");
email = document.getElementById("id_email");
full_name = document.getElementById("id_full_name");
specialCharRegex = new RegExp("[!@#$%^&*()+\\[\\]{}|;:,.<>?/`~]");
digitRegex = new RegExp("\\d");
lowercaseRegex = new RegExp("\\p{Ll}","u");
uppercaseRegex = new RegExp("\\p{Lu}","u");
var passwordTips = document.querySelector('.password-tips');

password.oninput = function() {
    let strength = 0;
    let full_name_value = "{{ request.user.full_name }}"
    let email_value = "{{ request.user.email }}"
    if (email_value == "")
        email_value = email.value
    if (full_name_value == "")
        full_name_value = full_name.value
    let password_value = password.value
    const tips = generatePasswordTips(password_value, email_value, full_name_value);
    displayPasswordTips(tips);

    if (12 <= password_value.length && password_value.length <= 128)
        strength++
    if (specialCharRegex.test(password_value))
        strength++
    if (digitRegex.test(password_value))
        strength++
    if (lowercaseRegex.test(password_value))
        strength++
    if (uppercaseRegex.test(password_value))
        strength++
    if (email_value && email_value.includes("@") && password_value.includes(email_value.split("@")[0]))
        strength--
    if (full_name_value && password_value.includes(full_name_value))
        strength--

    meter.value = strength
}

function generatePasswordTips(password, email_value, full_name_value) {
    var tips = [];

    if (password.length < 1) {
```

```
        return tips;
    }

    // Example tips
    if (!specialCharRegex.test(password)) {
        tips.push('Include a special character !@#$%^&*()+[]{}|;:,.<>?/`~');
    }

    if (!digitRegex.test(password)) {
        tips.push('Include a number');
    }

    if (!uppercaseRegex.test(password)) {
        tips.push('Include an uppercase letter');
    }

    if (!lowercaseRegex.test(password)) {
        tips.push('Include a lowercase letter');
    }

    if (password.length < 12) {
        tips.push('Use at least 12 characters');
    }

    if (password.length > 128) {
        tips.push('Use at most 128 characters');
    }

    if (email_value && email_value.includes("@") && password.includes(email_value.split("@")[0])) {
        tips.push('Do not include your email');
    }

    if (full_name_value && password.includes(full_name_value)) {
        tips.push('Do not include your name');
    }

    return tips;
}

function displayPasswordTips(tips) {
    passwordTips.innerHTML = '<p>' + tips.join('<br>') + '</p>';
}
```

The password strength meter provides users with a visual representation of the password's strength, determined by the current value of the "strength" variable. This real-time feedback helps users create secure passwords by adhering to recommended strength criteria.

The meter is in the **Sign-up** and **Change password** pages, where new passwords are created.

Demonstration:

https://github.com/detiuaveiro/2nd-project-group_04/assets/97046574/938d2235-da96-47f3-8d66-61ca2ceaa995

3.7.1 - Re-authentication or secondary verification before allowing account modifications or sensitive transactions

This was not supported originally in our application. Users should be prompted for re-authentication when performing sensitive transactions or account modifications. This is just an extra validation of a user's session in crucial moments. It prevents the possibility of an attacker gaining access to a session without authentication and performing sensitive operations while unauthorized. Furthermore, as we decided to implement a TOTP authentication login it made sense for us to just prompt the user to verify another TOTP token to conclude that he is in fact the legitimate user.

Implementation

In our application, we decided to prompt users for re-authentication when they try to *checkout* a (mock) purchase and when they try to make account modifications such as changing full name/email or password.

We changed the logic that renders those pages in the following way:

- When a user tries to render the page, redirect him to a 2-step verification page, which prompts for a TOTP from the assigned authenticator.
- If user successfully passes the TOTP verification, redirect him back to the protected page. Now the user can make changes/do sensitive operations

This snippet of code, is what runs when a user accesses `localhost:8080/accounts/profile/changepassword`:

- it initially checks if the user has the `valid_otp_accounts` variable set to `True` in his session data (session data is safe and cannot be altered, since we use server-side session data - check this for more information, we use database-backed sessions)
- if the user does not have that variable set to true, it sets the `redirect_to` variable to `accounts:change_password`, which, in Django, points to the changepassword endpoint. And finally redirects the user to the 2 step verification.

app_sec/accounts/views.py line 182

```
def change_password(request):
    if request.session.get("valid_otp_accounts") != True:
        request.session["redirect_to"] = "accounts:change_password"
        return redirect('accounts:reauth_2s')
    if request.method == "POST":
        form = ChangePasswordForm(request.POST)
        if form.is_valid():
            user = request.user
            data = form.cleaned_data
            if not request.user or not authenticate(request, email=user.email, password=data.get("current_password")):
                messages.error(
                    request, "Current password doesn't match.", "danger"
                )
                context = {"title": "Change Password", "form": form}
                return render(request, "change_password.html", context)

            user.set_password(data.get("new_password"))
            user.save()
            messages.success(request, "Your password has been successfully updated", "success")
            del request.session["valid_otp_accounts"]
            return redirect("accounts:change_password")
    else:
        form = ChangePasswordForm()

    context = {"title": "Change Password", "form": form}
    return render(request, "change_password.html", context)
```

After the user is redirected to the `localhost:8000/accounts/reauth/2step`, this code is ran:

app_sec/accounts/views.py line 133

```
@verified_required
def otp_view_verify(request):
    context = {}
    if request.method == "POST":
        form = MyOTPTokenForm(user=request.user, data=request.POST)
        if form.is_valid():
            redirect_to = request.session.get("redirect_to")
            if redirect_to.startswith("accounts:"):
                request.session["valid_otp_accounts"] = True
            else:
                request.session["valid_otp_sensitive"] = True
            request.session["valid_otp"] = True
            del request.session["redirect_to"]
            return redirect(redirect_to)
    else:
        form = MyOTPTokenForm(user=request.user)
    context['form'] = form
    return render(request, 'login2s.html', context)
```

The most important part starts at `if form.is_valid()`, which checks if the given OTP is valid or not.

First off, we get the `redirect_to` session variable, to know where to redirect the user after successfully verifying his token. Next, we set the appropriate session variable to `True` (valid_otp_accounts or valid_otp_sensitive, first one is related to account modifications and last one related to other sensitive operations) and finally, redirect the user back to the appropriate page.

It is also important to mention that, for **additional security** the user is prompted 2 step verification every time he enters the page.

This is done using some middleware that intercepts any endpoint call:

- checks if the URL called is not one of the URLs protected by 2 step verification
- if it's not one of those, invalidates the `valid_otp_accounts` / `valid_otp_sensitive`, so users have to re-authenticate again if they try to enter protected pages.

app_sec/online_shop/middleware.py line 13

```
 # URLs protected behind 2FA with TOTP
protected_urls = [
    reverse('orders:create_order'),
    reverse('accounts:edit_profile'),
    reverse('accounts:change_password')
]

# delete session variables associated to OTP verification
# when going to URLs that don't require OTP verification
if request.path not in protected_urls and not request.path.startswith("/static"):
    if "valid_otp_accounts" in request.session:
        del request.session["valid_otp_accounts"]
    if "valid_otp_sensitive" in request.session:
        del request.session["valid_otp_sensitive"]
return self.get_response(request)
```

The endpoints protected by re-authentication:

- Change full name/email (check `edit_profile(request)` @ line 134) or password (check `change_password(request)` @ line 155)
- Checkout current cart (check `create_order(request)` @ line 22)

4.1.1 - Application enforces access control rules on a trusted service layer

Our last project complied with this requirements almost entirely. By using `@login_required` and `@user_passes_test` on the views created at views.py we enforced access control rules on the specified view. However in django webapps in advised not to have DEBUG=True in deployment. Despite not having deployed the app we tried to follow this requirement. With the intent of setting DEBUG to False we introduced a little bug where a user could access invoices of other users through the path to the specific files. By inspecting our HTML code the user could understand that images and possibly other files were being served at the /media endpoint. Of course the user would have to guess the path and name of files because we don't disclose that (e.g. /media/invoices/7.txt) but is a issue we needed to address nevertheless. By having to explicitely serve each folder of media folder we prevent data leaks that could jeopardise our user's privacy which is a major security requirement.

- **Previous Version** - Here you can see that all media and static files are served with no restrictions regarding permissions and thereby not enforcing access control.

```
urlpatterns = [
    path('', views.home_page, name='home_page'),
    path('<slug:slug>', views.product_detail, name='product_detail'),
    path('add/favorites/<int:product_id>/', views.add_to_favorites, name='add_to_favorites'),
    path('remove/favorites/<int:product_id>/', views.remove_from_favorites, name='remove_from_favorites'),
    path('favorites/', views.favorites, name='favorites'),
    path('search/', views.search, name='search'),
    path('filter/<slug:slug>/', views.filter_by_category, name='filter_by_category'),
    re_path(r'^media/(?P<path>.*)$', serve,{'document_root': settings.MEDIA_ROOT}),
    re_path(r'^static/(?P<path>.*)$', serve,{'document_root': settings.STATIC_ROOT})
]
```

- **Fixed Version** - in this version we served all files that should be public to all users. We serve the static files and we serve media files corresponding to product and review images (as these two should be viewed by all users). The third folder `invoices/` that was present in the media folder was not served as we didn't need to access any of its files in our code and we downloaded the invoices at a specific endpoint (`orders/downloads`) where access control was enforced and the corresponding invoice file was downloaded manually.

```
urlpatterns += [
    # serve all static files
    re_path(r'^static/(?P<path>.*)$', serve, {'document_root': settings.STATIC_ROOT}),

    # serve only public media files
    # serve all product files (images of each product)
    re_path(r'^media/products/(?P<path>.*)$', serve, {'document_root': os.path.join(settings.MEDIA_ROOT, 'products')}),
    # serve all review images (reviews are made so everyone can see the review)
    re_path(r'^media/review_images/(?P<path>.*)$', serve, {'document_root': os.path.join(settings.MEDIA_ROOT, 'review_images')}),
    # no need to serve invoices because they are downloaded manually at a specific endpoint
    # at that endpoint we only allow the corresponding user to download the invoice
]
```

4.1.4 - Principle of Deny by default

The Django framework (unlike Django Rest Framework) doesn't have a built-in way to deny access to all views by default. By using `@login_required` and `@user_passes_test` on the views created at views.py we enforced access control rules on that specified view. However if there was no decorator on the view and the endpoint was registered at `urls.py` the view could be accessed by any user. The inclusion of this decorators could easily be forgotten during development resulting in endpoints open to the public. By having to specify in the allow/white list each view we can prevent this from happening.

In order to enforce access control rules within our system according to the Principle of Deny by default, we have implemented a set of decorators, each serving a specific purpose. These decorators are `@no_access`, `@no_auth_required`, `@my_login_required`, `@verified_required`, `@not_verified_required`, and `@manager_required` which enable us to meticulously manage and regulate the access levels of different user types to specific views. The implementation of these decorators follow the same flow as django's current decorators (e.g. django decorators implementation) with some tweaks that allow to deny all access to views by default. The main decorator that's going to make everything possible is the `@no_access` decorator:

- The main objective here was to encapsulate all views with these decorator which is done in the main urls.py file. Here we loop through any defined path and wrap each view with `no_access`

```
urlpatterns = [
    path('', include('shop.urls', namespace='shop')),
    path('accounts/', include('accounts.urls', namespace='accounts')),
    path('cart/', include('cart.urls', namespace='cart')),
    path('orders/', include('orders.urls', namespace='orders')),
    path('dashboard/', include('dashboard.urls', namespace='dashboard')),
]

for pat in urlpatterns:
    # print("PAT: ", pat)
    if isinstance(pat, URLPattern):
        pat.callback = no_access(pat.callback)
    else:
        for subpat in pat.url_patterns:
            # print(subpat)
            # no nested subpatterns
            subpat.callback = no_access(subpat.callback)
```

- If on a specific view, in addition to this decorator there is one more decorator of the previous list the evaluation function `lambda u: getattr(function, WRAPPED_BY_AUTH, False)` will return true and the next decorator in line will evaluate and control the access to the view. Bear in mind this will only happen if the other decorator is one of `@no_auth_required`, `@my_login_required`, `@verified_required`, `@not_verified_required` or `@manager_required`. This flow is achieved by setting WRAPPED_BY_AUTH to true in these views and accessing it in `no_access`.

```
WRAPPED_BY_AUTH = "is_wrapped_by_auth"

def no_access(
        function=None, redirect_field_name=REDIRECT_FIELD_NAME, login_url=None
):
    """
    Decorator that denies access to all views by default.
    This decorator will delegate authentication to the following decorators if they are specified:
    - no_auth_required
    - my_login_required
    - verified_required
    - not_verified_required
    - manager_required
    """
    if function:
        # print(getattr(function, WRAPPED_BY_AUTH, False))
        actual_decorator = user_passes_test(
            lambda u: getattr(function, WRAPPED_BY_AUTH, False),
            login_url=login_url,
            redirect_field_name=redirect_field_name,
        )
        return actual_decorator(function)

    actual_decorator = user_passes_test(
        lambda u: False,
        login_url=login_url,
        redirect_field_name=redirect_field_name,
    )
    return actual_decorator
```

- `@my_login_required` - As you can see in these view we set the WRAPPED_BY_AUTH to True to be access by `no_access` and, consequentely, delegate the

access control to this decorator.

```
def my_login_required(
        function=None, redirect_field_name=REDIRECT_FIELD_NAME, login_url=None
):
    """
    Decorator for views that checks that the user is "logged in" (without 2FA - without being verified), redirecting
    to the log-in page if necessary.
    Already verified users cannot access the views marked with my_login_required.
    Actually the only view marked with my_login_required is otp_view() and we don't want verified users accessing it
    because it resets the session cookie expiry.
    """
    actual_decorator = user_passes_test(
        lambda u: u.is_authenticated and not u.is_verified(),
        login_url=login_url,
        redirect_field_name=redirect_field_name,
    )
    if function:
        setattr(function, WRAPPED_BY_AUTH, True)
        return actual_decorator(function)
    return actual_decorator
```

Before referencing the list of all implemented decorators is important to mention that in our application a current user can be in 3 different states.

- **anonymous**
- **authenticated** which can be verified through `is_authenticated` attribute
- **authenticated + verified** which can be verified through `is_verified` function

Here is the actual list of all implemented decorators and the authorization they set:

- **no_access** - denies access to all views by default (unless one of the following decorators is specified);
- **no_auth_required** - explicitly allows access to any user;
- **my_login_required** - explicitly allows access to any authenticated user (but not verified);
- **verified_required** - explicitly allows access to any verified user;
- **not_verified_required** - explicitly allows access to any user which is not verified;
- **manager_required** - explicitly allows access to any verified user that's the manager;

4.2.2 - Ensure application or framework enforces strong anti-CSRF measures

Despite commanding efforts to implement this in the last project there were some places where we didn't respect this requirement. We used Django's Template Tag `{% csrf_token %}` in multiple forms and now after testing with OWASP ZAP tool we realized that some forms didnt had it.

In this version the issue has been fixed. This token is sent together with the POST request when a form is submitted and the server checks if it is valid. This is useful so that attackers cannot make cross-site requests from their bad websites to our endpoints, since guessing the token is ridiculously unlikely.

To prevent any bad requests from attackers to retrieve our website's data, we added the following:

app_sec/shop/templates/base.html

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

5.2.2 - Sanitize unstructured data to enforce safety measures such as allowed characters and length

The original version of our application had good XSS/SQL injection prevention, but we missed an important place with unstructured data - **reviews**.

Users were able to input reviews of great length (tested with 1MB long text), since we did not limit it. This could potentially cause a easy to perform DoS attack - simply spamming very long reviews, the database write throughput would be significantly lowered.

To fix it, we added the following code:

- limited the *review* text field length to 2048 characters.

app_sec/shop/forms.py line 9

```
review = forms.CharField(label='',widget=forms.Textarea(
        attrs={'class': 'form-control', 'id': 'reviewProduct', 'rows': '4',
               'placeholder': "Your opinion on the product"}
    ),
    max_length=2048)
```

- added the following verification to `clean()` data (which is called when calling `form.is_valid()`) as a second verification mechanism:

app_sec/shop/forms.py line 24

```
if len(comment) > 2048:
    raise forms.ValidationError("Comment must be 2048 characters long or less.")
```

It's important to remember that users cannot fabricate a POST request to circumvent these requirements, since a *csrf token* is used in the form.

Other locations where users could submit unlimited length data were: `UserRegistrationForm`, specifically in the `email` and `full_name` fields; `UserLoginForm` on the field `email`; `ManagerLoginForm` on the field `email`.

The same fix was done:

- added max length for all fields, of 100 characters (`max_length=100`) and `validators` for those fields.

app_sec/accounts/forms.py

```
class UserLoginForm(forms.Form):
    email = forms.EmailField(
        widget=forms.EmailInput(attrs={"class": "form-control", "placeholder": "email"}),
        max_length=100,
        validators=[validate_email_length]
    )
    password = forms.CharField(
        widget=forms.PasswordInput(
            attrs={
                "class": "form-control",
                "placeholder": "password",
                "id": "passwordInput",
            }
        )
    )


class UserRegistrationForm(forms.Form):
    email = forms.EmailField(
        widget=forms.EmailInput(attrs={"class": "form-control", "placeholder": "email"}),
        max_length=100,
        validators=[validate_email_length]
    )
    full_name = forms.CharField(
        widget=forms.TextInput(
            attrs={"class": "form-control", "placeholder": "full name"}
        ),
        max_length=100,
        validators=[validate_full_name_length]
    )
    password = forms.CharField(
        widget=forms.PasswordInput(
            attrs={"class": "form-control", "placeholder": "password", "id": "passwordInput"},
        ),
        validators=[validate_password, validate_breached_password]
    )


class ManagerLoginForm(forms.Form):
    email = forms.EmailField(
        widget=forms.EmailInput(attrs={"class": "form-control", "placeholder": "email"}),
        max_length=100,
        validators=[validate_email_length]
    )
    password = forms.CharField(
        widget=forms.PasswordInput(
            attrs={"class": "form-control", "placeholder": "password", "id": "passwordInput"}
        )
    )
```

app_sec/accounts/validators.py line 33

```
def validate_email_length(value):
    if len(value) > 100:
        raise ValidationError("Email must be at most 100 characters long.")


def validate_full_name_length(value):
    if len(value) > 100:
        raise ValidationError("Full name must be at most 100 characters long.")
```

8.3.2 - Verify users have a method to remove or export their data on demand

In our original application, users could not remove their data from our application. Once a user had created an accounts or made a review of a product, they could not

delete it, for example. This is crucial in the process of respecting user privacy and providing them with control over their data.

To fix this, we added a button to the user profile page that allows the user to delete his account. When the user clicks the button, all his data is deleted from the database including his reviews, images, order invoices, etc. A button to delete a single review was also added to the review page.

Implementation

```
# accounts/views.py
def delete_account(request):
    reviews = request.user.review_set.all()
    for review in reviews:
        if review.user_review_image:
            review.user_review_image.delete()
    orders = request.user.orders.all()
    for order in orders:
        order_id = order.id
        file_name = f"{order_id}.txt"
        file_path = os.path.join(settings.MEDIA_ROOT, "invoices", file_name)
        if os.path.exists(file_path):
            os.remove(file_path)

    request.user.delete()
    messages.success(request, "Your account has been successfully deleted", "success")
    return redirect("accounts:user_login")
```

9.1.1 - Secure TLS for client connectivity

Previously, our application did not support HTTPS.

That meant that all the data sent between the client and the server was not encrypted, and could be intercepted by an attacker. Since our application is running locally and we do not own a domain, we implemented HTTPS using a **self-signed certificate**. We used a tool named `mkcert` to generate the certificate - this tool generates a root CA certificate and a certificate for localhost signed by the root CA. Then we made our system trust the root CA certificate, so that the browser would trust the certificate for localhost.

Implementation

To generate the certificates, we ran the following commands:

```
# Make our system trust the root CA certificate
mkcert -install

# Generating the root CA certificate
mkcert -cert-file cert.pem -key-file key.pem localhost 127.0.0.1
```

Since the default django command "manage.py runserver" does not support HTTPS, we had to use another one which is part of the excellent Django Extensions package:

```
python3 manage.py runserver_plus --cert-file cert.pem --key-file key.pem
```

Now we can access our application using HTTPS on localhost.

11.1.2 - Application will only process business logic flows with all steps being processed in realistic human time

Our original application only had rate limiting protection for unsuccessful login attempts in the login form - limiting users to 5 attempts every 10 minutes. (line 145 in this file). But other business logic flows were not limited (e.g. items could be added to a user's cart at a rapid pace and checkout). This could result in a Denial of Service (DoS) attack through rapid and automated actions, overwhelming the system's capacity and causing service disruption for legitimate users.

To prevent this, we set a limit of 15 requests per minute for each IP address (this number may vary depending on the view). This means that if an IP address tries to make more than 15 requests in a minute, it will be blocked for 1 minute. This is a good protection against this type of attacks, since it is very unlikely that a normal user will make more than 15 requests in a minute.

Implementation:

In each view, we used `django-ratelimit` to limit endpoints requests by user IP.

Example:

```
@ratelimit(key='ip', rate='20/m', block=True)
@no_auth_required
def home_page(request):
    products = Product.objects.filter(quantity__gte=1)
    context = get_ordered_products_context(request, products)
    return render(request, 'home_page.html', context)
```

The home page (`localhost:8000/`) will be limited to 20 requests per minute.

Here's a demonstration of rate limiting in the website's home page. The server returns 403 Forbidden after 20 requests within a time window:

https://github.com/detiuaveiro/2nd-project-group_04/assets/97046574/7742dfdc-f592-4c38-bd09-ffe8410e5205

8.2.1 - Anti-Caching Headers for Sensitive Data

Usually, browsers cache the pages that the user visits, to make the navigation faster.

To prevent the browser from caching sensitive data, we added the anti-caching headers to those pages, so that the browser does not cache them.

Implementation

In the views that contain sensitive data, we added the following header:

```
@never_cache
```

Implemented Software Features

Multi-Factor Authentication - TOTP authentication login

Implemented with the package django-otp and complies with the L1 requirements of ASVS V2.8

- 2.8.1 - defined lifetime before expiring at step attribute of TOTP algorithm (follows implementation of the TOTP algorithm from RFC 6238)

Regarding the implementation we wanted to maintain our authentication login and the related required verifications because it was related to other issues related to this project. Because of that we implemented a form to verify the TOTP tokens by inheriting `django-otp's OTPTokenForm` . `MyOTPTokenForm` behaves differently from the previous form regarding the following aspects:

- Becase we decided that our implementation would only allow the existance of one device per user, if the user doesn't have an existing device it's created automatically. This one device is previously selected in the constructor (`OTPTokenForm` allowed for multiple devices per user) and hidden from the user. Bear in mind that this device is created with confirmed attribute set to False until the user later uses it to validate a TOTP token, proving they have the secret key. The idea is to prevent locking a user out of their account if they generate a secret key but fail to set it up correctly;
- Changed the labels (custom messages) that would appear to the user;
- Deleted otp_challenge field as it was not required and didn't add any functionality to the form as we used TOTPDevices. As we can see in the docs OTP devices come in two general flavors: passive and interactive. A passive device is one that can accept a token from the user and verify it with no preparation. Examples include devices corresponding to dedicated hardware or smartphone apps that generate sequenced or time-based tokens (in our case google authenticator). An interactive device needs to communicate something to the user before it can accept a token. Two common types are devices that use a challenge-response OTP algorithm and devices that deliver a token to the user through an independent channel, such as SMS. With this in mind if we take a quick look at the source code of the django-otp form that is the base class of OTPTokenForm the `_handle_challenge function` would be called regarding the `otp_challenge` field if it did in fact exist. In turn the `_handle_challenge function` uses the `generate_challenge function` which will return None unless overridden (here a stub is also set to True). It is in fact overriden for example in EmailDevice but not for TOTPDevice. We can also see that what classifies an interactive device in the Device base class is not having the previously mentioned stub set. Now if we look into the `_handle_challenge function` implementation again we can see that when the returned challenge is None a validation error is raised saying the selected OTP device isn't interactive (which we already knew too) revealing `otp_challenge` as a useless attribute for our usecase.

```
class MyOTPTokenForm(OTPTokenForm):
    # otp error messages from OTPTokenForm are fine
    # no need to make custom error messages

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        if not self.fields['otp_device'].choices:
            # if device doesn't exist it is created automatically with confirmed=false (only 1 device per user)
            # confirmed=false until the user later uses it to validate a TOTP token (proving he/she has the secret key)
            # The idea is to prevent locking a user out of the account if he generates a secret key but fails the setup
            device, created = TOTPDevice.objects.get_or_create(user=self.user, name="1st TOTP Device", confirmed=False)
            self.fields['otp_device'].choices = [(f"otp_totp.totpdevice/{device.id}", device.name)]

        # Set default value for device (the one device that exists per user)
        self.fields['otp_device'].initial = self.fields['otp_device'].choices[0][0]
        # Modify labels
        self.fields['otp_device'].label = 'Chose Your Device'
        self.fields['otp_token'].label = 'Enter the Verification Code'
        # Hide device because there is only one per user
        self.fields['otp_device'].widget = HiddenInput()
        # remove challenge attribute
        del self.fields['otp_challenge']
```

Before starting to explain the full login process is important to mention that in our application a current user can be in 3 different states.

- **anonymous**
- **authenticated** which can be verified through `is_authenticated` attribute
- **authenticated + verified** which can be verified through `is_verified` function

According to the last 3 states a user it would be perfectly normal to assume that there are some features that aren't available to an anonymous user but are available to an authenticated user. However, that's not true and these 3 states exists just because we have two different forms/pages in the login process to verify credentials and to verify the TOTP token. The full login process includes the following steps:

1. A login view responsible for verifying the credentials of the user and logging him in. There are some topics that should be pointed out:
    - `auth_login` sets the user to the **authenticated** state (therefore `is_authenticated` will return true and `is_verified` false). An **authenticated** user only gains one additional permission than an **anonymous** one. An **authenticated** user will pass the tests enforced by `@my_login_required` at the `otp_view` which corresponds to the second step of out authentication process;
    - an expiry of 120 seconds is set - this means that after the user has validated his credentials he has two minutes to verify his identity by providing a TOTP token (we'll look into that in the next topic). If we've not done this the session time would be the django's default which is equivalent to 2 weeks and the user would not be required to validate it's credentials in that period of time (which would pose as a security risk);
    - the password is checked for breaches and this is saved to inform the user of this when he logs in. It's also important to mention that django sessions saved data are stored server side and the corresponding session cookie only contains the session id - not the data itself.

```
from django.contrib.auth import login as auth_login

@ratelimit(key='ip', rate='15/m', block=True)
@not_verified_required
def user_login(request):
    if request.method == 'POST':
        form = UserLoginForm(request.POST)
        if form.is_valid():
            data = form.cleaned_data
            user = authenticate(
                request, email=data['email'], password=data['password']
            )
            if user is not None:
                # user "logged in" - still requires 2FA to be verified and access the rest of the webapp
                auth_login(request, user)
                request.session.set_expiry(120)
                request.session['haveBeenPwnd'] = check_for_breach(data['password'])
                return redirect('accounts:user_login_2s')
                # unsafe password - redirect user to change password with error message
            else:
                messages.error(
                    request, 'Email or password is invalid.', 'danger'
                )
    else:
        form = UserLoginForm()
    context = {'title': 'Login', 'form': form}
    return render(request, 'login.html', context)
```

2. The next view is responsible for the TOTP authentication which is divided in the following steps:
    - If the request is not a POST than `MyOTPTokenForm` is prompted for the current user (creates an unconfirmed device if that's the first time);
    - If the request is indeed a POST `MyOTPTokenForm` is used to verify the validity of the token. If the token is valid (and consequently the form) we get the corresponding TOTP device and we set the confirmed attribute to True because the user has now proved that has the secret key and the setup process was successful. After that `otp_login(request, device)` sets the user state to **verified (authenticated + verified)** which means the user can now access all intended application features. Additionally we set the session expiry to django's default (2 weeks worth of seconds) overriding the 2 minute session previously defined (the user has successfuly logged in under 2 minutes). If the token is not valid the form variable carries the errors to be displayed to the user. This errors, in our usecase, include the following:
        - 'token_required': 'Please enter your OTP token.'
        - 'invalid_token': 'Invalid token. Please make sure you have entered it correctly.'
        - 'n_failed_attempts': "Verification temporarily disabled because of %(failure_count)d failed attempt, please try again soon.", "Verification temporarily disabled because of %(failure_count)d failed attempts, please try again soon."
    - If the corresponding device is not confirmed (the user hasn't successfully logged in for the first time or it's the first time accessing the page) than the corresponding qrcode is displayed for the user to scan with Google Authenticator, for example;

```
from django_otp import login as otp_login

@my_login_required
def otp_view(request):
    context = {}
    if request.method == "POST":
        form = MyOTPTokenForm(user=request.user, data=request.POST)
        if form.is_valid():
            # user proves that has the secret key when validates a token for the first time
            # device -> confirmed
            device = TOTPDevice.objects.get(user=request.user)
            device.confirmed = True
            device.save()
            otp_login(request, device)
            request.session.set_expiry(settings.SESSION_COOKIE_AGE)
            return get_redirect_login(request)
    else:
        form = MyOTPTokenForm(user=request.user)
    # only 1 device per user
    device = TOTPDevice.objects.get(user=request.user)
    if not device.confirmed:
        qr_code_img = qrcode.make(device.config_url)
        buffer = BytesIO()
        qr_code_img.save(buffer)
        buffer.seek(0)
        encoded_img = b64encode(buffer.read()).decode()
        qr_code_data = f'{encoded_img}'
        context['qr_code_data'] = qr_code_data
    context['form'] = form
    return render(request, 'login2s.html', context)
```

Lastly, is important to mention that if a user signs up (creates an account) successfully he isn't logged in automatically. He's required to login for the first time, validate his login credentials and set up his 2FA device if he wishes to use the webapp.

Password Strength Evaluation

Associated to this improvement, we also implemented all the password strength ASVS V2.1 requirements from which we highlight the following:

- 2.1.1 - password length is at least 12 characters
- 2.1.2 - password length is at most 128 characters
- 2.1.3 - password truncation is not performed
- 2.1.4 - unicode characters allowed in password
- 2.1.7 - password **breach** verification with an external API
- 2.1.9 - no password composition rules
- 2.1.10 - no password credential rotation requirement
- 2.1.12 - temporarily view masked password

2.1.1

- Verify that user set passwords are at least 12 characters in length (after multiple spaces are combined).

Using Django's form validators, we used the following validation function for the **password** field in **account registration** and **change password** functionality.

```
# according to ASVS 2.1 password strength requirements
def validate_password(value):
    if len(value) < 12:
        raise ValidationError("Password must be at least 12 characters long.")

    if len(value) > 128:
        raise ValidationError("Password must be, at most, 128 characters long.")
```

This form validator can be found in the following forms:

- `UserRegistrationForm` - at line 25, the `password` field is validated with the `validate_password` function. It is validated when a user is creating a new account
- `ChangePasswordForm` - at line 64, both the `new_password` and `confirm_new_password` fields are validated with the previously mentioned function. Checked when a user tries to change his password.

2.1.2

- Verify that passwords of at least 64 characters are permitted, and that passwords of more than 128 characters are denied.

Using Django's form validators, we used the following validation function for the **password** field in **account registration** and **change password** functionality.

```
# according to ASVS 2.1 password strength requirements
def validate_password(value):
    if len(value) < 12:
        raise ValidationError("Password must be at least 12 characters long.")

    if len(value) > 128:
        raise ValidationError("Password must be, at most, 128 characters long.")
```

This form validator can be found in the following forms:

- `UserRegistrationForm` - at line 25, the `password` field is validated with the `validate_password` function. It is validated when a user is creating a new account
- `ChangePasswordForm` - at line 64, both the `new_password` and `confirm_new_password` fields are validated with the previously mentioned function. Checked when a user tries to change his password.

2.1.3

- Verify that password truncation is not performed. However, consecutive multiple spaces may be replaced by a single space. We did not implement any logic to truncate passwords. What the user types is exactly what the password will be. Multiple spaces are also not replaced by a single space.

The following video shows an example of this working:

https://github.com/detiuaveiro/2nd-project-group_04/assets/97046574/e293b9cb-d0b1-4109-8fb4-edbcf32639e3

2.1.4

- Verify that any printable Unicode character, including language neutral characters such as spaces and Emojis are permitted in passwords.

Django supports Unicode data straight away, we already had this requirement fulfilled.

2.1.7

- Verify that passwords submitted during account registration, login, and password change are checked against a set of breached passwords either locally (such as the top 1,000 or 10,000 most common passwords which match the system's password policy) or using an external API. If using an API a zero knowledge proof or other mechanism should be used to ensure that the plain text password is not sent or used in verifying the breach status of the password. If the password is breached, the application must require the user to set a new non-breached password. (2.1.7)

We check the password against a set of breached passwords using the *Have I Been Pwned* API.

We call the API endpoint with an HTTP GET at `https://api.pwnedpasswords.com/range/{first 5 hash chars}`, including the first 5 SHA-1 chars from hashing a password, and compare the remaining hash characters (full hash excluding the first 5 hash chars) to the response hashes. If our hash suffix matches anything, that means the password has been breached and is vulnerable to dictionary attacks, meaning it shouldn't be used.

For additional security, we include the `Add-Padding: true` header. This makes it so that responses have a random amount of extra hash entries (which we discard on receival), to make it harder for an outsider to determine anything about the nature of the request, such as which hash prefix we queried about. (blog post about it)

Implementation:

accounts/utils.py line 4

```
 # takes in password and returns number of times it has been breached
# returning 0 means it has not been breached
# returns -1 for invalid request
def check_for_breach(password):
    digest = hashes.Hash(hashes.SHA1())
    digest.update(password.encode("utf-8"))
    password_hash = digest.finalize().hex().upper()

    prefix, suffix = password_hash[:5], password_hash[5:]

    # request passwords with given prefix from have i been pwned API
    url = f"https://api.pwnedpasswords.com/range/{prefix}"
    res = requests.get(url=url, headers={"Add-Padding": "true"})
    if res.status_code == 200:  # OK
        # list of lists with [suffix, count] while also filtering out padded results
        pwned_passwords = [
            parts
            for line in res.text.splitlines()
            if (parts := line.split(":")) and parts[1] != "0"
        ]
        for hash_suffix, count in pwned_passwords:
            if suffix == hash_suffix:  # matched one of the breached passwords
                return count

        return 0
    return -1
```

It receives the password, hashes it with SHA-1 (using the python cryptography module) and calls the API with the first 5 hash chars. Filters out response entries where the count is 0 (0 breaches of password with a certain hash suffix means its a padded entry) and then loops the response entry list and tries to match our suffix to the response suffixes.

This is called on 3 occasions:

- **Account registration** - blocks creating an account if user tries to create account with breached password
- **Password change** - blocks changing password if user tries to change his password to a password that has been breached
- **Account login** - automatically warns user if his password has been breached and advises him to change it (doesn't lock user out of using website features)

2.1.9

- Verify that there are no password composition rules limiting the type of characters permitted. There should be no requirement for upper or lower case or numbers or special characters.

In the previous project, we had the following rules for passwords:

- at least one digit
- at least one special character
- at least one uppercase and one lowercase character
- password cannot contain part of email
- password cannot contain a substring of your username

For this project, we scrapped these rules in favor of ASVS 2.1.9.

Our previous requirements encourage weak passwords to meet those requirements, such as adding a '1' and '@' to the end, and making the first letter uppercase. (`P@ssword1` is a commonly breached password).

Instead, we should move towards using multi-factor authentication so that independently of which password a user chooses (may set simpler passwords to remember) we use another authentication factor that doesn't depend on human memory and can guarantee a higher level of security and protection against unauthorized access and breaches.

2.1.10

- Verify that there are no periodic credential rotation or password history requirements.

We had already met this requirement in the previous project.

Credential rotation and password history requirements are bad since it encourages making closely related weak passwords to meet those requirements.

2.1.12

- Verify that the user can choose to either temporarily view the entire masked password, or temporarily view the last typed character of the password on platforms that do not have this as built-in functionality.

We added a *Show Password* button to the registration and login pages, that allows a user to temporarily view his password.

static/js/utility.js (password input boxes have the 'passwordInput' id)

```
function togglePassword() {
    var x = document.getElementById("passwordInput");
    if (x.type === "password") {
      x.type = "text";
    } else {
      x.type = "password";
    }
  }
```

```
<input class="form-check-input" type="checkbox" onclick="togglePassword()" id="checkbox">
<label class="form-check-label" for="checkbox">
    Show Password
</label>
```

Whenever the user checks the input checkbox, the password will be shown.

Checklist Support Screenshots/Videos

This section contains videos and screenshots to support our observations in the ASVS checklist where relevant (we mention it in the checklist).

OWASP ZAP Automated Scan Results

Checked many details of both requests and results and input behaviours, such as injections (XSS/SQL/OS command injection), default credentials/broken authentication, CSRF (checks forms for CSRF token, for example), HTTP security headers in response (CSP, embedding related, Strict-Transport-Security, etc...), outdated vulnerable libraries/components, and others.

https://github.com/detiuaveiro/2nd-project-group_04/assets/97046574/df9daccf-3ef4-4f45-8e96-c58d03b74d3e

2.1.3

Password does not get truncated and multiple spaces are maintained, so the created user password is exactly what he gets:

https://github.com/detiuaveiro/2nd-project-group_04/assets/97046574/852e2ebc-fc95-4622-8898-7a9b00f0776a

2.1.11

Pasting passwords and using browser password managers is supported:

2_1_11_PASSWORD_MANAGER_PASTE.webm

3.2.1

When a new session is created, a new session cookie is generated and given to the user for the current session:

https://github.com/detiuaveiro/2nd-project-group_04/assets/97046574/e4889cdc-e681-4297-a7b0-a8c2264217ee

3.2.3

https://github.com/detiuaveiro/2nd-project-group_04/assets/97046574/41f010b1-8df6-4ab2-9940-985ca00ddc15

3.3.1

Pressing the back button shows a cached page (that appears as the user resumed his session) but doing any action, such as clicking the home page button correctly shows the user logged out:

https://github.com/detiuaveiro/2nd-project-group_04/assets/97046574/f03348da-b1df-4489-ac21-ecbf7b56c8d8

3.3.2

Operations refresh the session's expiration time:

https://github.com/detiuaveiro/2nd-project-group_04/assets/97046574/b7c95917-f145-4a10-9701-9dce62c27a06

3.4.1

https://github.com/detiuaveiro/2nd-project-group_04/assets/27146462/c57bd8d2-431d-4897-bb61-be66ced48417

3.4.2

https://github.com/detiuaveiro/2nd-project-group_04/assets/97046574/4b524090-6af2-4589-ab63-71f4f521dfef

3.4.3

https://github.com/detiuaveiro/2nd-project-group_04/assets/97046574/3c89ad19-de66-49a7-85ac-7a7864d6a099

3.4.4

https://github.com/detiuaveiro/2nd-project-group_04/assets/97046574/f2607db4-fd85-4ff4-9d7c-bc805cfc7264

3.7.1

Changing account data, such as e-mail does not prompt re-authentication:

https://github.com/detiuaveiro/2nd-project-group_04/assets/97046574/cdb86abd-4a20-48ee-9b01-6aa24ac845c7

4.1.1

Invoices files being served since they're in the media/ folder:

https://github.com/detiuaveiro/2nd-project-group_04/assets/97046574/f2f0e80d-7390-46a9-9a40-6ac8430a1cd5

4.1.5

Trying to access forbidden endpoints for regular users (the `dashboard` endpoints are for managers) does not throw a fatal exception, it blocks the user from accessing those pages:

https://github.com/detiuaveiro/2nd-project-group_04/assets/97046574/2e58652a-8d16-4986-b254-6eb8c7388b0d

4.2.2

https://github.com/detiuaveiro/2nd-project-group_04/assets/97046574/08ad4b87-1ec5-49f6-a184-25d1be1d22c8

5.1.1

Passing two `filename` URL parameters works, only the last one is considered:

https://github.com/detiuaveiro/2nd-project-group_04/assets/97046574/9c3c32be-2f15-4552-8d05-502ee12c8fa9

5.2.2

Review text didn't have length limitation:

https://github.com/detiuaveiro/2nd-project-group_04/assets/97046574/8b6260b6-e4e6-4de5-865e-83d92673a2b8

5.2.7

.svg files are forbidden, among many other dangerous extensions:

5_2_7_SVG_SAFE.webm

5.3.2

UTF-8 encoding is used:

https://github.com/detiuaveiro/2nd-project-group_04/assets/97046574/7ce2cb4a-a301-4118-a8c1-e197988a6412

7.4.1

When a user fails to login multiple times:

https://github.com/detiuaveiro/2nd-project-group_04/assets/97046574/ba337147-469b-4c45-83de-787f12def4b6

When a user inputs wrong email or password:

https://github.com/detiuaveiro/2nd-project-group_04/assets/97046574/00f7da16-8192-4150-b29f-e801a6ca6f44

8.2.2

After many actions in the webpage, the browser storage is still empty, except for cookies:

https://github.com/detiuaveiro/2nd-project-group_04/assets/97046574/7e795376-ca1e-450c-8b9a-469aa9c765df

8.3.1

Parameters are passed in HTTP body:

https://github.com/detiuaveiro/2nd-project-group_04/assets/97046574/43898619-0c09-4a3a-bba9-24d178dd76ae

12.3.1 & 12.3.2

Invoices created have the order ID as their name, and trying to path traversal through the file download URL API is not possible:

https://github.com/detiuaveiro/2nd-project-group_04/assets/97046574/2cd5696a-71b1-4de4-a8ec-6fcbcdccf49a

12.5.1

All of media folder is served, with no file extension restriction:

https://github.com/detiuaveiro/2nd-project-group_04/assets/97046574/311200c6-d1ff-4628-a206-04b202a58fb2

14.4.6

https://github.com/detiuaveiro/2nd-project-group_04/assets/97046574/54cc8155-3996-4d4e-874f-31a15cd370af

14.4.7

Trying to embed our website in a simple Flask web app (check network tab, it denied the request):

https://github.com/detiuaveiro/2nd-project-group_04/assets/97046574/dba4d601-7454-4559-b8c2-756e6da3ffe7

14.5.1

When an unused HTTP method (DELETE HTTP call to `/accounts/login` or invalid request is made to the server:

https://github.com/detiuaveiro/2nd-project-group_04/assets/27146462/43201b3e-ed9c-4e49-821d-ecfcb7ec144d