NTNU
Kunnskap for en bedre verden

Department of Computer Science

TDT4237 Software Security and Data Privacy

Exercise 4. Mitigating Vulnerabilities

Group 21

Authors:
Patrick Øivind Helvik Legendre
Gunnar Nystad
Dag Kirstihagen

Table of Contents

1 Introduction

We have previously mapped and exposed several vulnerabilities in the Trustedsitters application. We have given both white- and black-box explanations of these and will now address 15 of the vulnerabilities previously found. Based on the OWASP's web security testing guide, we will provide a generalized mitigation strategy and specific code changes.

2 WSTG-ATHN-01 / WSTG-CRYP-03

Sensitive Information Sent via Unencrypted Channels:

All information is sent through plain/unencrypted channels using the HTTP protocol when interacting with the application. This is especially dangerous when sending credentials.

2.1 Mitigation strategy

To mitigate the problem of Trustedsitters only using HTTP, we decided to implement HTTPS in the Nginx server. We generated a certificate and a private key for the HTTPS implementation.

2.2 Code change

The fix was made in commit **883f968d**.

We first change the protocol variable used in the settings.py file from using HTTP to HTTPS.

```
// backend/trustedsitters/settings.py

PROTOCOL = os.environ.get("PROTOCOL", "HTTPS")
```

We also change the protocol variable in the .env file to use HTTPS.

```
// .env

PROTOCOL=https
```

We changed ports in the docker-compose file to 443.

```
// docker-compose.yml

gateway:
    container_name: gateway_group_${GROUP_ID}
    build:
      context: nginx/
      dockerfile: Dockerfile
    ports:
      - ${PORT_PREFIX}${GROUP_ID}:443
```

We change the port from 80 to 443 SSL in the Nginx configuration file. We added an SSL certificate we generated. We also generated an RSA private key that should usually not be included in the repository. There is also support for TLS version 1.1 to version 1.3. The ciphers included are all ciphers considered strong, and we removed every cipher that uses MD5 as a hashing algorithm.

```
// nginx/nginx.conf
```

```
server {
       listen      443 ssl;
       server_name localhost;
       keepalive_timeout 70;
       ssl_certificate /etc/nginx/certs/localhost.crt;
       ssl_certificate_key /etc/nginx/certs/localhost.key;
       ssl_protocols TLSv1.1 TLSv1.2 TLSv1.3;
       ssl_ciphers HIGH:!aNULL:!MD5;
   add_header X-Content-Type-Options nosniff;
```

3 WSTG-ATHN-03

Unlimited login attempts, no lockout:

The application does not apply any lockout mechanism leaving it vulnerable to brute force attacks. By using a lockout mechanism, an attacker would not be able to perform a login password or username guessing attack.

3.1 Mitigation strategy

To mitigate the unlimited number of login attempts, we decided to use a Django library called Django Axes [1]. This library already has a lockout mechanism implementation built-in with different settings for us to modify.

3.2 Code change

The fix was made in commit **91cc4d16**.

```
// /backend/apps/users/serializers.py

def credentials(self, attrs):
       data = super().credentials(attrs)

       data['user'] = UserSerializer(self.user).data.username
       return data
```

Each time a user fails to match the username and the password, we will increase a counter for that username. If the user manages to get it right when the counter is less than 5, the number is reset to 0. If the user fails 5 times, he is locked out of his account.

```
// /backend/apps/users/views.py

@method_decorator(axes_dispatch, name='dispatch')
class LoginViewSet(viewsets.ModelViewSet, TokenObtainPairView):
    serializer_class = LoginSerializer
    permission_classes = (AllowAny,)
        try:
            serializer.is_valid(raise_exception=True)
        except TokenError as e:
            signals.user_login_failed.send(
                sender=User, request=request,
```

```
                credentials={
                    'username': serializer.credentials()
                })
            raise InvalidToken(e.args[0])
        user = serializer.validated_data.get("user")
        reset(username=user.get("username"))
        return Response(serializer.validated_data, status=status.HTTP_200_OK)

def lockout(request, credentials, *args, **kwargs):
    return JsonResponse({"status": "Locked out due to too many login failures"},
         status=403)
```

Using Django Axes made it easier for us to implement the lockout mechanism. We defined the number of tries you have to match your username and password to 5. After a lockout, the cool-off time for getting your account back is 23 hours.

```
// /backend/trustedsitters/settings.py

#Lock out Django Axes

AXES_FAILURE_LIMIT = 5
AXES_LOCKOUT_CALLABLE = "users.views.lockout"
AXES_ONLY_USER_FAILURES = True
AXES_COOLOFF_TIME = 23

AUTHENTICATION_BACKENDS = [
    # AxesBackend should be the first backend in the AUTHENTICATION_BACKENDS list.
    'axes.backends.AxesBackend',

    # Django ModelBackend is the default authentication backend.
    'django.contrib.auth.backends.ModelBackend',
]
```

4 WSTG-ATHN-04

Vulnerable MFA:

The application checks whether or not the user actually verifies itself through MFA and instantly sends the access tokens. Therefore the user can bypass the MFA by accessing any other page.

4.1 Mitigation strategy

To mitigate the vulnerable MFA exploit, we added a new category in the frontend for the MFA. We also changed the login view and created a new MFA login that does not ask for authentication. When the user logs in with MFA, it no longer goes through the usual login view and instead is treated as a separate login process.

4.2 Code change

The fix was made in commit **bed1ad2b**.

We added a new route call /api/mfaLogin/.

```
// backend/apps/users/urls.py

urlpatterns = [*router.urls,
               path("api/verify-email/<uid>/<status>",
                    views.VerificationView.as_view(), name="verify-email"),
               path('api/reset-password/<uidb64>/<token>/',
                    views.ResetPasswordView.as_view(), name='password-reset'),
               path('api/request-reset-password/',
                    views.PasswordResetEmailView.as_view(), name='password-reset-email'),
               path('api/reset-password-validate/',
                    views.SetNewPasswordView.as_view(), name='password-reset-valid'),
               path('api/mfa/', views.MFAView.as_view(), name='mfa'),
               path('api/mfaLogin/', views.MFALogin.as_view(), name='mfa-login'),
```

Instead of only sending OTP, we now send a request with the username and the OTP only after the username and password match.

```
// frontend/src/components/LoginForm.jsx

const onSubmitOTP = (e) => { //send confirmation to db
    e.preventDefault();
    const request = {username: username, otp: OTP};
    AuthService.postMFALogin(request).then((verified)=>{
      setUser({
        id: id,
        username: username,
        email: email
      })
```

When we receive the response from the backend to the frontend, we set the refresh, access token, and the user in the browser's local storage.

```
// frontend/src/services/auth.js

const postMFALogin = (otp) => {
  return api.post('/mfaLogin/', otp).then((response) => {
    if (response.data.access) {
      TokenService.setUser(response.data.user);
      TokenService.updateLocalAccessToken(response.data.access);
      TokenService.setLocalRefreshToken(response.data.refresh);
    }

    return response.data;
  });
}
```

MFALogin serializer makes a data dictionary and appends all the values needed for the frontend.

```
// backend/apps/users/serializers.py

class MFALoginSerializer(TokenObtainPairSerializer):

    def validate(self, user):
        data = {'user': None, 'refresh': None, 'access': None, 'mfa_verified': None}
        data['user'] = UserSerializer(user).data
        refresh = self.get_token(user)
```

```
        data['refresh'] = str(refresh)
        data['access'] = str(refresh.access_token)
        data['mfa_verified'] = user.mfa_active

        return data
```

The login serializer now makes a difference between logging in with and without MFA by checking the MFA verified value stored in the database for each user.

```
// backend/apps/users/serializers.py

class LoginSerializer(TokenObtainPairSerializer):

    def validate(self, attrs):
        if User.objects.get(username = self.user).mfa_active == False:
            refresh = self.get_token(self.user)
            data['refresh'] = str(refresh)
            data['access'] = str(refresh.access_token)
        else:
            data['refresh'] = None
            data['access'] = None

        data['mfa_verified'] = User.objects.get(username = self.user).mfa_active

        if api_settings.UPDATE_LAST_LOGIN:
            update_last_login(None, self.user)

        return data
```

MFA login checks the OTP sent in the request alongside the username. If the OTP token is valid, we generate a serializer that returns the data values in the request.

```
// backend/apps/users/views.py

class MFALogin(generics.GenericAPIView):
    serializer_class = MFALoginSerializer
    permission_classes = (AllowAny,)
    http_method_names = ['post']

    def post(self, request):
        serializers = self.get_serializer(data=request.data)
        otp = request.data.get('otp')
        username = request.data['username']
        try:
            if not otp.isdigit():
                return Response(status=status.HTTP_403_FORBIDDEN)
        except TokenError as e:
            raise InvalidToken(e.args[0])
        user = get_user_model().objects.get(username=username)

        if user.mfa_token is None:
            return Response(status=status.HTTP_403_FORBIDDEN)

        mfa_token = pyotp.TOTP(user.mfa_token)
        if mfa_token.verify(otp):
            data = serializers.validate(user)
            return Response(data, status=status.HTTP_200_OK)

        else:
```

```
        return Response(status=status.HTTP_401_UNAUTHORIZED)
```

5 WSTG-ATHN-07

Weak password validation:

The application only checks that the passwords aren't all numbers. It doesn't put any requirements for the content or the length of the password.

5.1 Mitigation strategy

Django has password validation functionality built-in. We can implement the standard validators to make the passwords more secure.

5.2 Code change

This fix was made in commit **c5e0d465**.

We expand the password validator in Django with several built-in functionalities, as given by the comments in the code below.

```
// /trustedsitters/backend/trustedsitters/settings.py


# Expanding the validator function to make sure passwords dont match user information,
    are of required length and aren't too common
AUTH_PASSWORD_VALIDATORS = [
    {
        # Makes sure the password isn't simmilar to the user's username, email, first name
            or last name. Set the max simmilarity to 0.6
        'NAME': 'django.contrib.auth.password_validation.UserAttributeSimilarityValidator',
        'Options': {
            'user_attributes': (
                'username', 'email', 'first_name', 'last_name'
            ),
            'max_smiliarity': 0.6
        }
    },
    { # Sets a required minimum password length of 8 characters
        'NAME': 'django.contrib.auth.password_validation.MinimumLengthValidator',
    },
    { # We check the password against a list of common passwords, we will be using the
        Django's default list.
        'NAME': 'django.contrib.auth.password_validation.CommonPasswordValidator',
    },
    { # Additionally we require the password to have a special character
        'NAME':
            'django_password_validators.password_character_requirements.password_validation.PasswordCharacter
            'OPTIONS': {
            'min_length_digit': 1,
            'min_length_special': 1,
            'min_length_lower': 1,
            'min_length_upper': 1,
            'special_characters': "~!@#$%^&*()_+{}:;'[]" + '"'
```

```
        }
    }
]

]
```

6 WSTG-ATHZ-02

Children API endpoint open:

When sending GET requests to /api/children/, the application returns a list of all Child objects. Only parents and guardians should have access to a child's information.

6.1 Mitigation strategy

The ChildViewSet class returned a QquerySet containing all Child objects. To mitigate this security flaw, we produce a QuerySet that applies a filter so that only parents and guardians can access a child's information.

6.2 Code change

The fix was made in commit **3bbc8a0a**.

We override the get_queryset function and check for which children the current user has a parent or guardian role.

```
// /backend/apps/children/views.py

    def get_queryset(self):
        qs = Child.objects.none()

        if self.request.user.id:
            user = self.request.user

            qs = Child.objects.filter(
                Q(parent__id=user.id) | Q(guardians__id=user.id)
            )
        return qs
```

7 WSTG-ATHZ-02

Accept offers on behalf of others:

The application does not check in the backend to verify that the one accepting the offer is the offer's recipient. Hence, by using Postman, for instance, a malicious user can send POST requests to /api/offers/ accepting offers on behalf of others.

7.1 Mitigation strategy

We apply a check to confirm that the user accepting the offer is its recipient to mitigate this security flaw.

7.2 Code change

The fix was made in commit **2c6af077**.

In the AnswerOfferView, we apply an additional check to confirm that the user trying to accept an offer is its recipient.

```
// /backend/apps/offers/views.py

    if offer.recipient != self.request.user.username:
        return Response({'success': False, 'message': 'Offer recipient does not match
            current user'}, status=status.HTTP_403_FORBIDDEN)
```

8 WSTG-SESS-01

Guessable email verification link:

The email verification link increments by one at a time and is therefore guessable. This allows malicious actors to navigate directly to the verification link and verify any email without the actual email owner's authentication.

8.1 Mitigation strategy

To mitigate against predictable email verification links, we had to include some random factors and make it harder to guess the process of composing the URL.

8.2 Code change

This fix was made in commit **f78152ef**.

We added a new field called verify_mail_url to the User database to store the token that we generated and then check if it matches with the one that the user requests to verify the mail.

```
// backend/apps/users/models.py

class User(AbstractUser):
    mfa_token = EncryptedCharField(max_length = 50, null=True, blank=True)
    mfa_active = models.BooleanField(default=False, blank=True)
    reset_url = models.CharField(max_length=256, null=True, blank=True)
    verify_mail_url = models.CharField(max_length=256, null = True, blank=True)
```

The first thing is to hide the uid. We first base64 encode and add 100 to the id to obfuscate further. Then we start working on the token. We record the exact time the request is sent in UNIX time. We use this Unix time as a seed for our random number generator. This number is used as a loop, and we have a minimum of 149 iterations on this loop in case the number generator returns

0. In this loop, we change the UNIX time by adding the number and subtracting the i (number of iterations). Then we use this new seed to generate a new number counter. This counter is appended to the seed and converted to a string. This string is then hashed with a sha256 function and base64 encoded. Then we store this token in the database and send the URL with uid and the token in a new mail.

```
// backend/apps/users/serializers.py

class UserSerializer(serializers.ModelSerializer):

        user.save()
        email = validated_data["email"]
        email_subject = "Activate your account"
        uid = user.pk
        uid = urlsafe_base64_encode(force_bytes(user.pk + 100))
        domain = get_current_site(self.context["request"])
        link = reverse('verify-email', kwargs={"uid": uid, "status": 1})
        date = datetime.datetime.utcnow()
        utc_time = calendar.timegm(date.utctimetuple())

        seed(utc_time)
        nb = random.randrange(0, 1000)
        for i in range(0, nb + 149):
            utc_time += nb
            utc_time -= i

        seed(utc_time)
        counter = randint(0, 10000)
        string = str(counter + utc_time)
        hashed_str = hashlib.sha256(string.encode('utf-8')).hexdigest()
        token = urlsafe_base64_encode(force_bytes(hashed_str))
        user.verify_mail_url = token
        user.save()
        link = reverse('verify-email', kwargs={"uid": uid, "status": token})

        url = f"{settings.PROTOCOL}://{domain}{link}"
```

We recover the uid, and base64 decode it. Then we have to subtract 100 from the id. We look up the user from the id in the database. We retrieve the token stored in the database and base64 decode it with the token that has been sent in with the URL. If it matches, then we activate the account. If it does not, we redirect the user to an invalid URL.

```
\\ backend/apps/users/views.py

class VerificationView(generics.GenericAPIView):
    def get(self, request, uid, status):
        verified_url = settings.URL + "/verified"
        invalid_url = settings.URL + "/invalid"
        try:
            id = force_text(urlsafe_base64_decode(uid))
            newid = int(id) - 100
            user = get_user_model().objects.get(pk=newid)
            token = user.verify_mail_url

            newtoken = force_text(urlsafe_base64_decode(token))
            newstatus = force_text(urlsafe_base64_decode(status))

            if str(newstatus) == str(newtoken):
                user.is_active = True
                user.verify_mail_url = None
```

```
            user.save()
        else:
            user.is_active = False
            user.save()
            return redirect(invalid_url)

        return redirect(verified_url)
```

9 WSTG-SESS-01 / WSTG-SESS-07

Authentication tokens lifetimes:

Trustedsitter's authentication token lifetime is too long. This gives malicious actors too long a time to try to hijack the user's session.

9.1 Mitigation strategy

We can easily change the duration of the access tokens we provide. We also have to update the lifetime of the refresh tokens. We base the new durations on the guidelines laid out by OWASP.

9.2 Code change

This fix was made in commit **3b945f42**.

```
// /trustedsitters/backend/trustedsitters/settings.py

SIMPLE_JWT = {
    # Changed access token lifetime from 6000 minutes to 30 minutes.
    'ACCESS_TOKEN_LIFETIME': timedelta(minutes = 30),
    # Changed the refresh token lifetime from 15 days to 1 day.
    'REFRESH_TOKEN_LIFETIME': timedelta(days = 1),
    'ROTATE_REFRESH_TOKENS': True,

}
```

10 WSTG-SESS-06

Access token only deleted client side:

Trustedsitters only deletes their access tokens client side. This allows users to recreate the access tokens and for them to be operational server side.

10.1 Mitigation strategy

We define a new view in the backend that handles the logout process to mitigate the problem of access tokens only being deleted on the client-side.

10.2 Code change

This fix was made in commit **66e00200**.

```
// /backend/apps/users/urls.py

path('api/logout/', views.LogoutView.as_view(), name='logout')

// /backend/apps/users/views.py

class LogoutView(generics.GenericAPIView):

    def post(self, request):
        refresh_token = self.request.data.get('refresh')
        token = RefreshToken(refresh_token)
        token.blacklist()
        return Response({"message": "logout successful"}, status=status.HTTP_200_OK)

def lockout(request, credentials, *args, **kwargs):
    return JsonResponse({"status": "Locked out due to too many login failures"},
         status=403)
```

11 WSTG-INPV-02 / WSTG-CLNT-03 / WSTG-CLNT-05

Unsanitized html field allowing injection:

By not checking the users-input, malicious actors could inject code that the server runs. This allows them to do actions they might not be authorized to do or are supposed to.

11.1 Mitigation strategy

Our strategy was to sanitize the child info input field. To do this, we used an open-source library called DOMPurify. It is designed to sanitize HTML strings and document objects from DOM-based XSS attacks.

11.2 Code change

The fix was implemented in commit **4605f307**.

The only necessary change was to sanitize the child.info field using DOMPurify.

```
// /frontend/src/components/Child.jsx

const DOMPurify = require('dompurify')(window);

<div dangerouslySetInnerHTML={{ __html: DOMPurify.sanitize(child.info) }}></div>

// /
```

12 WSTG-INPV-05

SQL injection when sending guardian offers:

When sending guardian offers, the input field allows malicious actors to write SQL code, allowing them to perform actions that are not intentionally allowed.

12.1 Mitigation strategy

Instead of SQL, we decided to use built-in Django functions to sanitize the input.

12.2 Code change

The fix was implemented in commit **f7e4e4dc**.

Instead of performing a raw SQL query, we decided to use the get function to check if the recipient is an actual user. We also check whether the sender exists.

```
// /backend/apps/offers/views.py

class OfferViewSet(viewsets.ModelViewSet):

    def perform_create(self, serializer):
        if self.request.data.get('recipient'):
            recipient_username = self.request.data['recipient'].strip()
            sender_username = self.request.user.username
            try:
                get_user_model().objects.get(username=recipient_username)
            except:
                raise ValidationError("Recipient does not exist")

            if not sender_username:
                raise ValidationError("Sender does not exist")

            if sender_username == recipient_username:
                raise ValidationError("Cannot send offer to yourself")

            serializer.save(recipient=recipient_username, sender=sender_username)
```

13 WSTG-INPV-05 / A06:2021

Vulnerable Django version:

Some Django versions are vulnerable to various security risks. Django version 3.2.4 is one of these [2].

13.1 Mitigation strategy

To mitigate this security flaw, we changed the Django version used in the application. Django version 3.2.12 is not considered vulnerable [2].

13.2 Code change

The fix was implemented in commit **f6505f3e**.

The only change necessary to make was to specify a safe Django version in requirements.txt.

```
// /backend/requirements.txt

Django==3.2.12
```

14 WSTG-CRYP-04

Insecure password hasher:

The application uses a weak password hasher that is cryptographically broken. This leaves the users' sensitive data exposed.

14.1 Mitigation strategy

To mitigate this security flaw, we switch to using a more secure password hasher in our application. Multiple hashers are considered cryptographically secure, and we can choose any one of them.

14.2 Code change

This fix was implemented in commit **9a9ee18a**.

We added the default set of password hashers for Django in settings.py. Django will use PBKDF2 to store all passwords but will support checking passwords stored with PBKDF2SHA1, argon2, and bcrypt [3].

```
// /trustedsitters/backend/trustedsitters/settings.py

PASSWORD_HASHERS = [
    'django.contrib.auth.hashers.PBKDF2PasswordHasher',
    'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
    'django.contrib.auth.hashers.Argon2PasswordHasher',
    'django.contrib.auth.hashers.BCryptSHA256PasswordHasher'
]
```

15 WSTG-BUSL-08

Unexpected file types:

The application allows file uploads but does not check whether or not the files match the intended file formats. The application description from exercise 3 states that only PDF and PNG files should be allowed.

15.1 Mitigation strategy

The ChildFile class did not specify allowed_mimetypes and allowed_extensions. Hence, no checks against the file format were made in the FileValidator class. We had to set these fields to fix the security flaw.

15.2 Code change

The fix was implemented in commit **85664680**.

The only change necessary was to specify the allowed extensions and MIME-types in the ChildFile class.

```
// /backend/apps/children/models.py

class ChildFile(models.Model):

    file = models.FileField(upload_to=child_directory_path, blank=False,
        validators=[FileValidator(allowed_mimetypes=('image/png', 'application/pdf'),
        allowed_extensions=('pdf', 'png'), min_size=100, max_size=8*1024*1024)])
```

16 WSTG-CLNT-09

Clickjacking:

The application is vulnerable to clickjacking which can be used to trick users into making interactions with Trustedsitters.

16.1 Mitigation strategy

The X-Frame-Options HTTP response header allows content publishers to prevent their content from being used by attackers. The header was already set for calls to the backend, so we had to do it so that frontend calls also sent the HTTP response header.

16.2 Code change

The change was made in commit **b0f5c1c9**.

The only change necessary was to add the X-Frame-Options header to nginx.conf.

```
// /nginx/nginx.conf

http {
    server {
       listen      443 ssl;
        server_name localhost;
        keepalive_timeout 70;
        ssl_certificate /etc/nginx/certs/localhost.crt;
        ssl_certificate_key /etc/nginx/certs/localhost.key;
        ssl_protocols TLSv1.1 TLSv1.2 TLSv1.3;
        ssl_ciphers HIGH:!aNULL:!MD5;
```

```
add_header X-Content-Type-Options nosniff;
add_header X-Frame-Options DENY always;
```

17 Conclusion

We have provided a mitigation strategy and specific code changes for 15 of the vulnerabilities present in the Trustedsitter application. By implementing these security improvements, Trustedsitter would help ensure the users could use the babysitting application while being safe from malicious actors.

References

[1] *DjangoAxes*. 2022. URL: https://github.com/jazzband/django-axes (visited on 06/04/2022).

[2] *django vulnerabilities*. 2022. URL: https://snyk.io/vuln/pip:django (visited on 04/04/2022).

[3] *Password management in Django*. 2022. URL: https://docs.djangoproject.com/en/4.0/topics/auth/passwords/ (visited on 06/04/2022).