NTNU
Kunnskap for en bedre verden

Department of Computer Science

TDT4237 Software Security and Data Privacy

Exercise 3. Mitigating Vulnerabilities

Group 19

Author(s):
Andrea Ritossa
Nerea Francés Pérez
Oskar Hetey

Table of Contents

1 Introduction

This report has been prepared by three second-year Computer Engineering students from Spain, Germany and Italy for the course TDT4237. The overall task of this exercise is to mitigate a predefined set of 10 vulnerabilities found within the application SecureHelp, that was provided by our teachers. In order to mitigate these vulnerabilities, we first defined a Mitigation Strategy for each vulnerability and then implemented it in the code.

Each vulnerability refers to a specified section of OWASP WSTG. For a deeper understanding of this report, it is suggested to review the OWASP page. In this report, we explain how we have mitigated each vulnerability and show the code differences: all code references can be found in our GitHub repository [1]. The initial code provided is in the main branch and our implementation can be found in the production branch.

2 WSTG-ATHN-01 / WSTG-CRYP-03 Sensitive Information Sent via Unencrypted Channels(TLS)

Sensitive information sent through Unencrypted Channels (TLS) makes the website vulnerable as it can expose data to unauthorized accesses.

For example, if a web application sends authentication credentials or JSON Web Tokens (JWTs) over an unencrypted channel, an attacker can capture them and use them to impersonate the user or access their account. [2] [WSTG-SESS-10]

2.1 Mitigation strategy

The mitigation is to use encrypted channels (HTTPS in our case) for all communications that involve sensitive data. [2] [WSTG-CRYPT-03]

2.2 Code change

Change the PROTOCOL .env variable to https:

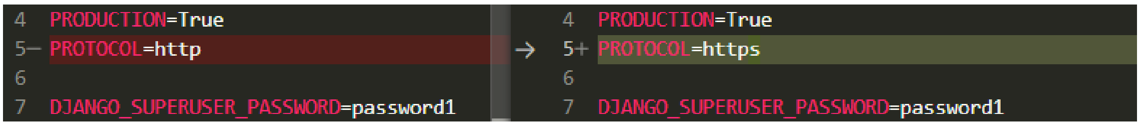


Figure 1: .env file

Change the port to 443 to support encrypted connection using SSL/TLS, with a change in docker-compose.yml as well. And specify the *self signed* certificate and key, after creating them with **openssl**.

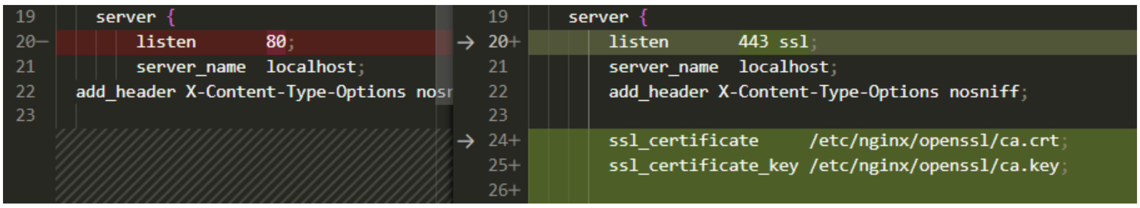


Figure 2: nginx.conf file

3 WSTG-ATHN-03 Unlimited login attempts, no lockout

WSTG-ATHN-03 deals with testing for weak lockout mechanisms. Account lockout mechanisms are used to mitigate brute force attacks: an example of the attacks that can be defeated by using lockout mechanism include login password or username guessing attack.

3.1 Mitigation strategy

Explain how you proceed to mitigate the vulnerability with a high level of abstraction. The first step is to get a deeper understanding of how the login was handled between frontend and backend: which functions are involved.

The function that from the backend operates on the login request is in views.py, line 70:

```
class LoginViewSet(viewsets.ModelViewSet, TokenObtainPairView):
    """ViewSet for logging in users. Extended from TokenObtainPairView"""
    serializer_class = LoginSerializer
    permission_classes = (AllowAny,)
    http_method_names = ['post'] # Only allow POST requests

    def create(self, request, *args, **kwargs):
        serializer = self.get_serializer(data=request.data)

        try:
            serializer.is_valid(raise_exception=True)
        except TokenError as e:
            raise InvalidToken(e.args[0])

        return Response(serializer.validated_data, status=status.HTTP_200_OK)
```

The application is missing a layer that checks the behaviour of the user before accessing the authentication made with simpleJWT.
There are multiple possibilities to mitigate it and we opted for implmenting ratelimit from django_ratelimit [3].

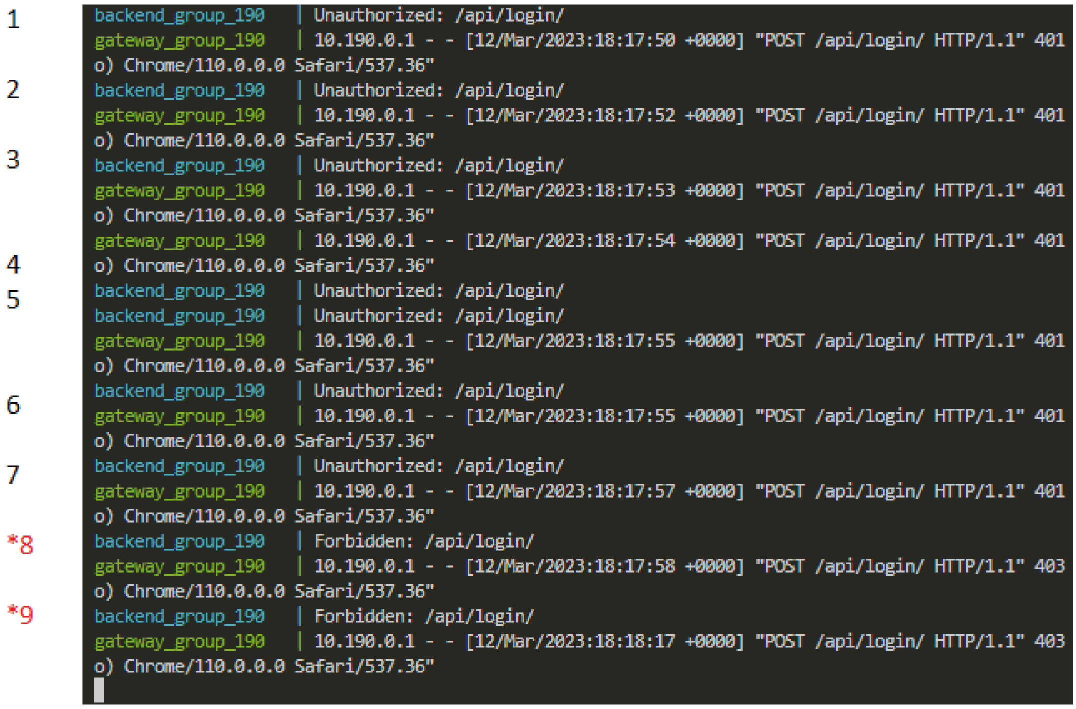
3.2 Code change

We added the @ratelimit decorator to the create method and set the rate limit to 10 requests per minute based on the user's IP address. If a user exceeds this limit, their request will be blocked.

```
# /views.py
from django.utils.decorators import method_decorator
from django_ratelimit.decorators import ratelimit
#[...]

@method_decorator(ratelimit(key='ip', rate='10/70m'), name='create')
class RegistrationViewSet(viewsets.ModelViewSet, TokenObtainPairView):
    """ViewSet for registering new users"""
    serializer_class = RegisterSerializer
    permission_classes = (AllowAny,)
    http_method_names = ['post']

    def create(self, request, *args, **kwargs):
        serializer = self.get_serializer(data=request.data)

        serializer.is_valid(raise_exception=True)
        user = serializer.save()
```

```python
# Create refresh token for user using simplejwt
refresh = RefreshToken.for_user(user)
res = {
    "refresh": str(refresh),
    "access": str(refresh.access_token),
}

return Response({
    "user": serializer.data,
    "refresh": res["refresh"],
    "token": res["access"]
}, status=status.HTTP_201_CREATED)
```

As a result when a potential attacker exceeds 7 requests in 15 minutes has forbidden access:



```
1   backend_group_190   | Unauthorized: /api/login/
    gateway_group_190   | 10.190.0.1 - - [12/Mar/2023:18:17:50 +0000] "POST /api/login/ HTTP/1.1" 401
    o) Chrome/110.0.0.0 Safari/537.36"
2   backend_group_190   | Unauthorized: /api/login/
    gateway_group_190   | 10.190.0.1 - - [12/Mar/2023:18:17:52 +0000] "POST /api/login/ HTTP/1.1" 401
    o) Chrome/110.0.0.0 Safari/537.36"
3   backend_group_190   | Unauthorized: /api/login/
    gateway_group_190   | 10.190.0.1 - - [12/Mar/2023:18:17:53 +0000] "POST /api/login/ HTTP/1.1" 401
    o) Chrome/110.0.0.0 Safari/537.36"
    gateway_group_190   | 10.190.0.1 - - [12/Mar/2023:18:17:54 +0000] "POST /api/login/ HTTP/1.1" 401
4   o) Chrome/110.0.0.0 Safari/537.36"
5   backend_group_190   | Unauthorized: /api/login/
    backend_group_190   | Unauthorized: /api/login/
    gateway_group_190   | 10.190.0.1 - - [12/Mar/2023:18:17:55 +0000] "POST /api/login/ HTTP/1.1" 401
    o) Chrome/110.0.0.0 Safari/537.36"
6   backend_group_190   | Unauthorized: /api/login/
    gateway_group_190   | 10.190.0.1 - - [12/Mar/2023:18:17:55 +0000] "POST /api/login/ HTTP/1.1" 401
    o) Chrome/110.0.0.0 Safari/537.36"
7   backend_group_190   | Unauthorized: /api/login/
    gateway_group_190   | 10.190.0.1 - - [12/Mar/2023:18:17:57 +0000] "POST /api/login/ HTTP/1.1" 401
    o) Chrome/110.0.0.0 Safari/537.36"
*8  backend_group_190   | Forbidden: /api/login/
    gateway_group_190   | 10.190.0.1 - - [12/Mar/2023:18:17:58 +0000] "POST /api/login/ HTTP/1.1" 403
    o) Chrome/110.0.0.0 Safari/537.36"
*9  backend_group_190   | Forbidden: /api/login/
    gateway_group_190   | 10.190.0.1 - - [12/Mar/2023:18:18:17 +0000] "POST /api/login/ HTTP/1.1" 403
    o) Chrome/110.0.0.0 Safari/537.36"
```

Figure 3: Lockout after 7th access

N.B. In the test number nine the username and password were right. Nevertheless in order to block a possible bruteforce attack it is required to forbid the access.

4 WSTG-SESS-01 No timeout on email verification

WSTG-SESS-01 deals with the Session Management Schema. In particular in securehelp there wasn't a timeout on email verification. This means that an attacker could use an old email verification link to gain access to a user's account.

4.1 Mitigation strategy

The first step was to understand the management of the Register function: it uses the following serializer to send the e-mail serializers.py, line 50:

```
class RegisterSerializer(UserSerializer):
 """Serializer for user registration"""
 password = serializers.CharField(
     max_length=128, min_length=1, write_only=True, required=True)
 email = serializers.CharField(
     max_length=128, min_length=1, required=True)

 class Meta:
     model = get_user_model()
     fields = ['id', 'username', 'email', 'password', 'is_volunteer']

 def create(self, validated_data):
 #   [...]
     return user
```

And after the verification is handled by the VerificationView in views.py, line 100.

```
 class VerificationView(generics.GenericAPIView):
 """View for verifying user registration links"""

 def get(self, request, uid):
     verified_url = settings.URL + "/verified"
     invalid_url = settings.URL + "/invalid"
     try:
         username = urlsafe_base64_decode(uid).decode()
         user = get_user_model().objects.filter(username=username).first()
         user.is_active = True # Activate user
         user.save()

         return redirect(verified_url)

     except Exception as ex:
         pass

     return redirect(invalid_url)
```

What's missing is a check on the time between the email delivery and the actual verification: fortunately the user model in 0001_initial.py, line 18, has an unused field named *date_joined*, ready to store the value of the registration.

```
 migrations.CreateModel(
         name='User',
         fields=[
             ('id', models.BigAutoField(auto_created=True, primary_key=True,
                 serialize=False, verbose_name='ID')),
             ('password', models.CharField(max_length=128, verbose_name='password')),
             [...]
```

```
            ('date_joined', models.DateTimeField(default=django.utils.timezone.now,
                verbose_name='date joined')),
            [...]
            ('user_permissions', models.ManyToManyField(blank=True,
                help_text='Specific permissions for this user.',
                related_name='user_set', related_query_name='user',
                to='auth.Permission', verbose_name='user permissions')),
        ],
        options={
            'verbose_name': 'user',
            'verbose_name_plural': 'users',
            'abstract': False,
        },
        managers=[
            ('objects', django.contrib.auth.models.UserManager()),
        ],
    ),
```

4.2 Code change

We added the **timestamp** from django.utils to the field *date_joined* of user:

```
class RegisterSerializer(UserSerializer):
    """Serializer for user registration"""
    password = serializers.CharField(
        max_length=128, min_length=1, write_only=True, required=True)
    email = serializers.CharField(
        max_length=128, min_length=1,  required=True)

    class Meta:
        model = get_user_model()
        fields = ['id', 'username', 'email', 'password', 'is_volunteer']

    def create(self, validated_data):
        user = get_user_model().objects.create_user(**validated_data)

        user.is_active = False  # set user to inactive until email is verified
        user.date_joined = timezone.now()  # record when the verification email was sent
        user.save()

        # create email to send to user
        email = validated_data["email"]
        email_subject = "Activate your account"
        uid = urlsafe_base64_encode(user.username.encode())
        domain = get_current_site(self.context["request"])
        link = reverse('verify-email', kwargs={"uid": uid})

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

        mail = EmailMessage(
            email_subject,
            url,
            None,
            [email],
        )
        mail.send(fail_silently=False)  # send email to user

        return user
```

Figure 4: date$_joineddeclarationinsideRegisterSerializer$

And implemented the **expiration** inside the VerificationView:

```python
class VerificationView(generics.GenericAPIView):
    """View for verifying user registration links"""

    def get(self, request, uid):
        verified_url = settings.URL + "/verified"
        invalid_url = settings.URL + "/invalid"
        expired_url = settings.URL + "/expired"
        try:
            username = urlsafe_base64_decode(uid).decode()
            user = get_user_model().objects.filter(username=username).first()
            if user and (timezone.now() - user.date_joined) < timedelta(hours=1):
                user.is_active = True  # Activate user
                user.save()
                return redirect(verified_url)
            else:
                return redirect(expired_url)
        except Exception as ex:
            print(ex)

        return redirect(invalid_url)
```

Figure 5: Checking if the link has been accessed in less than an hour

We also added an **expiration page** in the frontend, that can be found at "/expired".

5 WSTG-SESS-01 Weak email verification link.

WSTG-SESS-01 deals with the Session Management Schema. In particular in the sign-up phase the link that was sent from the server exposed the user id. The link was a base64 encoding of the username: it could easily be generated by an adversary.

5.1 Mitigation strategy

The lines of code that handle the signup are stated in the previous chapter SESS-01 No timeout on email verification.

A different way of storing the user id in authentication link is required: one way is by using a JSON Web Token (JWT) [4]. The server should first create a JWT, signed with a secret key. And in the JWT it can include the username. So when the link should be verificated it can be retrieved.

5.2 Code change

The user is so encoded with JWT:

1. Header: JWT with HS256 as algorithm
2. Payload: username
3. Signature: the SECRET KEY of the server

```
# create email to send to user
email = validated_data["email"]
email_subject = "Activate your account"

# create JWT with playload: username
playload = {'username': user.username}
secret_key = settings.SECRET_KEY
uid = jwt.encode(playload, secret_key, algorithm='HS256')

domain = get_current_site(self.context["request"])
link = reverse('verify-email', kwargs={"uid": uid})

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

Figure 6: Class RegisterSerializer, of serializers.py

And in VerificationView, view.py line 104 we added the decoding:

```
#decode JWT and extract username from playload
secret_key = settings.SECRET_KEY
playload = jwt.decode(uid, secret_key, algorithms=['HS256'])
username = playload['username']
```

6 WSTG-ATHZ-02 Approve certification as a normal user

The *permission_classes* attribute in Django REST Framework views is used to specify the access control permissions required to access the view. Permissions are used to determine whether a user is allowed to perform a certain action on a resource.

6.1 Mitigation strategy

By changing the permission class to *permissions.IsAdminUser*, only administrators have access. We also delete *permissions.IsAuthenticated*, thus restricting access to the rest of the users.

6.2 Code change

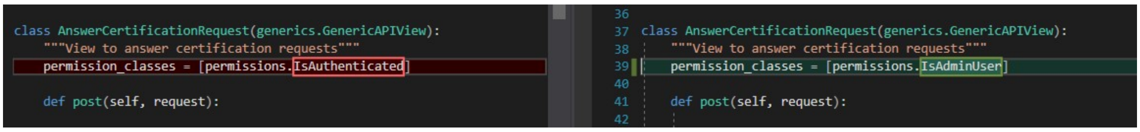


Figure 7: updated permissions in backend/apps/certifications/views.py

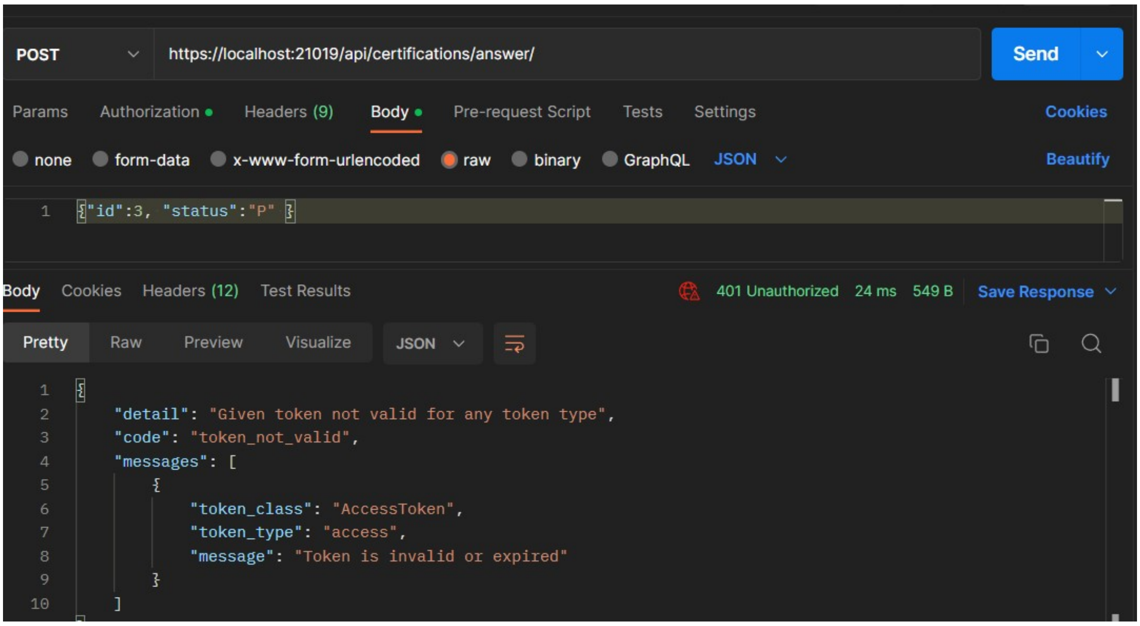


Figure 8: Postman output for approving a certification request as a normal user after changing the permissions.

7 WSTG-SESS-06 Access token only deleted client side

It is possible getting a new access token by using an old refresh token, after logging out of the application.

7.1 Mitigation strategy

To fix this vulnerability, the server-side code was updated to revoke the token upon logout, invalidating it on both the client and server-side. This was accomplished by sending a request to the server to revoke the token, along with removing it from the client-side storage. This ensures that the token cannot be used by any party after logout.

It is no longer possible to use an old refresh token to obtain a new access token. This is because the server will reject the old refresh token as invalid, and the user will have to log in again to obtain a new refresh token and access token.

7.2 Code change

```
router.register('api/login', views.LoginViewSet, basename='login')
router.register('api/refresh', views.RefreshViewSet, basename='refresh')
router.register('api/documents', views.DocumentViewSet, basename='documents')
router.register('api/logout', views.LogoutView, basename='logout')
```

Figure 9: Registering the LogoutView class in backend/apps/users/urls.py

```
class LogoutView(APIView):
    def post(self, request, format=None):
        try:
            access_token = AccessToken(request.data.get('access_token', ''))
            access_token.blacklist()
        except TokenError:
            pass
        request.user.auth_token.delete()
        return Response(status=status.HTTP_200_OK)
```

Figure 10: Invalidating tokens with the LogoutView class in backend/apps/users/views.py

```
const removeUser = () => {
    const access_token = getLocalAccessToken();
    axios.post("/api/logout/", { access_token }).then(() => {
        localStorage.removeItem("user");
        localStorage.removeItem("access_token");
        localStorage.removeItem("refresh_token");
    });
};

export { removeUser };
```

Figure 11: Sending a request to the server in frontend/src/services/token.js

8 WSTG-INPV-02 WSTG-CLNT-03 WSTG-CLNT-05 Unsanitized html field allowing injection.

To fix this vulnerability, it is necessary to properly sanitizing input data and validating it before it is displayed in HTML.

8.1 Mitigation strategy

To mitigate this vulnerability, the code was changed to use the *DOMPurify.sanitize()* function to sanitize the *description* attribute before inserting it into the HTML. This function uses a whitelist-based approach to remove any potentially malicious HTML tags or attributes from the input.

8.2 Code change

Figure 12: import DOMPurify in frontend/src/components/HelpRequest.jsx

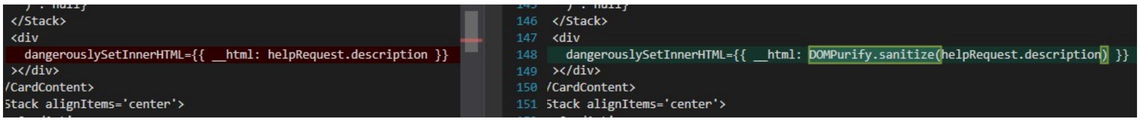


Figure 13: Sanitizing description attribute using DOMPurify in frontend/src/components/HelpRequest.jsx

9 WSTG-INPV-05 SQL injection when finishing a help request

The input variable for finishing help requests is not sanitized and allows for SQL injection. The basic SQL-Injection for reading out more request than it should be allowed from the solution is being used:

```
{"request_id":"1' or 1=1; --"}
```

9.1 Mitigation strategy

In this modified code, **%s** is replaced with a placeholder that is then passed to the **objects.raw()** method along with the value of **rId**. Using parameterized queries automatically escapes any special characters in the user input, making it safe to use in an SQL query.

9.2 Code change

Change the in views.py at query processing

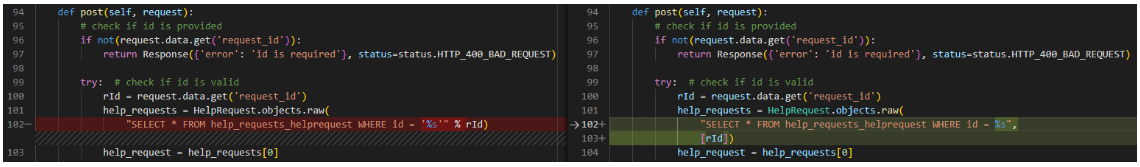


Figure 14: views.py file

9.3 Comparison

These screenshots show the request results as seen in firefox before and after the code change. The different status codes and response show the results.

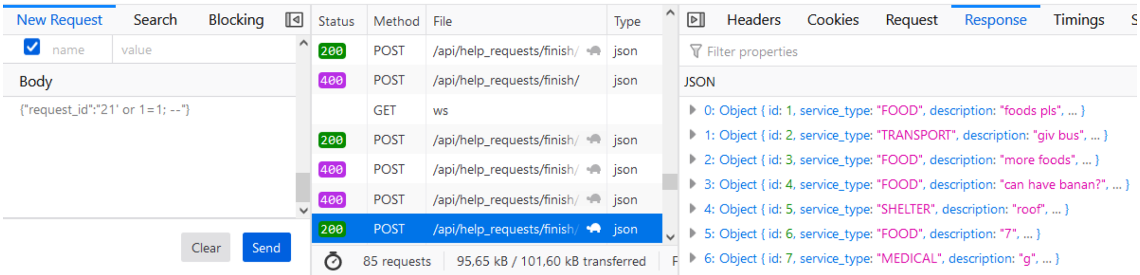


Figure 15: Successful SQL-Injection

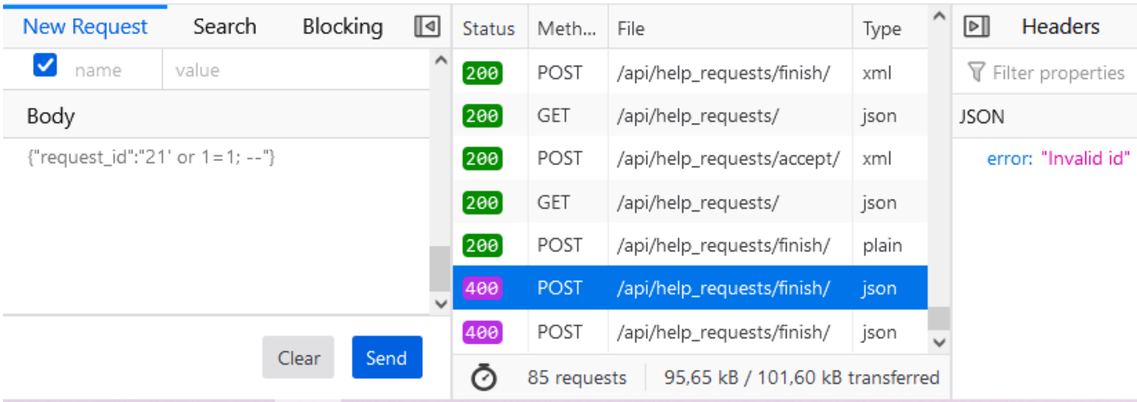


Figure 16: Sanitized SQL-Injection

10 WSTG-CRYP-04 Insecure password hasher

Among the outdated and insecure encryption algorithms is also found SHA1, which is being used here [5] [WSTG-CRYP-04]. Since it is being used unhashed it can simply be decoded as shown in the solutions. It is recommended to use password hashers such as PBKDF2 with a SHA256 hash function and a large number of iterations, which provides much stronger protection against password cracking attacks. [6] [Django Password Management].

10.1 Mitigation strategy

The used password hasher SHA1 is being replaced by the recommended secure algorithms. 150000 number of iterations are used to prevent attackers from attempting the cracking of passwords and also the salt length is specified so it's long enough to ensure that each salt is unique and unpredictable. Doing this in developement mode results in a change of all existing passwords, so it may be needed to be still logged in as admin and create a new user after changing the algorithm to not lose access to the accounts.

10.2 Code change

The PBKDF2 password hasher was specified as the primary hasher, followed by the BCrypt and Argon2 hashers as fallbacks.

backend/securehelp/settings.py

```diff
@@ -174,9 +174,15 @@ DEFAULT_AUTO_FIELD = 'django.db.models.BigAutoField'
 PASSWORD_RESET_TIMEOUT = 3600  # Token valid for one hour
 
 PASSWORD_HASHERS = [
-    'django.contrib.auth.hashers.UnsaltedSHA1PasswordHasher',
+    'django.contrib.auth.hashers.PBKDF2PasswordHasher',
+    'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',
+    'django.contrib.auth.hashers.Argon2PasswordHasher',
 ]
 
+PBKDF2_ITERATIONS = 150000
+PBKDF2_SALT_LENGTH = 32
+
+
 REST_FRAMEWORK = {
     'DEFAULT_AUTHENTICATION_CLASSES': (
         'rest_framework_simplejwt.authentication.JWTAuthentication',
```

Figure 17: updated password hashers in /backend/securehelp/settings.py

11 WSTG-CONF-12 CSP Default source not set

No Content Security Policy is set. Users can upload content like scripts without inspection.

11.1 Mitigation strategy

To avoid this, the nginx configuration has to be changed. An header is added in the http-block, self is being used as default source, with the aim of not blocking the original page. There have been some problems on localhost trying to figure out how to configure the header. When adding it, content may be misplaced. When changing from *self* to *localhost* even images may not be loading correctly.

11.2 Code change

The added header is found in the http section in /nginx/nginx.conf

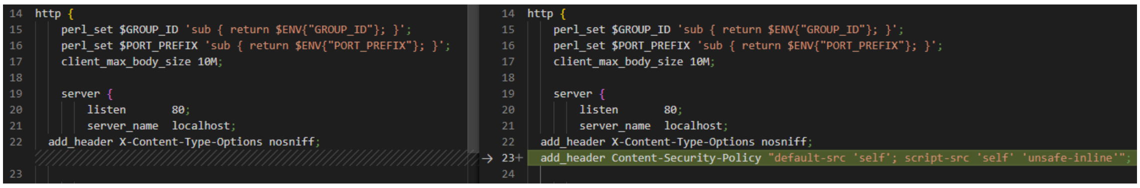


Figure 18: added header in nginx.conf

12 Conclusion

Vulnerabilities in code can be exploited by attackers to gain unauthorized access to sensitive data or to disrupt the normal operation of an application. To address these vulnerabilities, developers should implement appropriate mitigation strategies that help to reduce the risk of exploitation. It is important to use secure coding practices and to perform regular security testing and code reviews in order to identify vulnerabilities in code before they can be exploited by attackers. Moreover, using security frameworks and libraries can also be an effective way to mitigate code vulnerabilities. By implementing these mitigation strategies, developers can improve the security of their applications and protect them from potential attacks.

References

[1] group 019. *github repository of the project*. 2023. URL: https://gitlab.stud.idi.ntnu.no/tdt4237/2023/group-019.

[2] OWASP. *OWASP WSTG*. URL: https://owasp.org/www-project-web-security-testing-guide/v41/.

[3] django$_r$*atelimit*. *Django Ratelimit Docs*. 2022. URL: https://django-ratelimit.readthedocs.io/en/stable/usage.html.

[4] JWT. *JWT Documentation*. URL: https://jwt.io/.

[5] OWASP. *OWASP WSTG CRYP 04*. URL: https://owasp.org/www-project-web-security-testing-guide/stable/4-Web_Application_Security_Testing/09-Testing_for_Weak_Cryptography/04-Testing_for_Weak_Encryption.

[6] Django Documentation. *Django Password Management*. URL: https://docs.djangoproject.com/en/4.1/topics/auth/passwords/.