NTNU
Kunnskap for en bedre verden

Department of Computer Science

TDT4237 Software Security and Data Privacy

Exercise 2. Mitigating Vulnerabilities

Group 41

Author(s):
Gard Huse Storebø
Arthur Marc Jacques Saunier
Kjetil André Woll Vik

Table of Contents

1 Introduction

Web applications are often targeted by attackers looking to exploit security vulnerabilities. This report focuses on identifying and mitigating several security issues found within our application, following the guidelines from the Web Security Testing Guide (WSTG). By addressing these issues, we aim to improve the overall security of our system and protect user data from potential threats.

2 WSTG-CRYP-04 - Testing for Insecure Password Hasher

The password hasher was set to ‘UnsaltedSHA1PasswordHasher‘, which is an insecure hashing algorithm that is susceptible to brute-force and rainbow table attacks. SHA-1 is considered weak due to its vulnerability to collision attacks and lack of salting, making it easy to crack stored passwords.

2.1 Mitigation Strategy

To mitigate this vulnerability, the password hashing algorithm was updated to use a secure alternative. Django provides multiple built-in secure hashers, including:

- PBKDF2 with SHA-256 (default)
- Argon2 (recommended)
- BCrypt

The application was updated to enforce a secure hasher by modifying the ‘PASSWORD_HASHERS‘ setting in ‘settings.py‘. In addition, all users were required to reset their passwords to ensure that they were stored using the new secure hashing algorithm.

2.2 Code Change

The following changes were made in the ‘settings.py‘ file to replace the insecure SHA-1 hasher with more secure alternatives:

```python
# /backend/secfit/settings.py

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

After applying this change, all users were prompted to reset their passwords, ensuring that old passwords hashed with SHA-1 were replaced with securely hashed passwords.

3 WSTG-SESS-07 - Testing Session Timeout

The session timeout for access tokens was found to be excessively long. The access token lifetime was set to 72 hours, which significantly increases the risk of misuse if the token is stolen.

3.1 Mitigation Strategy

To mitigate this issue, the access token expiration time was reduced to 15 minutes. The refresh token lifetime was also reviewed to ensure security.

3.2 Code Change

The following changes were applied in the ‘settings.py‘ file to enforce a more secure session timeout.

```
# /backend/secfit/settings.py

SIMPLE_JWT = {
    'ACCESS_TOKEN_LIFETIME': timedelta(minutes=15),
    'REFRESH_TOKEN_LIFETIME': timedelta(days=7),
    'ROTATE_REFRESH_TOKENS': True
}
```

4 WSTG-SESS-06 - Testing for Logout Functionality

4.1 Mitigation strategy

When a user logs out of the application, his refresh token is blacklisted, so it can not be used anymore (by him or any attacker).

4.2 Code change

Prior to its functionality being activated, the blacklisting feature for Django must be enabled in the settings.

```
# /backend/secfit/settings.py

SIMPLE_JWT = {
    'ACCESS_TOKEN_LIFETIME': timedelta(minutes=15),
    'REFRESH_TOKEN_LIFETIME': timedelta(days=7),
    'ROTATE_REFRESH_TOKENS': True,
    'BLACKLIST_AFTER_ROTATION': True,
}
```

Subsequently, the logout function must be modified to invoke the /api/logout/ route, which allows the token to be blacklisted.

```
# /frontend/src/components/AuthContext.jsx

const logout = async () => {
    const refreshToken = localStorage.getItem("refreshToken");
    if (refreshToken) {
      try {
      const result = await fetch("/api/logout/", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ "refresh": refreshToken }),
      });
      const data = await result.json();
    } catch (error) {
      console.error("Logout failed:", error);
    }
  }
    SessionService.removeLocalAccessToken();
    SessionService.removeLocalRefreshToken();
    SessionService.removeUserId();
    SessionService.removeUserName();
    SessionService.removeIsCoach();

    setIsAuthenticated(false);
  };
```

5 WSTG-INPV-05 - Testing for SQL Injection

5.1 Mitigation strategy

The issue arises from the use of raw SQL queries with unsanitized input. Instead, Django's ORM should be used as follow to safely query the database.

5.2 Code change

```
# /backend/users/views.py

query = "SELECT * FROM users_user WHERE username = %s"
    with connection.cursor() as cursor:
        cursor.execute(query, [username])
        columns = [col[0] for col in cursor.description]
        rows = cursor.fetchall()
```

6 WSTG-ATHZ-02 - Testing for Bypassing Authorization

6.1 Mitigation strategy

The issue occurs because the current implementation does not verify if the authenticated user is the intended recipient of the offer before allowing modifications. This allows unauthorized users to accept or decline offers on behalf of others.

To mitigate this, the backend should enforce an additional check to ensure that only the recipient of the offer can modify its status.

6.2 Code change

```
# backend/users/views.py

    def put(self, request, *args, **kwargs):
        id = kwargs.get('pk')
        offer = Offer.objects.get(pk=id)

        if request.user != offer.recipient:
            return Response({'error': 'You are not authorized to modify this offer'},
                status=403)
        try:
            offer.status = request.data.get('status')

            # Add the sender of the offer to the recipients list of athletes, and add the
                recipient to the senders coach list
            if offer.status == 'a':
                if not offer.recipient.isCoach:
                    return Response({'error': 'Recipient is not a coach'}, status=400)

                offer.recipient.athletes.add(offer.owner)
                offer.recipient.save()

                # Delete other pending offers from the same sender to the same recipient
                Offer.objects.filter(owner=offer.owner, recipient=offer.recipient,
                    status='p').delete()
        ...
```

7 WSTG-ATHN-04 - Testing for Bypassing Authentication

7.1 Mitigation strategy

This flaw exists because the athletes' routes are not protected by the authentication mechanism implemented in the `ProtectedRoute` component. To fix this, we should ensure that all routes that require authentication are wrapped within the `ProtectedRoute` component. This way, unauthorized users attempting to access restricted pages will be redirected to the login page.

7.2 Code change

```
// Before
<Route path="/athletes" element={<Athletes />} />
<Route path="/athletes/:id" element={<AthletesFilesRoute />} />

// After
<Route element={<ProtectedRoute />}>
    <Route path="/athletes" element={<Athletes />} />
    <Route path="/athletes/:id" element={<AthletesFilesRoute />} />
</Route>
```

8 WSTG-ATHN-03 - Testing for Weak Lockout Mechanism

8.1 Mitigation strategy

The issue arises because there is no mechanism to limit consecutive failed login attempts. This leaves the application vulnerable to brute force attacks. To address this, we can use the `django-axes` library, which integrates with Django to automatically track and block multiple failed login attempts. This solution is more efficient and consistent than writing custom middleware.

8.2 Code change

```
# In settings.py, add axes to installed apps and middleware

```

```
INSTALLED_APPS = [
    ...
    'axes',
    ...
]

MIDDLEWARE = [
    ...
    'axes.middleware.AxesMiddleware',
    ...
]

AUTHENTICATION_BACKENDS = [
    'axes.backends.AxesBackend',
    'django.contrib.auth.backends.ModelBackend',
]

# Configuration options to customize lockout behavior
AXES_FAILURE_LIMIT = 5 # Number of allowed attempts before lockout
AXES_COOLOFF_TIME = timedelta(minutes=5) # Lockout duration
AXES_LOCKOUT_CALLABLE = 'axes.utils.is_locked' # Lockout function
```

```
     .catch((error) => {
        if (error.response) {
          const status = error.response.status;
          const data = error.response.data;

          if (status === 403 && data.retry_after_minutes) {
            setErrorMessage(`You're locked out due to too many failed login attempts.
                Please try again in ${data.retry_after_minutes} minutes.`);
          } else if (status === 403) {
            setErrorMessage("You're locked out due to too many failed login attempts.
                Please try again later.");
          } else if (status === 401) {
            setErrorMessage("The username or password you entered is incorrect");
          } else {
            setErrorMessage("Something went wrong. Please try again.");
          }
        } else {
          setErrorMessage("Network error. Please check your connection.");
        }
      });
  };
```

9 WSTG-INPV-02 - Testing for Stored Cross-Site Scripting

9.1 Mitigation strategy

The vulnerability arises because user-generated content is rendered as HTML instead of plain text. The use of the dangerouslySetInnerHTML attribute allows the execution of injected scripts, which can lead to stored XSS attacks.

To mitigate this issue, we must:

- Sanitize user inputs before storing them in the database to remove malicious scripts.
- Escape HTML entities when rendering user-generated content to ensure it is treated as text, not executable code.

- Use libraries such as DOMPurify to strip harmful tags while still allowing safe formatting.
- Implement Content Security Policy (CSP) headers to limit the execution of inline scripts.

By applying these protections, we ensure that user inputs cannot execute malicious scripts, preventing attackers from injecting harmful code into the application.

9.2 Code change

```
# frontend/src/components/CommentSectionForm.jsx

import DOMPurify from 'dompurify';
...
comments.map((comment) => (
           <Paper sx={{ textAlign: "start" }} key={comment.id}>
             <div style={{ display: "flex" }}>
               <h3>{comment.owner}</h3>
               {comment.owner === user.username && (
                 <IconButton
                   sx={{ marginLeft: "auto" }}
                   type="button"
                   onClick={() => handleDeleteComment(comment.id)}
                   aria-label="delete"
                 >
                   <DeleteIcon />
                 </IconButton>
               )}
             </div>
             <p>{DOMPurify.sanitize(comment.content)}</p>
             <p>{getTimeSincePosted(comment.timestamp)}</p>
           </Paper>
         ))
```

Also added **DOMPurify** module in `package-lock.json` and in `package.json` as a dependency.

10 WSTG-ATHN-01 - Testing for Credentials Transported over an Encrypted Channel

10.1 Mitigation strategy

The issue arises because the application does not encrypt network traffic, allowing attackers to intercept credentials sent in plaintext. To fix this, we configure TLS (Transport Layer Security) on the Nginx server to ensure secure communication between the client and the server.

This involves:

- Updating the Nginx configuration to listen on port 443 instead of port 80.
- Generating an SSL/TLS certificate and a corresponding private key.
- Configuring Nginx to use the generated certificate for secure HTTPS connections.
- Redirecting all HTTP traffic to HTTPS to enforce encryption.

10.2 Code change

```
# nginx/nginx.conf

http {
    server {
        listen 80;
        listen [::]:80;
        return 301 https://$host$request_uri;
    }
    server {
        listen 443 ssl;
        listen [::]:443 ssl;

        ssl_certificate /etc/nginx/ssl/self.crt;
        ssl_certificate_key /etc/nginx/ssl/self.key;
        ssl_protocols TLSv1.2 TLSv1.3;

        add_header Strict-Transport-Security "max-age=31536000; includeSubDomains" always;

# docker-compose.yml

nginx:
    container_name: nginx
    build:
      context: nginx/
      dockerfile: Dockerfile
    image: nginx:latest
    networks:
      - backend_bridge
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf
      - ./nginx/ssl:/etc/nginx/ssl
    depends_on:
      - frontend
      - backend
    restart: unless-stopped

# nginx/Dockerfile

FROM nginx:perl

RUN mkdir -p /etc/nginx/ssl && \
    chown -R nginx:nginx /etc/nginx/ssl && \
    chmod -R 755 /etc/nginx/ssl

COPY . /etc/nginx/

# .github/workflows/deploy.yml

jobs:
  deploy:
    runs-on: self-hosted
    steps:
      - name: Checkout repository
        uses: actions/checkout@v3

      - name: Generate SSL Certificates
        run: |
mkdir -p nginx/ssl
openssl req -x509 -nodes -days 365 -newkey rsa:2048 \
    -keyout nginx/ssl/self.key \
    -out nginx/ssl/self.crt \
    -subj "/NO=tdt4237-41.idi.ntnu.no"
```

11 WSTG-BUSL-08 - Testing for Upload of unexpected files WSTG-BUSL-09 - Testing for Upload of malicious files

11.1 Mitigation strategy

The issue is that the WorkoutFile model allows coaches to upload files without robust validation, which could lead to the upload of unexpected file types. To mitigate this, we applied the existing FileValidator to the file field in the Workout file models.py. This validator checks the file extension, MIME type (based on the file name), and file size. This approach ensures that only files with allowed extensions and MIME types are uploaded, while also enforcing a maximum file size limit.

11.2 Code change

```
# /backend/workouts/models.py

file = models.FileField(
        upload_to=workout_directory_path,
        validators=[FileValidator(
            allowed_extensions=['jpg', 'png', 'pdf'],
            allowed_mimetypes=['image/jpeg', 'image/png', 'application/pdf'],
            max_size=10 * 1024 * 1024 # 10 MB
        )]
    )
```

12 Conclusion

Through this security assessment, we resolved multiple vulnerabilities that could have been exploited by malicious users. By implementing proper authentication, authorization checks, input sanitization, and secure data transmission, we significantly strengthened the security of the application.