



DEPARTMENT OF COMPUTER SCIENCE

TDT4237 SOFTWARE SECURITY AND DATA PRIVACY

---

## Exercise 1. Finding Vulnerabilities

---

GROUP 100

*Authors:*

Ana Barrera Novas

Andrea Cicinelli

Nicola Katja Gisela Ferger

---

## Table of Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>WSTG-CONF-02</b>	<b>1</b>
<b>3</b>	<b>WSTG-IDNT-02</b>	<b>2</b>
<b>4</b>	<b>WSTG-ATHN-01</b>	<b>5</b>
<b>5</b>	<b>WSTG-ATHN-02</b>	<b>7</b>
<b>6</b>	<b>WSTG-ATHN-03</b>	<b>8</b>
<b>7</b>	<b>WSTG-ATHN-04</b>	<b>10</b>
<b>8</b>	<b>WSTG-ATHZ-03</b>	<b>11</b>
<b>9</b>	<b>WSTG-ATHZ-04</b>	<b>14</b>
<b>10</b>	<b>WSTG-ATHZ-04</b>	<b>14</b>
<b>11</b>	<b>WSTG-SESS-06</b>	<b>15</b>
<b>12</b>	<b>WSTG-SESS-07</b>	<b>18</b>
<b>13</b>	<b>WSTG-INPV-02</b>	<b>18</b>
<b>14</b>	<b>WSTG-INPV-05</b>	<b>20</b>
<b>15</b>	<b>WSTG-CRYP-03</b>	<b>21</b>
<b>16</b>	<b>WSTG-CRYP-04</b>	<b>23</b>
<b>17</b>	<b>WSTG-CLNT-09</b>	<b>24</b>
<b>18</b>	<b>WSTG-CLNT-12</b>	<b>26</b>
<b>19</b>	<b>WSTG-BUSL-05</b>	<b>29</b>
<b>20</b>	<b>Conclusion</b>	<b>29</b>

---

# 1 Introduction

This is the report of Group 100 for exercise 1 of the course TDT4237. Here, we list and give a detailed description and exploitation of some security vulnerabilities found in the code and deployed application given: SecFit.

The authors of this report are the following:

- Ana Barrera Novas: an exchange student studying a bachelor in Computer Science in the University of A Coruña, Spain.
- Andrea Cicinelli: an exchange student from Italy. I'm currently studying Computer Science at Sapienza, University of Rome.
- Nicola Katja Gisela Ferger: an exchange student from Germany, doing her master's in Computer Science at the Julius Maximilians University of Würzburg.

To approach this exercise, we met in several occasions to start, plan, solve doubts and put in common our development throughout the weeks working on the project. We kept constant track of the vulnerabilities investigated and found, while letting each member work individually on their discoveries.

We followed the instructions from the OWASP Web Security Testing Guide [1] and used the tools Zap, Postman and Wireshark to exploit (and sometimes find) the vulnerabilities.

Following this introduction, we present the found vulnerabilities and the appropriate white-box and black-box explanation. The vulnerabilities are ordered following the Web Application Security Testing Guide given for the project.

## 2 WSTG-CONF-02

Throughout the development and testing of the application, configuration files, documentation and test pages might contain a lot of functionality which is not needed and could actually develop into a big vulnerability in deployment. In this case, going over the source code, we found that something has been overlooked before releasing the application.

### 2.1 White-Box

---

```
# settings.py, line 29
# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = True
```

---

This line implies that the project is running in debug mode, which has several effects. For example, if an error occurs, Django will display a detailed traceback with debugging and sensitive information, including environment variables and request details.

It should be changed into `DEBUG = False` for it to stop being a vulnerability.

### 2.2 Black-Box

An attacker then could obtain information that could be useful for them for other attacks. To do so he would only have to access a page that gives an error, for example:

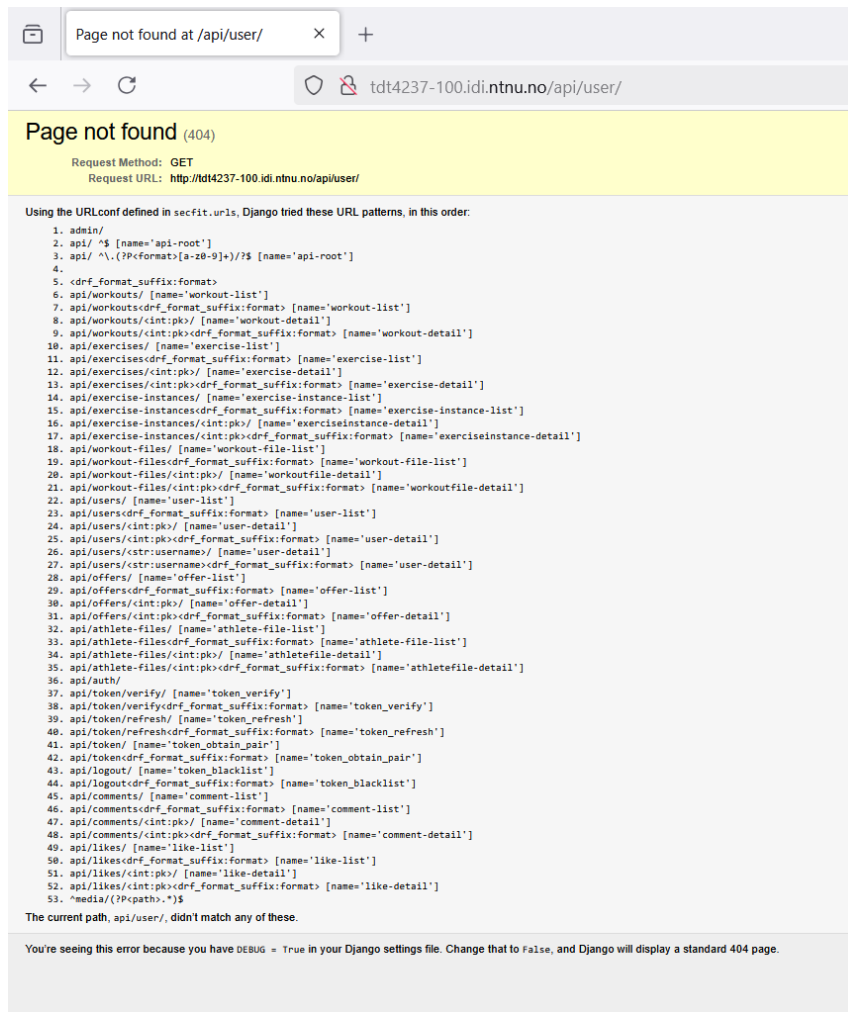


Figure 1: Extra info shown when error 404 pops

We can see there that all that information (in this case, about urls) has been exposed just because of this `DEBUG=True`, and in case it hadn't been there, just a normal error page would be shown, which is strictly what the client needs to see.

### 3 WSTG-IDNT-02

In some applications, the system access a user is granted is determined during the registration process. Depending on the app, this could be done on the basis of the role the user has selected out of several. This is the case in SecFit as well, where new users can choose between the role of athlete and coach, and coaches have more tools available than athletes, such as uploading training files for their athletes. In these cases, it is important that identity requirements are in place to ensure that users can only assign themselves the roles that correspond to the business and security requirements. If there are identity requirements, it is important that they are validated sufficiently.

In SecFit, however, this is not the case. The app does not check the identity or qualifications of users, so anyone can pose as somebody they are not or call themselves a coach. This is especially bad since the site crowns itself as being very secure, and gives users the option of hiding their workouts from others. Users might trust somebody because of their name, even though the person behind that account is someone completely different who the athletes might not want to share their workouts with.

---

### 3.1 White-Box

The biggest issue is the fact that users can assign themselves a coach status without any further verification. The problem lies in the insufficient implementation of the `UserSerializer` in `SetFit/backend/users/serializers.py`:

---

```
class UserSerializer(serializers.HyperlinkedModelSerializer):
    password = serializers.CharField(style={"input_type": "password"}, write_only=True)
    password1 = serializers.CharField(style={"input_type": "password"},
                                      write_only=True)

    class Meta:
        model = get_user_model()
        fields = [
            "url",
            "id",
            "email",
            "username",
            "password",
            "password1",
            "athletes",
            "isCoach",
            "coach",
            "specialism",
            "workouts",
            "coach_files",
            "athlete_files",
        ]

    def validate_password(self, value):
        data = self.get_initial()

        password = data.get("password")
        password1 = data.get("password1")

        try:
            password_validation.validate_password(password)
        except forms.ValidationError as error:
            raise serializers.ValidationError(error.messages)

        if password != password1:
            raise serializers.ValidationError("Passwords must match!")

        return value

    def create(self, validated_data):
        username = validated_data["username"]
        email = validated_data["email"]
        isCoach = validated_data["isCoach"]
        if (isCoach):
            specialism = validated_data["specialism"]
            user_obj = get_user_model()(username=username,
                                         email=email, isCoach=isCoach, specialism=specialism)
        else:
            user_obj = get_user_model()(username=username, email=email, isCoach=isCoach)
        password = validated_data["password"]
        user_obj.set_password(password)
        user_obj.save()

        return user_obj
```

---

The only custom validation is for `password`. All of the other fields are validated using the default

---

**Register as a new user**

Username

Email

Status

Specialism

Password

Confirm password

REGISTER

Figure 2: The attacker registers as Kayla Itsines with the coach status

Send a coach offer

SEND

**anabar**

whatever

**this\_is\_a\_coach**

Nothing

**Kayla\_Itsines**

High Intensity, Personal Trainer

Figure 3: On the coaches tab, the attacker is now displayed as Kayla.Itsines for all athletes to see

methods of the Django Rest Framework. For `isCoach` for example, it would only check if it is a boolean value or can be cast to one, which is not enough. Instead, coaches could be approved by admins beforehand and provided with unique usernames. When `isCoach` is set to `True`, the username could be validated against the whitelist. Alternatively, coach status could be set to pending until an admin approves it. Only after that, the new user is displayed as a coach and has access to the functionalities. However, there are many different ways to tackle this problem.

## 3.2 Black-Box

If an attacker wants to target a specific athlete, they could pose as a coach they know their athlete knows in their personal life. Alternatively, an attacker could target more users by posing as a well known coach. This is approach is demonstrated here.

First, the attacker registers as a coach with the name of famous fitness influencer Kayla Itsines, see Figure 2. Since the attacker is not required to prove this identity, they are now displayed for all athletes on the coach tab as Kayla Itsines, as shown in Figure 3. Users that know the influencer might now send that account a coach offer because they know and like her videos, implicitly trusting that account. The attacker then has access to all of these users' shared exercises, even though they might not want that.

---

## 4 WSTG-ATHN-01

Secure web applications use HTTPS to ensure a safe data transfer between the client and server in both directions. Authentication data can be transmitted during the submission of credentials to log in, and attackers can take over accounts by sniffing network traffic. In this case, we have found that the requests and responses look as the following:

---

```
POST http://tdt4237-100.idi.ntnu.no/api/token/ HTTP/1.1
host: tdt4237-100.idi.ntnu.no
(...)
{"username":"admin","password":"admin"}
```

---

This means that the information of the login is being sent over an unsafe channel (http). This also happens in other exchanges of valuable information, such as when creating a new user.

### 4.1 White-Box

This vulnerability occurs due to a bad configuration of the application. The "settings.py" file in backend/secfit/ needs to have at least these configuration lines somewhere:

---

```
# Redirect all HTTP traffic to HTTPS
SECURE_SSL_REDIRECT = True

# Ensure cookies are only sent over HTTPS
SESSION_COOKIE_SECURE = True
CSRF_COOKIE_SECURE = True
```

---

This way you force the traffic to go through HTTPS, without this lines the traffic can go through a non-secure channel.

### 4.2 Black-Box

An attacker can obtain credentials by sniffing the network. There are many ways of doing it, but we'll explain the way we tested it ourselves.

1. Configure OWASP ZAP as a proxy in the victim's browser (including installing ZAP's root certificate)

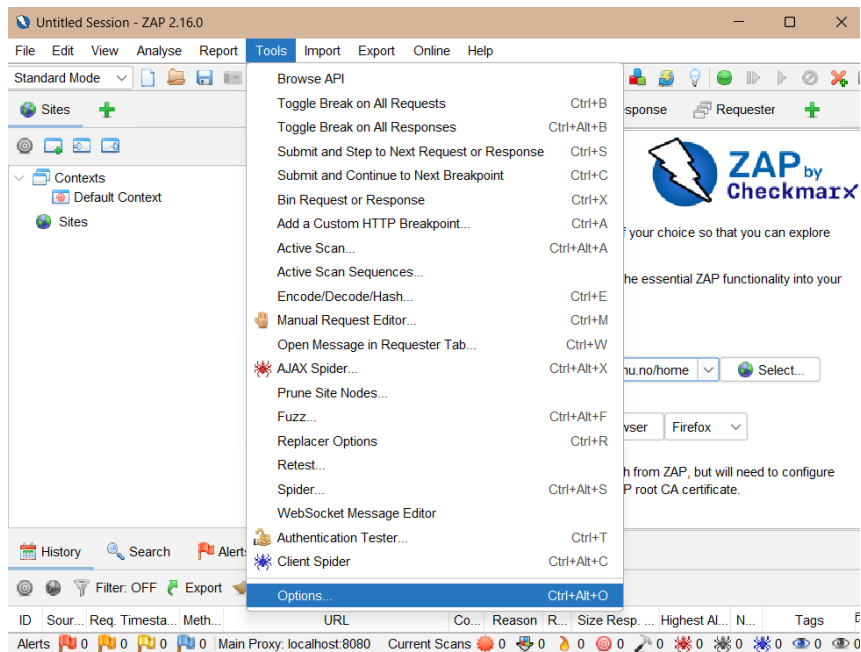


Figure 4: Navigate through ZAP menu

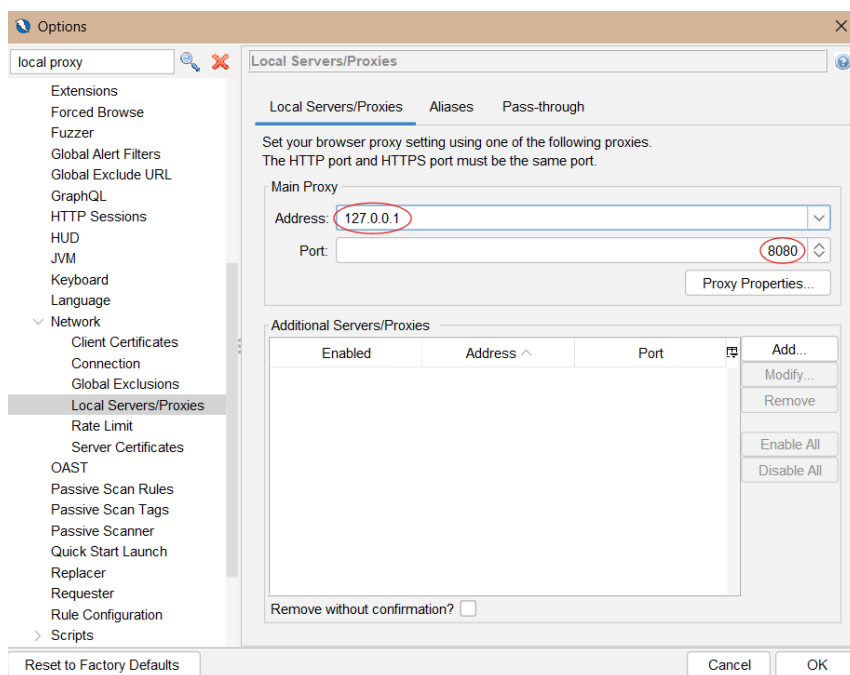


Figure 5: Find ZAP's address and port



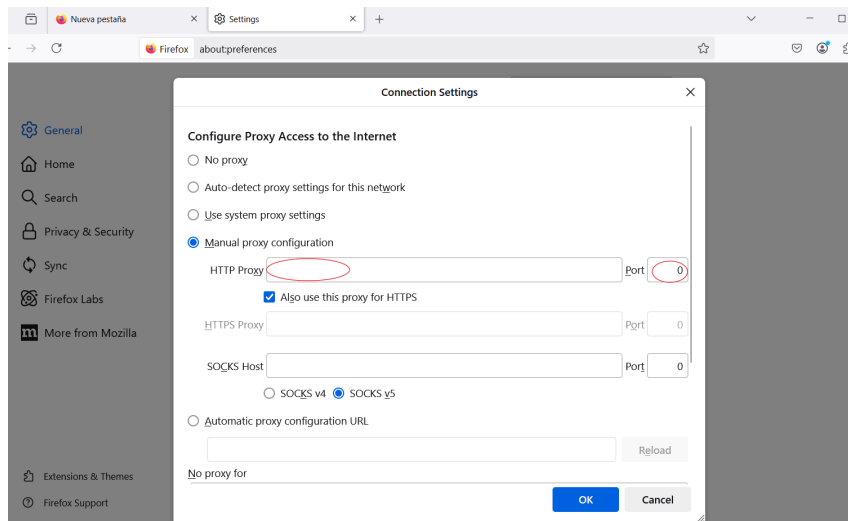


Figure 6: Find the network configuration in the preferred browser and establish ZAP's address as the proxy

2. Capture HTTP requests and inspect login forms, authentication headers, and session cookies.



Figure 7: How captured traffic might look like

3. Once the attacker has the authentication data they can log in as the victim and use it for any mean.

## 5 WSTG-ATHN-02

Many web applications and hardware appliances come with default credentials that are often not changed after installation. These credentials are widely known by both security testers and attackers, making applications vulnerable. This is the case of SecFit, which has the administrator login with username "admin" and password "admin".

---

## 5.1 Black-box

In order to exploit this, the attacker only has to brute force test these widely known credentials until they find the right ones. In this case:

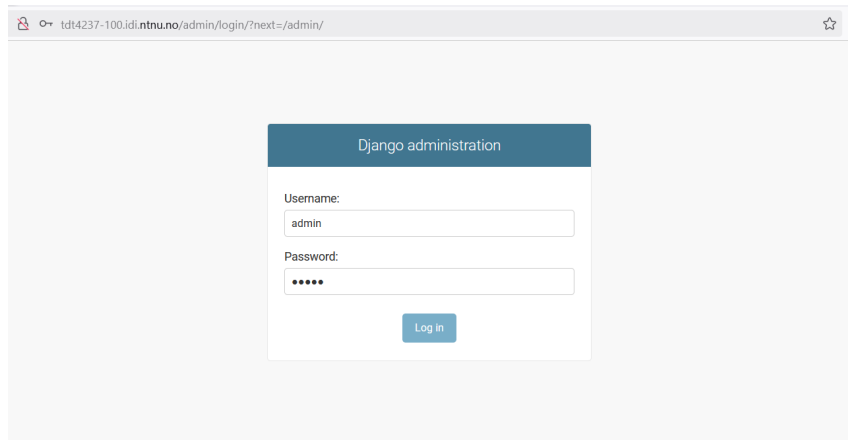


Figure 8: Django access: admin : admin

And now they would have access to everything: users, workouts, authentication and authorization details... And would have permission to create new, delete or change details, give or take away permissions...

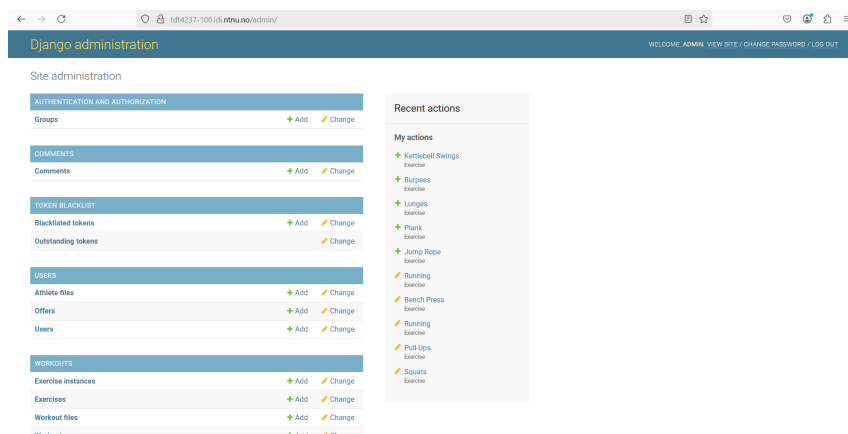


Figure 9: Admin interface from where you can access everything

## 6 WSTG-ATHN-03

A lockout mechanism can be weak in many different ways. On one hand, it could be too lenient, allowing brute force attacks. On the other hand, it could be too strict, locking valid users out of their accounts. A good lockout mechanism should find a compromise between these two sides.

Since SecFit handles sensitive workout data they promise to keep private, the application should value security and take precautions against brute force attacks. However, there is no lockout mechanism or CAPTCHA in place.

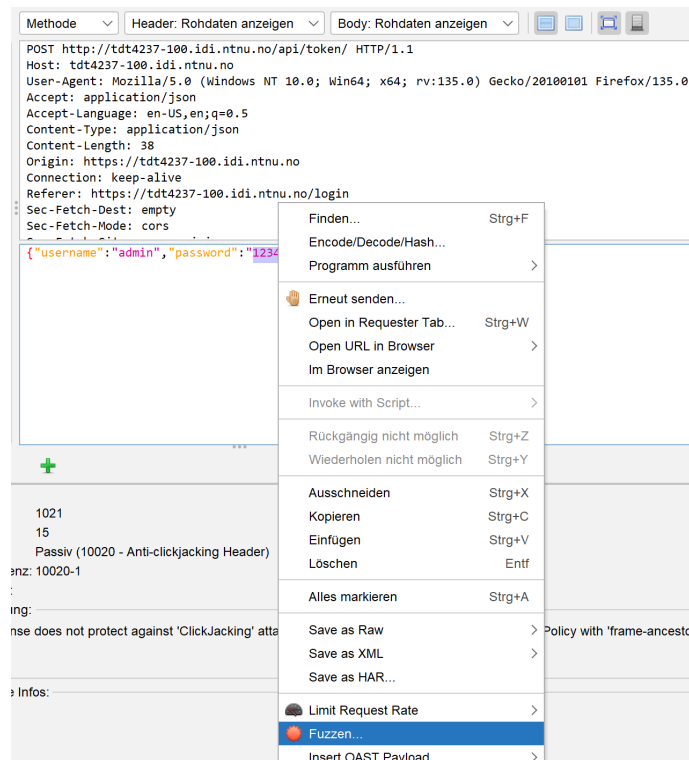


Figure 10: In the request, the attacker can select and right-click on the entered password. Then they choose 'Fuzz...'

## 6.1 White-Box

Pointing out the exact point in the code where this lockout mechanism is missing is not possible, since it could have been implemented in many different ways.

A good way would have been to use django-axes. In that case, the insufficient code would be in `backend/secfit/settings.py` where it should have been downloaded, added to middleware and configured. It would then have to be called during authentication.

## 6.2 Black-Box

Since executing brute-force attacks is not allowed during this exercise, this black-box exploitation will only be outlined.

Due to the non-existent lock-out mechanism, a brute-force attack can be performed, such as a dictionary attack. Many password dictionaries can be found and accessed easily on the internet (for example on the GitHub repository [Seclists](#) [2]). Additionally, there are tools that facilitate a dictionary attack. The one used in this explanation is ZAP.

The attacker puts in 'admin' as username, since it is very common, and wants to perform a dictionary attack on that account's password.

1. In ZAP: Set break on all requests and responses
2. In Browser: Try to log into account with a random password
3. In ZAP: Select Fuzz... on the password, pictured in Figure 10
4. Select a file fuzzer and start fuzzer; this step is explained more thoroughly in Figure 11
5. Wait for fuzzer to finish

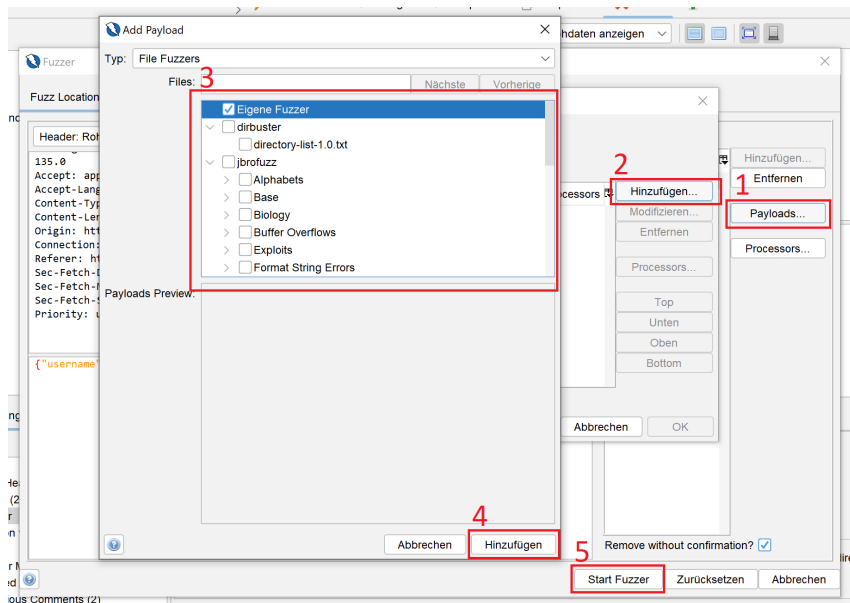


Figure 11: The attacker can now select the payloads that should be used for fuzzing, using the steps displayed in this Figure in chronological order. On Step 3, an existing file fuzzer or a local file fuzzer (which can be a txt file) can be selected

## 6. Sort by response code and find anomaly

The last two steps are not supported by screenshots since as per the instructions, the brute force attack was not executed. However, there most likely would have been an anomaly found when the payload "admin" was used. The attacker can now use the found password to log into the admin account.

## 7 WSTG-ATHN-04

A common issue is the ability to bypass the authentication schema. This could be done by changing requests or simply by calling an internal, unprotected site directly.

In SecFit, anyone, even logged out accounts, can access <http://tdt4237-100.idi.ntnu.no/api/users/>.

### 7.1 White-Box

The problem lies in `backend/users/view.py` in the following class:

```
class UserList(mixins.ListModelMixin, mixins.CreateModelMixin, generics.GenericAPIView):
    serializer_class = UserSerializer

    def get(self, request, *args, **kwargs):
        self.serializer_class = UserGetSerializer
        return self.list(request, *args, **kwargs)

    def post(self, request, *args, **kwargs):
        return self.create(request, *args, **kwargs)

    def get_queryset(self):
        qs = get_user_model().objects.all()

        if self.request.user:
```

---

```

# Return the currently logged in user
status = self.request.query_params.get("user", None)
if status and status == "current":
    qs = get_user_model().objects.filter(pk=self.request.user.pk)

return qs

```

---

There are no permission classes such as `permission_classes = [IsAuthenticated, IsAdminUser]` added which allows everyone to access the url.

## 7.2 Black-Box

This vulnerability can be exploited quite easily to steal users' private data, such as email addresses. The attacker navigates to the url `http://tdt4237-100.idi.ntnu.no/api/users/`. All users are displayed in the following manner:

---

```

{
  "url": "http://tdt4237-100.idi.ntnu.no/api/users/1/",
  "id": 1,
  "email": "admin@mail.com",
  "username": "admin",
  "athletes": [],
  "isCoach": false,
  "specialism": "",
  "coach": null,
  "workouts": [
    "http://tdt4237-100.idi.ntnu.no/api/workouts/1/"
  ],
  "coach_files": [],
  "athlete_files": []
}, ...

```

---

This makes extracting personal information – i.e. their email addresses, in this case `admin@mail.com` – quite easy.

## 8 WSTG-ATHZ-03

There are two types of accounts: athletes and coaches.

An athlete account has to send an "offer" to a coach account.

We assume that only the coach account can accept the offer, but this is not true.

### 8.1 White-Box

The PUT method is exposed to all user

---

```

class OfferDetail(
    mixins.RetrieveModelMixin,
    mixins.DestroyModelMixin,
    generics.GenericAPIView,
):
    permission_classes = [IsAuthenticatedOrReadOnly] #This is the vulnerability!
    queryset = Offer.objects.all()
    serializer_class = OfferSerializer

    def get(self, request, *args, **kwargs):

```

---

---

```
    return self.retrieve(request, *args, **kwargs)

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

    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()
```

---

This means that you need an authorization, but not the specific one of the coach.

## 8.2 Black-Box

When analyzing the traffic of acceptance of an offer, we can see what request actually accepts / denies the offer.

Look at figure 12

If the attacker tries to replicate the request with Postman,

We first insert our header (see figure 13)

This header doesn't need to be the header of the coach, we can insert the access token of any account.

The body is then in JSON format (see figure 14)

This is composed of the following: URL of the offer, id of the offer, owner (athlete), recipient (coach), and status.

Status may be a or p (accepted or pending).


If we send the PUT request on the link of the offer, we can accept the offer without the coach.


```


> Transmission Control Protocol, Src Port: 59769, Dst Port: 80, Seq: 505, Ack: 113, Len: 961
> Hypertext Transfer Protocol
v JavaScript Object Notation: application/json
  v Object
    v Member: url
      [Path with value: /url:http://tdt4237-100.idi.ntnu.no/api/offers/8/]
      [Member with value: url:http://tdt4237-100.idi.ntnu.no/api/offers/8/]
      String value: http://tdt4237-100.idi.ntnu.no/api/offers/8/
      Key: url
      [Path: /url]
    v Member: id
      [Path with value: /id:8]
      [Member with value: id:8]
      Number value: 8
      Key: id
      [Path: /id]
    v Member: owner
      [Path with value: /owner:newAtleta]
      [Member with value: owner:newAtleta]
      String value: newAtleta
      Key: owner
      [Path: /owner]
    v Member: recipient
      [Path with value: /recipient:http://tdt4237-100.idi.ntnu.no/api/users/14/]
      [Member with value: recipient:http://tdt4237-100.idi.ntnu.no/api/users/14/]
      String value: http://tdt4237-100.idi.ntnu.no/api/users/14/
      Key: recipient
      [Path: /recipient]
  > Member: status
  > Member: timestamp

```


Figure 12: Screenshot of WireShark. The PUT request when the coach accept the offer


<http://tdt4237-100.idi.ntnu.no/api/offers/2/>

 Save
 


 Share

PUT



http://tdt4237-100.idi.ntnu.no/api/offers/2/

Send



Params

Auth

Headers (9)


Body

Scripts

Settings

Cookies

Headers

 8 hidden

	Key	Value	D	...	Bulk Edit	Presets
<input checked="" type="checkbox"/>	Authorization	Bearer eyJhbGciOiJIUzI1Ni...				
	Key	Value			Description	

Figure 13: We can see how to insert the header with the access token

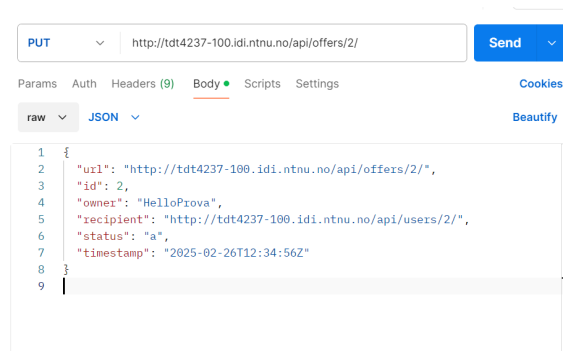


Figure 14: We can see an example of JSON body

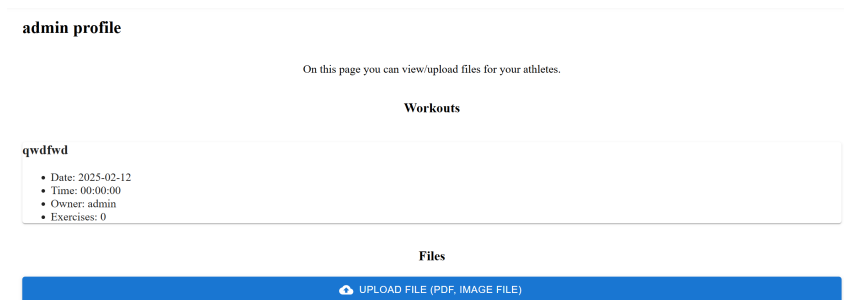


Figure 15: User profile of user admin

## 9 WSTG-ATHZ-04

An Insecure Direct Object Reference occurs when an application exposes internal objects without proper authorization checks. This allows attackers to access or manipulate data simply by modifying a URL, request parameter, or API call.

In SecFit, only the athlete's coach is supposed to be able to access an athlete's page and upload files for them. However, the path to each athlete's profile is defined as `/athletes/:id`, therefore making the url for each profile obvious. Through the url, the athlete's page with the file upload can be accessed, regardless if the user is the athlete's coach or not.

### 9.1 Black-Box

The attacker logs into an account and types in `http://tdt4237-100.idi.ntnu.no/athletes/1`, and is forwarded to the user 1's profile, which is the admin.

The attacker now has an overview of all of the public workouts of that user and also sees the user's username, as shown in figure 15. This way, the attacker can find existing profiles and use that information for further attacks.

## 10 WSTG-ATHZ-04

Almost every ID on the website is guessable and predictable. This is because IDs are assigned incrementally (1, 2, 3 etc..)

### 10.1 White-Box



---

```

class Offer(models.Model):
    """Django model for a coaching offer that one user sends to another.

    Each offer has an owner, a recipient, a status, and a timestamp.

    Attributes:
        owner:      Who sent the offer
        recipient:  Who received the offer
        status:     The current status of the offer (accept, declined, or pending)
        timestamp:  When the offer was sent.
    """
    owner = models.ForeignKey(
        get_user_model(), on_delete=models.CASCADE, related_name="sent_offers"
    )
    recipient = models.ForeignKey(
        get_user_model(), on_delete=models.CASCADE, related_name="received_offers"
    )

```

---

When you define a model in Django, the primary key field is automatically created with an auto-incrementing integer type unless you specify otherwise. The urls are defined in backend/users/urls.py as

---

```

urlpatterns = [
    ...
    path("api/offers/<int:pk>/", views.OfferDetail.as_view(), name="offer-detail"),
    ...
]

```

---

## 10.2 Black-Box

We can notice the incremental pattern in many IDs.

---

```

http://tdt4237-100.idi.ntnu.no/api/offers/1/
http://tdt4237-100.idi.ntnu.no/api/offers/2/
http://tdt4237-100.idi.ntnu.no/api/offers/.../
http://tdt4237-100.idi.ntnu.no/api/offers/N/

```

---

Look at figure 16

## 11 WSTG-SESS-06

A logout should have the function of invalidating the session cookies or access tokens, to ensure that it is not possible to log back in without having to enter the user's credentials. However, that is not the case in SecFit, where the same access token can be used indefinitely even after the user has logged out.

### 11.1 White-Box

A part of the problem lies in backend/secfit/settings.py in

---

```

SIMPLE_JWT = {
    ...
}

```

---

---

```
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
  {
    "url": "http://tdt4237-100.idi.ntnu.no/api/users/1/",
    "id": 1,
    "email": "admin@mail.com",
    "username": "admin",
    "athletes": [],
    "isCoach": false,
    "specialism": "",
    "coach": null,
    "workouts": [
      "http://tdt4237-100.idi.ntnu.no/api/workouts/1/"
    ],
    "coach_files": [],
    "athlete_files": []
  },
  {
    "url": "http://tdt4237-100.idi.ntnu.no/api/users/2/",
    "id": 2,
    "email": "efwef@effwe.com",
    "username": "HelloProva",
    "athletes": [],
    "isCoach": false,
    "specialism": "",
    "coach": "http://tdt4237-100.idi.ntnu.no/api/users/3/",
    "workouts": [],
    "coach_files": [],
    "athlete_files": []
  },
  {
    "url": "http://tdt4237-100.idi.ntnu.no/api/users/3/",
    "id": 3,
    "email": "wefwef@fweff.com",
    "username": "Prova",
    "athletes": [
      "http://tdt4237-100.idi.ntnu.no/api/users/2/",
      "http://tdt4237-100.idi.ntnu.no/api/users/4/"
    ],
    "isCoach": true,
    "specialism": "fe",
    "coach": null,
    "workouts": [],
    "coach_files": [],
    "athlete_files": []
  },
  {
    "url": "http://tdt4237-100.idi.ntnu.no/api/users/4/",
    "id": 4,
    "email": "wefwef@wefwef.com",
```

Figure 16: We can notice the IDs

---

where tokens should be invalidated by adding `'BLACKLIST_AFTER_ROTATION': True`,. Additionally, a view should handle logout and ensure the blacklisting of the access tokens during logout.

## 11.2 Black-Box

There are many different ways an attacker could gain access to a token. These ways, however, are not part of this vulnerability and we can assume the possibility of performing one that requires almost no hacking knowledge, but instead access to the victim's laptop (A more technical way to gain the access token is demonstrated in Section 18, though that vulnerability does not focus on the fact that the token remains valid but rather where it is stored).

Say the victim logs into the site and then leaves the laptop unlocked while going to the bathroom. The attacker can then use the victim's laptop and press F12 to open the developer's tools, navigate to Application and find the cookies or tokens. In this case, this can be found in local storage, as shown in Figure 17. The attacker then copies and saves these values on one of their own devices or could even just write them down on a piece of paper, close the developer's tools and leave undetected.

Once the victim comes back, there will be no trace of the attacker having been there. The victim resumes the browsing and will eventually log out. If the victim would now check the local storage, it would be empty.

The attacker can now – or even before, when the victim has not logged out yet – navigate to `http://tdt4237-100.idi.ntnu.no/home`, put the stolen values into their own browser's local storage, and refresh the page. As shown in Figure 18, the attacker is now logged in as the victim and has all the same access the victim has. It is now, for example, possible to access all private workouts and also set them as public.

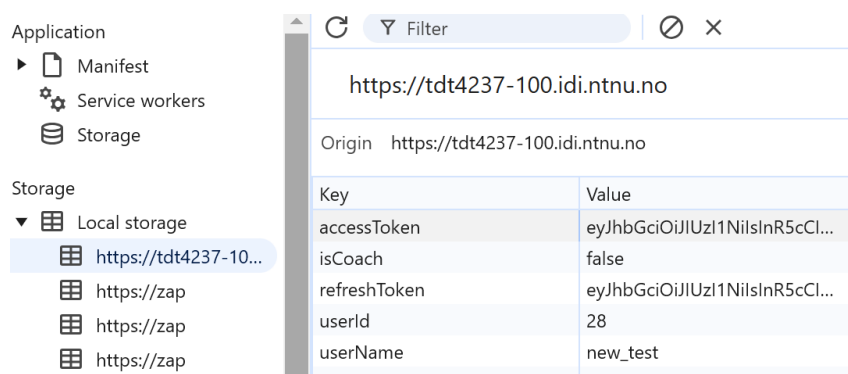


Figure 17: Local storage of victim's personal browser

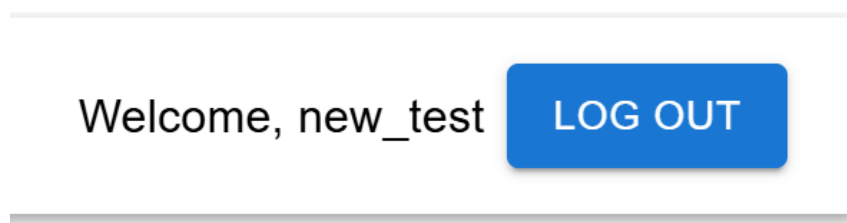


Figure 18: Display of successful login as new\_test

---

## 12 WSTG-SESS-07

To increase security, applications should have an idle session timeout and ensure that sessions do not stay active for too long. This is not done in SecFit.

### 12.1 White-Box

The lifetime for access and refresh tokens are specified in `backend/secfit/settings.py` to 3 and 60 days respectively:

---

```
SIMPLE_JWT = {
    'ACCESS_TOKEN_LIFETIME': timedelta(hours=72),
    'REFRESH_TOKEN_LIFETIME': timedelta(days=60),
    ...,
}
```

---

This is too long to handle them securely. Instead, the life time of the tokens should be reduced (for example to 1 hour for access and 7 days for refresh token).

## 13 WSTG-INPV-02

Athletes can post a workout privately, to their coach, or to everyone.

These workouts contain a comment section: This comment section is vulnerable to XSS.

### 13.1 White-Box

---

```
<form onSubmit={handleCommentSubmit(addComment)}>
  <Box>
    <h2>Comments</h2>
    {comments.length > 0 ? (
      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 dangerouslySetInnerHTML={{ __html: comment.content }}> </p>#!!! Here is
            the vulnerability
          <p>{getTimeSincePosted(comment.timestamp)}</p>
        </Paper>
      ))
    ) : (
      <p>This workout doesnt have any comments yet</p>
    )}
  </Box>
</Box>
```

---

---

```

<MyTextField
  label="Comment"
  name="comment"
  placeholder="Write a comment"
  type="string"
  control={commentControl}
></MyTextField>
{errorMessage && (
  <Box sx={{ color: "red", textAlign: "center" }}>{errorMessage}</Box>
)}
<MyButton label="Reply" type="submit" />
</Box>
</form>

```

---

`dangerouslySetInnerHTML` directly injects the `comment.content` as raw HTML into the DOM without checking.

Because of this, a script can be injected into the website. Anyone who clicks the workout can execute that code.

## 13.2 Black-Box

The exploiter once got to the workout page can comment with an html element.

The screenshot shows the 'Secfit' app interface. At the top, there's a navigation bar with 'HOME', 'WORKOUTS', 'EXERCISES', and 'ATHLETES'. A user named 'Prova' is logged in. The main form has several sections: 'Name' (with 'pictures' entered), 'Date/Time' (with '02/15/2025 01:00 AM'), 'Owner' (with 'Prova'), 'Visibility' (set to 'Private'), 'Exercises' (a dropdown menu showing 'Squats'), 'Sets' (a numeric input), and 'Number' (a numeric input). There's an 'ADD EXERCISE' button. Below this is a 'Notes' section with a text area containing 'fe'. At the bottom, there's a 'Comments' section with the message 'This workout doesn't have any comments yet'. A text input field for comments contains the HTML code: `<h1>This is an HTML element</h1>`. A 'REPLY' button is visible below the comment input.

Figure 19: HTML Element I'm writing

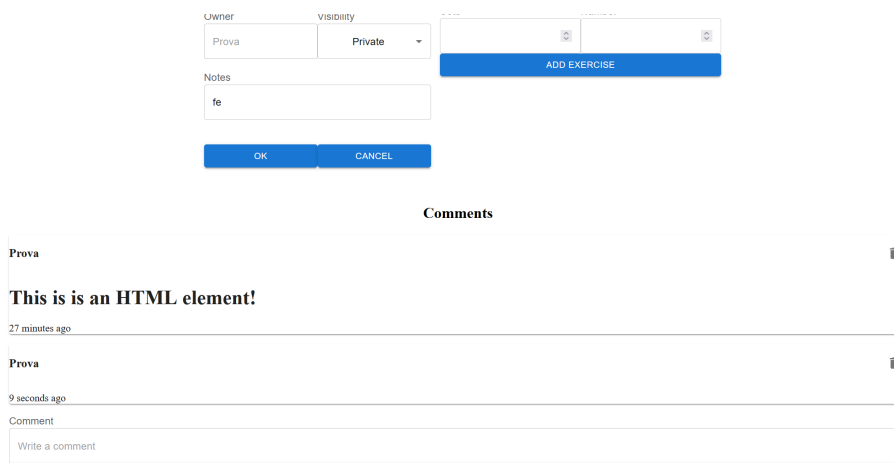


Figure 21: The script doesn't work! It doesn't show anything

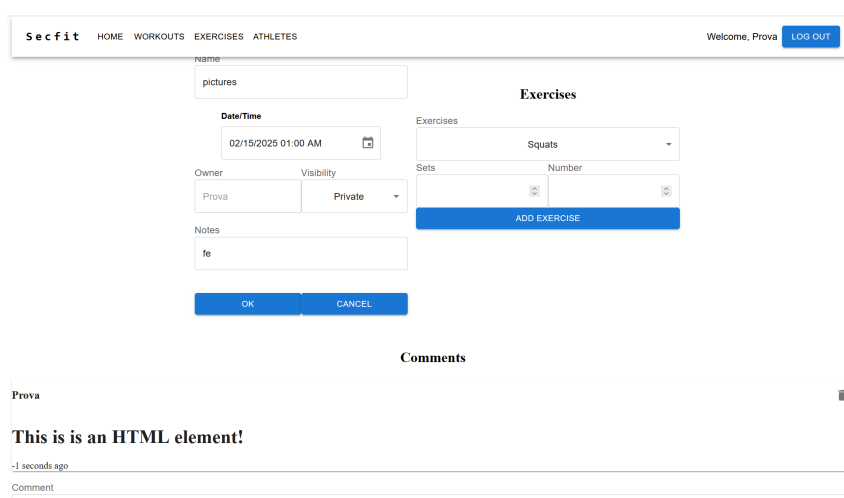


Figure 20: The result

If the exploiter try to comment a JavaScript code like this:

```
<script>alert("Test XSS")</script>
```

This cannot happen since `<script>` is blocked.

We can see that the exploiter can use the parameter "onerror" of the HTML element `<img>` to write a JavaScript function. This script will be stored. So, if anyone goes to the workout page, it will execute the JavaScript code automatically. (Look figure 21)  
This is the code:

```

```

## 14 WSTG-INPV-05

SQL injection is a very common security attack, and we could find an example of the vulnerability in this application. The vulnerability exists if the application constructs SQL queries using unvali-

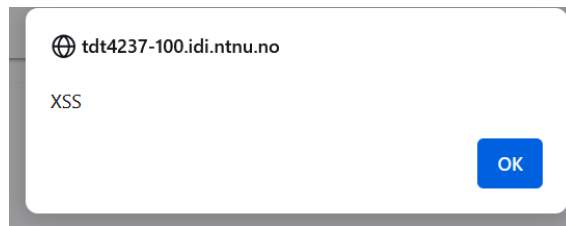


Figure 22: We can see that the browser is executing the JavaScript script

dated input, potentially allowing unauthorized access or data manipulation. This can be exploited by writing SQL code via user input or transmitted data to alter database operations.

## 14.1 White-Box

---

```
# /secfit/backend/users/views.py, line 80

#In class UserDetail
# (...)
query = f"SELECT * FROM users_user WHERE username = '{username}'"
with connection.cursor() as cursor:
    cursor.execute(query) # Executing the raw SQL query"
# (...)
```

---

Here we can see that the query is built by directly concatenating the value of "username" provided by the user, which means that there is a SQL injection vulnerability.

This is then used in the following urls:

---

```
#secfit/backend/users/urls.py

path("api/users/<int:pk>/", views.UserDetail.as_view(), name="user-detail"),
path("api/users/<str:username>/", views.UserDetail.as_view(), name="user-detail"),
```

---

## 15 WSTG-CRYP-03

It is very important to always keep in mind where critic information is stored or where it is sent through. We have already talked about credentials being sent over an insecure channel, but now we are going to address having sensitive information in the source code.

### 15.1 White-Box

---

```
# settings.py, line 26
# SECURITY WARNING: keep the secret key used in production secret!
SECRET_KEY = 'django-insecure-*)=v-+@_c-6(-60o%nv2b^a8br%$)%k+u+%(9ayozs79)abc'
```

---

Django uses the SECRET-KEY to sign cookies, hash passwords, and perform other cryptographic operations, to ensure the integrity and security of these processes. So this is sensitive information, which if compromised, attackers can potentially manipulate signed data, forge cookies, or break certain security protections. Then, it's best to generate the value randomly and store it in an environment variable and then retrieve it in settings.py

---

```
# Load secret key from environment variable
```

---

---

```
SECRET_KEY = os.getenv('DJANGO_SECRET_KEY', 'default-secret-key')
```

---

## 15.2 Black-Box

There are many ways of using the SECRET-KEY. Once exposed, the key may be used by attackers for session hijacking, password reset... We will present an example of impersonation or escalating privileges by forging a valid JWT 1. Capture the access token of a normal user (any, you can create it at the spot) by establishing OWASP ZAP as a proxy (see WSTG-ATHN-01 Black box testing step 1).

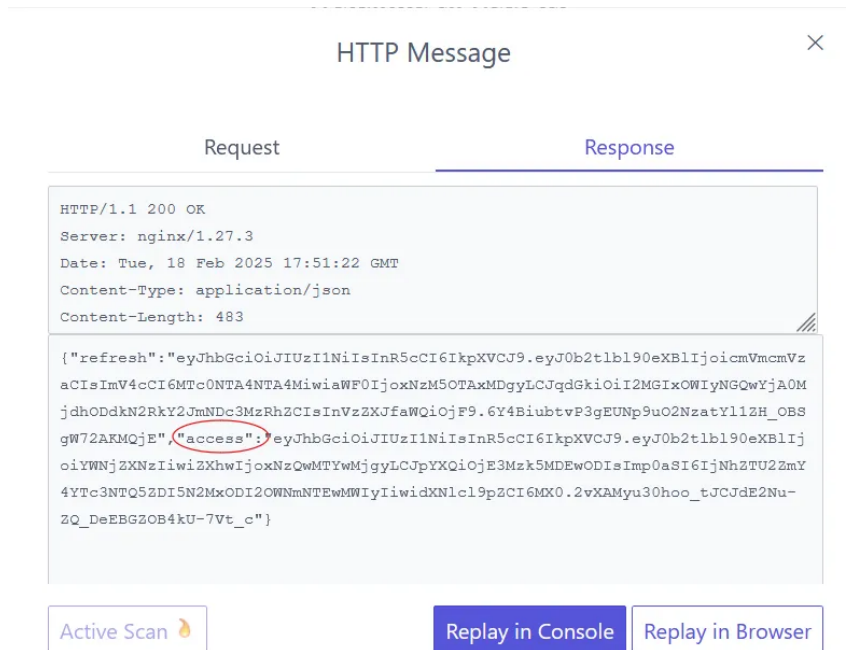
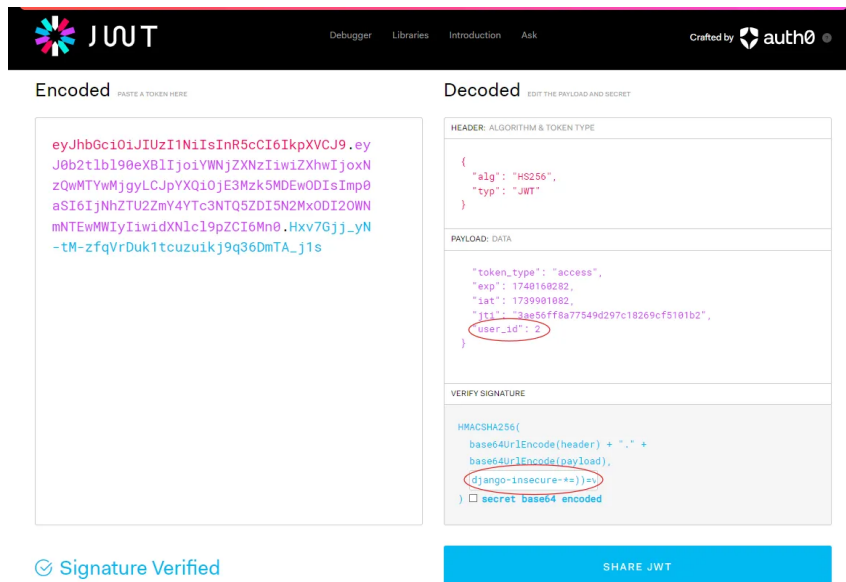


Figure 23: Capture response of login

2. Copy the key and use a tool like [jwt.io](https://jwt.io) to decode it. There you can modify the user id to the one that we want to change into and paste the SECRET KEY in the correspondent space.





3. Replace the original access token with the newly forged token and send the modified request to the server

Figure 25: Field where to insert the new token

## 16 WSTG-CRYP-04

---

For that reason, only hashed passwords should be stored and the hash method should be as save as possible. SecFit, however, performs lackluster hashing.

## 16.1 White-Box

In `backend/secfit/settings.py`, the password hasher is defined:

---

```
PASSWORD_HASHERS = [  
    'django.contrib.auth.hashers.UnsaltedSHA1PasswordHasher',  
]
```

---

This password hasher is known to be unsafe since it is possible to brute force the password and should be replaced with

`"django.contrib.auth.hashers.Argon2PasswordHasher"`.

## 16.2 Black-Box

In case the attacker gains access to the password storage, there are many options to crack the passwords, one of which being online tools, such as the SHA1 site by dCode.fr [3].

One of the retrieved hashed passwords could be `"177c2bd4dd4cb2fdcc041a38c61dff2db6fcfed4"`. This can then be input into the above mentioned website. After a few calculations, the website provides the output seen in Figure 26.

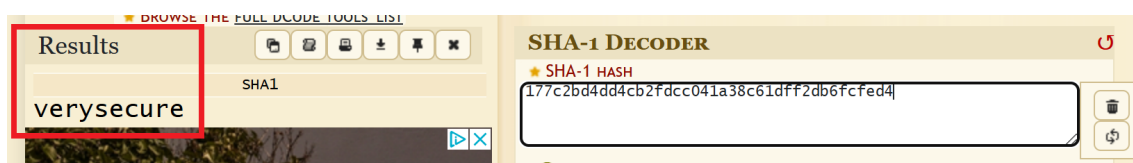


Figure 26: Result and corresponding hash code.

## 17 WSTG-CLNT-09

Clickjacking is a malicious technique where an attacker tricks a user into clicking on something different from what the user thinks they're clicking on. This is typically done by embedding a hidden or invisible frame (`iframe`) over a legitimate button or link on a website.

The login page of SecFit is vulnerable to clickjacking.

### 17.1 White-Box

(The login page of) SecFit is not protected against clickjacking, since the `XFrame` options are not disabled or restricted. Again, there are several places where this could have been implemented. One is `nginx/nginx.conf`, where depending on the desired outcome, the line `add_header X-Frame-Options "SAMEORIGIN"`; or `add_header X-Frame-Options "DENY"`; should have been added.

### 17.2 Black-Box

Using social engineering, e.g. via an email, the attacker can send the victim an url, which supposedly just lead to the login page of Secfit. However, this was done using an `iFrame` and the actual Login Button is overlayed with a fake one. This could be achieved using the following code:

---

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Clickjacking Exploit</title>
  <style>
    body {
      margin: 0;
      padding: 0;
      height: 100vh;
      display: flex;
      justify-content: center;
      align-items: center;
      position: relative;
      background-color: #f0f0f0;
    }

    /* Fake button */
    .fake-button {
      position: absolute;
      top: 72%;
      left: 50%;
      transform: translate(-50%, -50%);
      width: 300px;
      height: 50px;
      background-color: #4CAF50;
      color: white;
      text-align: center;
      line-height: 50px;
      font-size: 18px;
      border-radius: 5px;
      z-index: 100;
      cursor: pointer;
    }

    /* Invisible iframe (contains the login page) */
    iframe {
      width: 1200px;
      height: 800px;
      position: absolute;
      top: 50%;
      left: 50%;
      transform: translate(-50%, -50%);
      border: none;
      z-index: 1;
    }
  </style>
</head>
<body>

  <!-- Fake button -->
  <div class="fake-button" onclick="alert('You just sold your soul!');">Click to Log
    In</div>

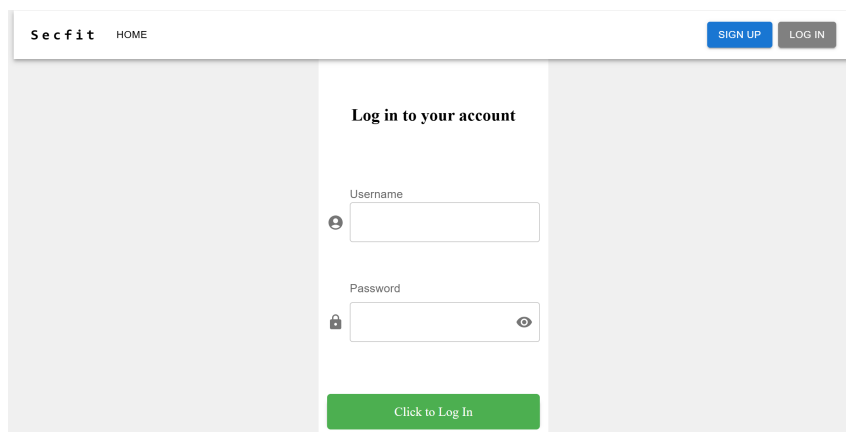
  <!-- Invisible iframe -->
  <iframe src="http://tdt4237-100.idi.ntnu.no/login" id="loginFrame"></iframe>

</body>
</html>
```

---

---

This results in the webpage displayed in Figure 27. Through clicking the fake login button, the victim triggers the (currently harmless) pop up in Figure 28.



The screenshot shows a web browser window with the URL 'Secfit HOME' in the top left. In the top right, there are two buttons: 'SIGN UP' (blue) and 'LOG IN' (grey). The main content area is titled 'Log in to your account'. Below this title, there are two input fields: 'Username' with a person icon and 'Password' with a lock icon and a toggle eye icon. At the bottom of the form is a green button labeled 'Click to Log In'.

Figure 27: Website with fake login button.

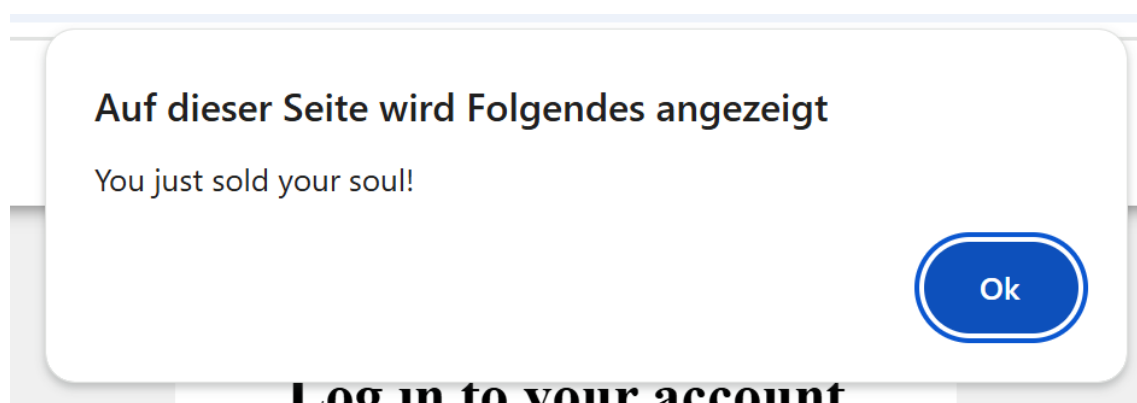


Figure 28: Alert after clicking login button, here displayed as green instead of the normal blue to show that it is different. In practice, it would be made to look exactly like the usual login button or invisible.

Instead of a harmless popup, the attacker can do a lot of damage and could, for example, have the fake button instead call a function that sends the credentials that the victim has input somewhere.

## 18 WSTG-CLNT-12

The website uses tokens to keep you logged in. In this way, you don't need to log in with your username and password every time you visit the website, since they are saved in the browser's local storage as long as you do not log out. This feature can be dangerous.

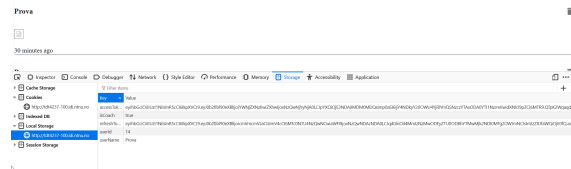


Figure 29: We can see them on Local Storage

## 18.1 White-Box

---

```
const setLocalAccessToken = (token) => {  
  localStorage.setItem("accessToken", token);  
};  
  
const removeLocalAccessToken = () => {  
  localStorage.removeItem("accessToken");  
};  
  
const getLocalRefreshToken = () => {  
  return localStorage.getItem("refreshToken");  
};  
  
const setLocalRefreshToken = (token) => {  
  localStorage.setItem("refreshToken", token);  
};
```

---

setLocalAccessToken and setLocalRefreshToken are the vulnerabilities. They set them into the local storage. Which is exposed to scripts.

## 18.2 Black-Box

There are several tokens that, together, can get you access to the account without using a username and password every time you visit the website.

For example the access token and the refresh token.

These tokens are in the local storage of the browser: This is a vulnerability, as a script can get access to that. In case of a malicious script, the account may be compromised.

As we saw in vulnerability number 13. There's an XSS vulnerability in the comment section of any workout. We can print the tokens with alert.

Look at figure 30

The code:

---

```

```

---

The attacker can send the malicious informations making a request with fetch. In this case we're using Discord webhook

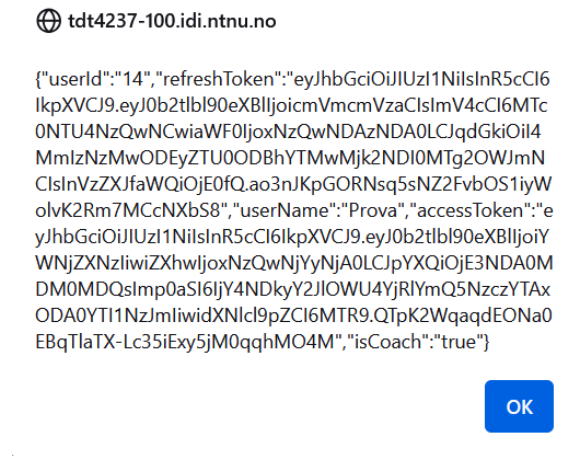


Figure 30: We can see that the local storage is printed

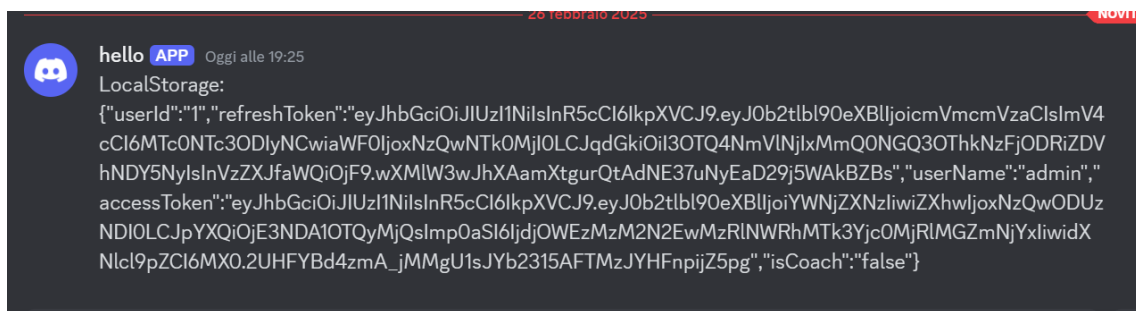


Figure 31: We can see that we received all the local storage

```

```

Look at figure 31

Now, with these informations, we can get access to the account that clicked on the workout thread.  
Look at figure 32 and 33

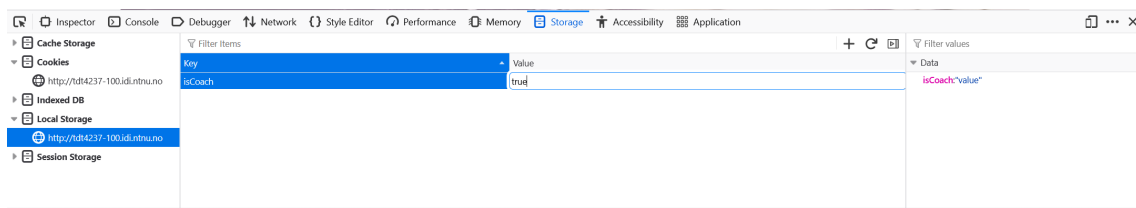


Figure 32: We can manually do this

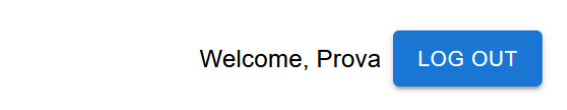


Figure 33: After inserting all the data and refreshing the page we get access to the account

## 19 WSTG-BUSL-05

There are no limits to the offers that an athlete can make. This may be a problem for the coach.

### 19.1 White-Box

---

```
class OfferList(
    mixins.ListModelMixin, mixins.CreateModelMixin, generics.GenericAPIView
):
    permission_classes = [IsAuthenticatedOrReadOnly]
    serializer_class = OfferSerializer

    def get(self, request, *args, **kwargs):
        return self.list(request, *args, **kwargs)

    def post(self, request, *args, **kwargs):
        serializer = self.get_serializer(data=request.data)
        serializer.is_valid(raise_exception=True)
        self.perform_create(serializer)
        headers = self.get_success_headers(serializer.data)
        return Response(serializer.data, status=status.HTTP_201_CREATED, headers=headers)
```

---

We can see there is no limit. Any user could send multiple offers.

### 19.2 Black-Box

Once we log in to the account. We can just go to "coach" and send more request to a single coach, therefore making it very hard for the coach to spot offers from other users.

Look figure 34

## 20 Conclusion

In summary, we can say that this is a very insecure application. Even though it does not handle actual valuable information (as it was probably created with educational purposes only), most of the data can be accessed and the application can be exploited in many different ways.

Though we already found a lot of vulnerabilities, it is quite possible that there are more and all of them should be addressed properly in order to make the application secure.

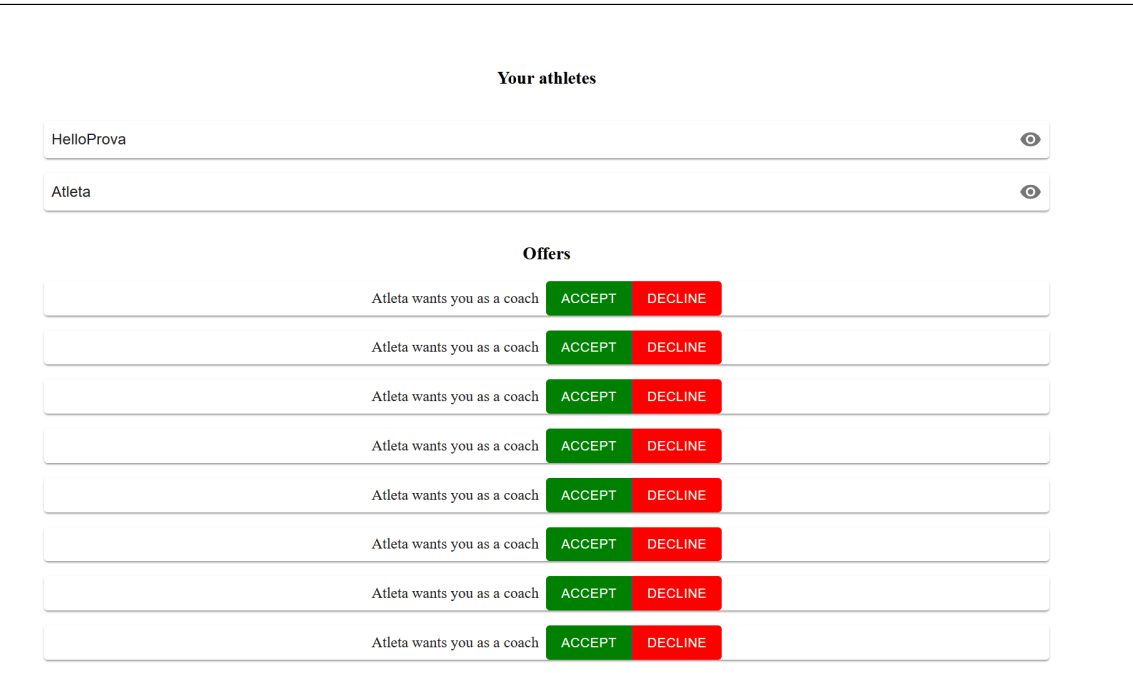


Figure 34: Even though Atleta is already an athlete of the coach. He can still send multiple offers

## References

[1] OWASP Foundation. *OWASP Web Security Testing Guide*. 4.2. OWASP Foundation, 2020. URL: <https://owasp.org/www-project-web-security-testing-guide/>.

[2] Daniel Miessler. *SecLists*. Accessed: 2025-02-06. 2021. URL: <https://github.com/danielmiessler/SecLists/Passwords>.

[3] dCode author. *SHA1 on dCode.fr*. Accessed: 2025-02-22. 2025. URL: <https://www.dcode.fr/sha1-hash>.