



DEPARTMENT OF COMPUTER SCIENCE

TDT4237 SOFTWARE SECURITY AND DATA PRIVACY

---

## Exercise 2. Finding Vulnerabilities

---

GROUP 21

*Authors:*

Patrick Øivind Helvik Legendre  
Gunnar Nystad  
Dag Kirstihagen

---

## Table of Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>WSTG-CLNT-09</b>	<b>1</b>
<b>3</b>	<b>WSTG-CRYP-03</b>	<b>2</b>
<b>4</b>	<b>WSTG-IDNT-02</b>	<b>3</b>
<b>5</b>	<b>WSTG-ATHN-07</b>	<b>3</b>
<b>6</b>	<b>WSTG-ATHN-02</b>	<b>4</b>
<b>7</b>	<b>WSTG-IDNT-05</b>	<b>5</b>
<b>8</b>	<b>WSTG-ATHN-03</b>	<b>6</b>
<b>9</b>	<b>WSTG-CRYP-04</b>	<b>7</b>
<b>10</b>	<b>WSTG-SESS-07</b>	<b>8</b>
<b>11</b>	<b>WSTG-ERRH-01</b>	<b>9</b>
<b>12</b>	<b>WSTG-ATHN-09</b>	<b>11</b>
<b>13</b>	<b>WSTG-SESS-06</b>	<b>13</b>
<b>14</b>	<b>WSTG-SESS-09</b>	<b>14</b>
<b>15</b>	<b>WSTG-BUSL-02</b>	<b>17</b>
<b>16</b>	<b>WSTG-BUSL-05</b>	<b>18</b>
<b>17</b>	<b>WSTG-BUSL-08</b>	<b>19</b>
<b>18</b>	<b>WSTG-BUSL-09</b>	<b>20</b>
<b>19</b>	<b>WSTG-ATHN-04</b>	<b>22</b>
<b>20</b>	<b>WSTG-ATHZ-02</b>	<b>24</b>
<b>21</b>	<b>Conclusion</b>	<b>25</b>

---

# 1 Introduction

Throughout the course, we have analyzed and exposed the security layout of the Trustedsitters website. In the previous exercise, we mapped out the page map of the application. This exercise will focus on finding any security vulnerabilities present in the application. We will use the OWASP Web Application Security Testing guide to categorize any vulnerabilities found and give a white- and black-box description of each.

## 2 WSTG-CLNT-09

### Testing for Clickjacking:

Trustedsitters is vulnerable to clickjacking. This attack attempts to trick users into clicking invisible web page elements or objects disguised as other elements. The attacker can, for example, load the webpage into an iframe, make it transparent and overlay objects to the site. When the victim clicks a button on the web page, they will click a transparent web page element.

### 2.1 White-Box

The backend sets the X-Frame-Options HTTP response header to 'DENY'. However, the browser can render the page in a frame, iframe, embed, or object since the frontend does not set the X-Frame-Options header. To fix this exploit, the following line of code should be included under 'http { server {' in nginx.conf:

```
add_header X-Frame-Options DENY;
```

---

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

MIDDLEWARE = [
    [...]
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]

// /nginx/nginx.conf

http {
    server {
```

---

### 2.2 Black-Box

To test for clickjacking, we put the link to the webpage into an iframe which successfully loaded as shown in Figure 1.

---

```
// test.html

<html>
  <head>
    <title>Clickjack test page</title>
  </head>
  <body>
    <iframe src="http://molde.idi.ntnu.no:21021" width="500" height="500"></iframe>
  </body>
</html>
```

---



Figure 1: Rendering Trustedsitters in an iframe.

### 3 WSTG-CRYP-03

#### Testing for Weak Transport Layer Security:

Sensitive data, e.g., information used in authentication, must be protected when it is transmitted through the network. It is considered a security risk if the application sends sensitive information via unencrypted channels like HTTP.

#### 3.1 White-Box

In settings.py, the following line should be implemented to redirect HTTP requests to HTTPS:

```
SECURE_SSL_REDIRECT = True
```

---

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

---

#### 3.2 Black-Box

We used Wireshark to analyze the network traffic as shown in Figure 2. Login credentials are sent unencrypted over HTTP, making them easy to access for an attacker listening to the network traffic, making a man-in-the-middle-attack possible.

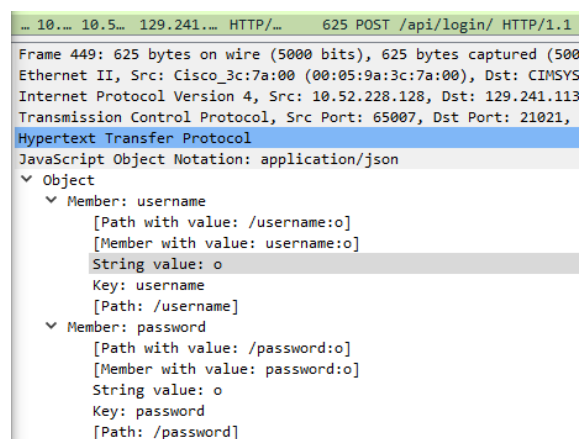


Figure 2: Using Wireshark to obtain username and password sent over HTTP.

---

## 4 WSTG-IDNT-02

### Test User Registration Process:

It is important that the identity requirements align with business and security requirements. In our opinion, this point is not met by the Trustedsitters website. On a website that connects parents and babysitters, both parties should be confident that the other party is who they pretend to be, especially since children are involved.

### 4.1 White-Box

In SignupForm.jsx line 16-31, shows that the application only asks for username, email and a password. The signup form should ask for more information from a user. Users of the site should be using their real names instead of username, emails should be unique, verification emails should not be easily guessable, and the users should ideally be using their mobile phone number in the registration process. The user's identity could be further authenticated using BankID.

---

```
// /frontend/src/components/SignupForm.jsx
```

```
const SignupForm = ({ setUser, setAppSnackbarOpen, setAppSnackbarText }) => {
  const history = useHistory();
  const [username, setUsername] = useState("");
  const [email, setEmail] = useState("");
  const [password, setPassword] = useState("");
  const [usernameErrorText, setUsernameErrorText] = useState("");
  const [passwordErrorText, setPasswordErrorText] = useState("");
  const [snackbarOpen, setSnackbarOpen] = useState(false);
  const [snackbarText, setSnackbarText] = useState("");
}
```

---

### 4.2 Black-Box

Anyone can register for access to the site, and they are automatically granted after verifying their email address. The same person can register multiple times using the same email address. The only proof of identity that is required for a registration to be successful is that the email must be verified. Nothing hinders identity information being faked. An example is shown in Figure 3.

```
POST http://molde.idi.ntnu.no:21021/api/register/ HTTP/1.1
Host: molde.idi.ntnu.no:21021
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:98.0
Accept: application/json, text/plain, */*
Accept-Language: en-GB,en;q=0.5
Content-Type: application/json
Content-Length: 64
Origin: https://molde.idi.ntnu.no:21021
Connection: keep-alive
Referer: https://molde.idi.ntnu.no:21021/signup
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin
{"username":"testuser","email":"test@test.no","password":"test"}
```

Figure 3: No identity verification.

## 5 WSTG-ATHN-07

### Testing for Weak Password Policy:

Passwords are an integral part of user authentication, and by not enforcing an adequate password policy, Trustedsitters are leaving their accounts vulnerable.

---

## 5.1 White-Box

Line 132-136 in settings.py shows the lacking password policy. The code only uses a built-in numeric password validator and doesn't impose any length or content requirements. This makes potential attacks especially vulnerable to brute force attacks.

---

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

AUTH_PASSWORD_VALIDATORS = [
    {
        'NAME': 'django.contrib.auth.password_validation.NumericPasswordValidator',
    },
]
```

---

## 5.2 Black-Box

We can exploit this vulnerability through a brute force attack of a known user's password. Since the application doesn't impose any length requirements, there will potentially be short passwords that can be exposed by brute-forcing every possible combination. As shown in Figure 4, we can register with a one-letter password, posing a significant security risk for the user's authentication.

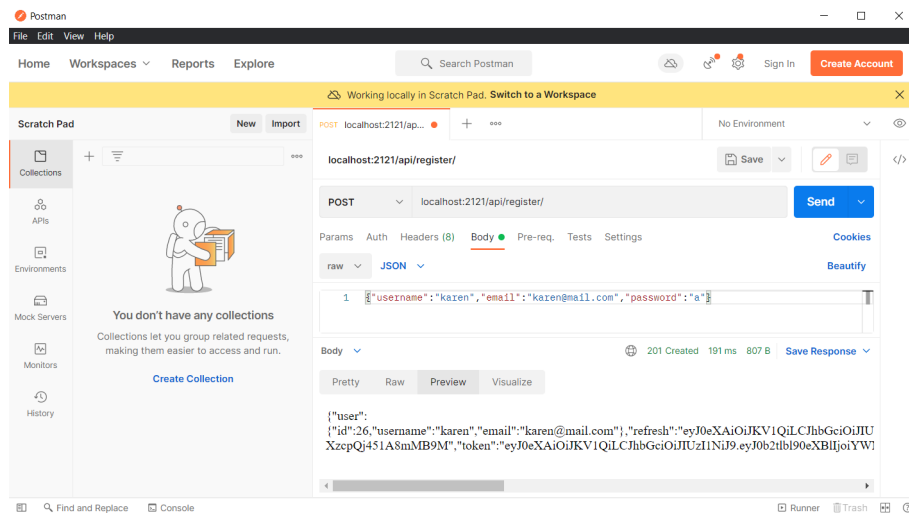


Figure 4: Accepted registration with "a" as password.

## 6 WSTG-ATHN-02

### Testing for Default Credentials:

When implementing openly available software, it's important to change the default passwords used. Even when the default password is changed, it should not be too generic and easily guessable.

## 6.1 White-Box

Line 7-9 in the .env file shows the default password for Django superusers which is fairly common and is therefore easily guessable.

---

```
// /trustedsitters/.env

DJANGO_SUPERUSER_PASSWORD=password123
```

---

---

```
DJANGO_SUPERUSER_USERNAME=admin123
DJANGO_SUPERUSER_EMAIL=admin@mail.com
```

---

## 6.2 Black-Box

Malicious actors could gain easy access to admin privileges by predicting the password of an admin account using the default credentials. The used passwords also show up in public password databases used to guess passwords [1]. An excerpt of these, including "password123" is shown in Figure 5.

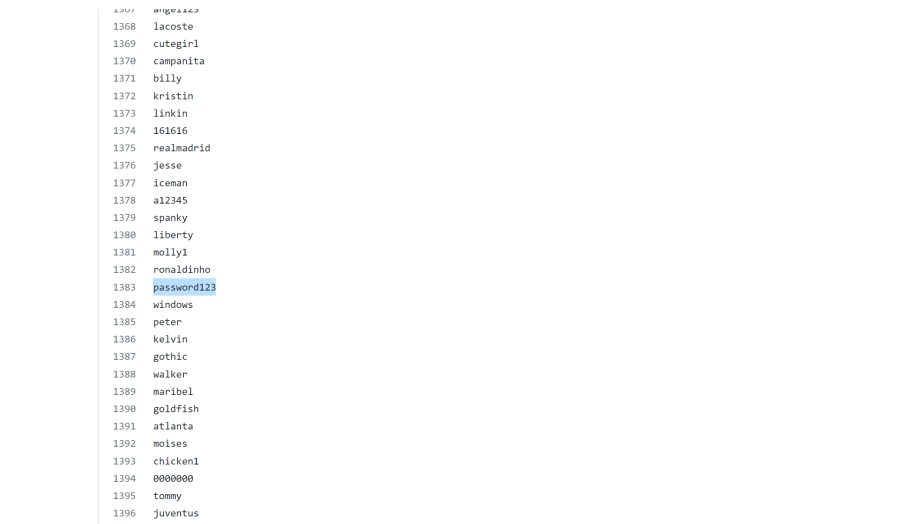


Figure 5: Publicly available password database exposing the default password used.

## 7 WSTG-IDNT-05

### Testing for Weak or Unenforced Username Policy:

Usernames are often highly structured, and it's necessary to implement a username policy to prevent easily guessable usernames.

### 7.1 White-Box

The username is unenforced in the user model or the serializer. Hence, the user can make easily guessable usernames. There should be a minimum restriction on the length of the username.

---

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

class User(AbstractUser):

// /trustedsitters/backend/apps/users/serializers.py

class RegisterSerializer(UserSerializer):
    password = serializers.CharField(
        max_length=128, min_length=1, write_only=True, required=True)
```

---

---

## 7.2 Black-Box

Malicious actors could use account information to guess the username easily. Additionally, exceptionally short usernames are easily cracked by brute force attacks. In Figure 6, we have shown a successful registration with a one-letter username, posing a significant security risk for the user's credentials.

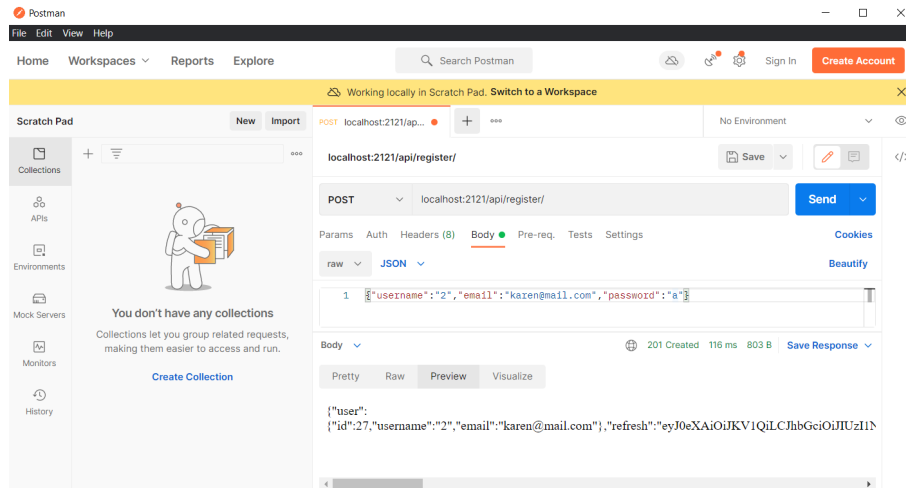


Figure 6: Accepted registration with "1" as username

## 8 WSTG-ATHN-03

### Testing for Weak Lock Out Mechanism:

If no lockout mechanism is implemented, malicious actors can test multiple passwords in a row as part of a brute force attack. If the user uses a common password, the credentials can easily be cracked by running through a dictionary of often-used passwords. If 2FA is enabled without a lockout mechanism, malicious actors can try every possible short 2FA code and access the account.

### 8.1 White-Box

Account lockout mechanisms are a very useful defense against brute force attacks. Lockout mechanisms block the user from making several incorrect login attempts in a row, thereby stopping malicious actors from testing every possible password or 2FA code quickly. The application lacks the code providing adequate lockout mechanisms. One way of fixing this issue is using the Django-axes plugin [2]. This plugin keeps track of failed login attempts and provides a simple block against brute-force attacks. A step-by-step fix to this problem is described below.

- Add axes to the list of INSTALLED\_APPS
- Add axes.backends.AxesBackend to the top of AUTHENTICATION\_BACKENDS
- Add axes.middleware.AxesMiddleware to your list of MIDDLEWARE

---

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

```
// /trustedsitters/backend/apps/users/views.py
class LoginViewSet(viewsets.ModelViewSet, TokenObtainPairView):
    serializer_class = LoginSerializer
    permission_classes = (AllowAny,)
    http_method_names = ['post']
```

---



---

```
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)
```

---

In the views.py code, it lacks any checks for how many times a user tries to login with his credentials.

## 8.2 Black-Box

In Figure 7, POST-requests are sent to `/api/login/` with incorrect password returning `401 Unauthorized`. After more than 20 failed login attempts, we log in using the correct password and are granted access.

13.03.22 12:40:36	POST	http://molde.idi.ntnu.no:21021/api/login/	401 Unauthorized
13.03.22 12:40:36	POST	http://molde.idi.ntnu.no:21021/api/login/	401 Unauthorized
13.03.22 12:40:37	POST	http://molde.idi.ntnu.no:21021/api/login/	401 Unauthorized
13.03.22 12:40:37	POST	http://molde.idi.ntnu.no:21021/api/login/	401 Unauthorized
13.03.22 12:40:37	POST	http://molde.idi.ntnu.no:21021/api/login/	401 Unauthorized
13.03.22 12:40:37	POST	http://molde.idi.ntnu.no:21021/api/login/	401 Unauthorized
13.03.22 12:40:37	POST	http://molde.idi.ntnu.no:21021/api/login/	401 Unauthorized
13.03.22 12:40:37	POST	http://molde.idi.ntnu.no:21021/api/login/	401 Unauthorized
13.03.22 12:40:37	POST	http://molde.idi.ntnu.no:21021/api/login/	401 Unauthorized
13.03.22 12:40:38	POST	http://molde.idi.ntnu.no:21021/api/login/	401 Unauthorized
13.03.22 12:40:38	POST	http://molde.idi.ntnu.no:21021/api/login/	401 Unauthorized
13.03.22 12:40:38	POST	http://molde.idi.ntnu.no:21021/api/login/	401 Unauthorized
13.03.22 12:40:38	POST	http://molde.idi.ntnu.no:21021/api/login/	401 Unauthorized
13.03.22 12:40:38	POST	http://molde.idi.ntnu.no:21021/api/login/	401 Unauthorized
13.03.22 12:40:38	POST	http://molde.idi.ntnu.no:21021/api/login/	401 Unauthorized
13.03.22 12:40:39	POST	http://molde.idi.ntnu.no:21021/api/login/	401 Unauthorized
13.03.22 12:40:39	POST	http://molde.idi.ntnu.no:21021/api/login/	401 Unauthorized
13.03.22 12:40:44	POST	http://molde.idi.ntnu.no:21021/api/login/	200 OK
13.03.22 12:40:44	GET	http://molde.idi.ntnu.no:21021/api/advert/	200 OK

Figure 7: Multiple failed login attempts without getting locked out.

## 9 WSTG-CRYP-04

### Testing for Weak Encryption:

Weak encryption used by internet applications can lead to sensitive data exposure, key leakage, broken authentication, insecure session, and spoofing attacks. Therefore it is of utmost importance to use secure encryption algorithms to protect, for example, passwords.

### 9.1 White-Box

In the settings.py file line 138-140, we can see that Trustedstitters uses an unsalted version of MD5. Using unsalted password hashers is not recommended. Additionally, MD5 hashing is not recommended as the MD5 algorithm is cryptographically broken and does not offer adequate protection.

---

```
// /trustedstitters/backend/turstedstitters/settings.py

PASSWORD_HASHERS = [
    'django.contrib.auth.hashers.UnsaltedMD5PasswordHasher',
]
```

---

---

## 9.2 Black-Box

Suppose some attacker manages to get hold of password data stored in the database of TrustedSitters. The attacker can easily decrypt the password without doing a lot of work because it uses the old deprecated MD5 hashing algorithm and does not use salt. We can see in Figure 8 the group managed to decrypt a password for an example account that was made using [3].

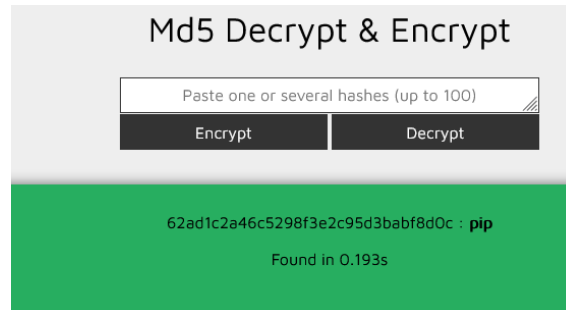


Figure 8: MD5 decryption for a hashed password in the database

## 10 WSTG-SESS-07

### Testing Session Timeout:

Session timeout is a tool to automatically log out the user after a certain amount of time. It is a crucial defense against malicious actors trying to guess and use a valid session ID from another user.

### 10.1 White-Box

As we can see in the settings.py file from lines 183-187, the session timeout is set to 15 days, which is way too long. Depending on the application's security requirements, a standard timeout window is between 15-60 minutes.

---

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

SIMPLE_JWT = {
    'ACCESS_TOKEN_LIFETIME': timedelta(minutes=6000),
    'REFRESH_TOKEN_LIFETIME': timedelta(days=15),
    'ROTATE_REFRESH_TOKENS': True,
}
```

---

### 10.2 Black-Box

A malicious actor can seize an access token or try to generate a valid access token for a user. We can see in this code sample that the access token lifetime validity is around 6000 minutes, and its refresh lifetime is 15 days. It is not recommended to have such an extended validity period because it gives the malicious actor a reasonable amount of time, to guess and eventually find a valid access token and then impersonate a legitimate user. We can see a user log in to TrustedSitters and having an access token in localStorage as shown in Figure 9.

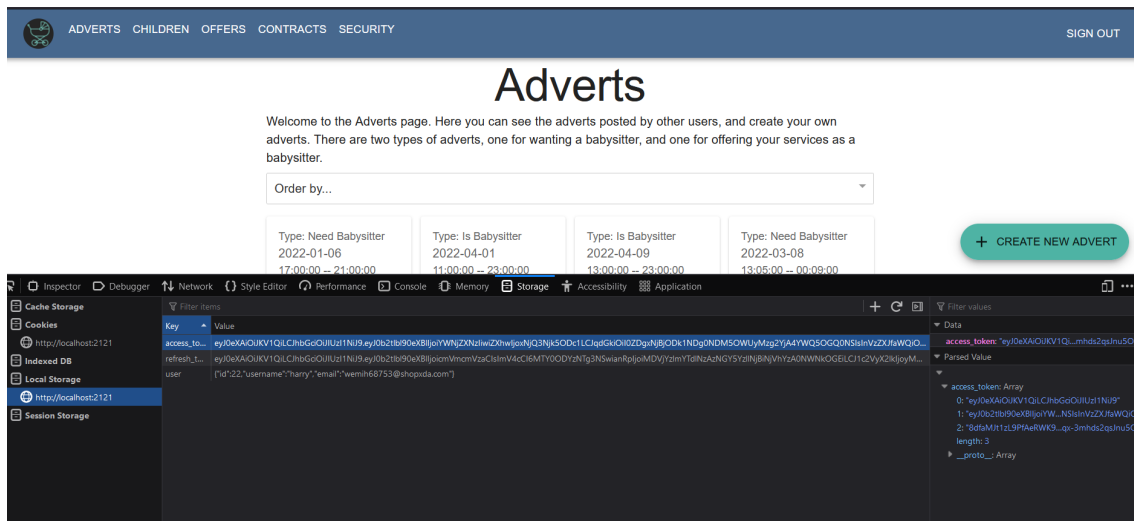


Figure 9: Browser localStorage access token.

A malicious user manages to get hold of the token and manages to log in the application and use it as the legitimate user in Figure 10.

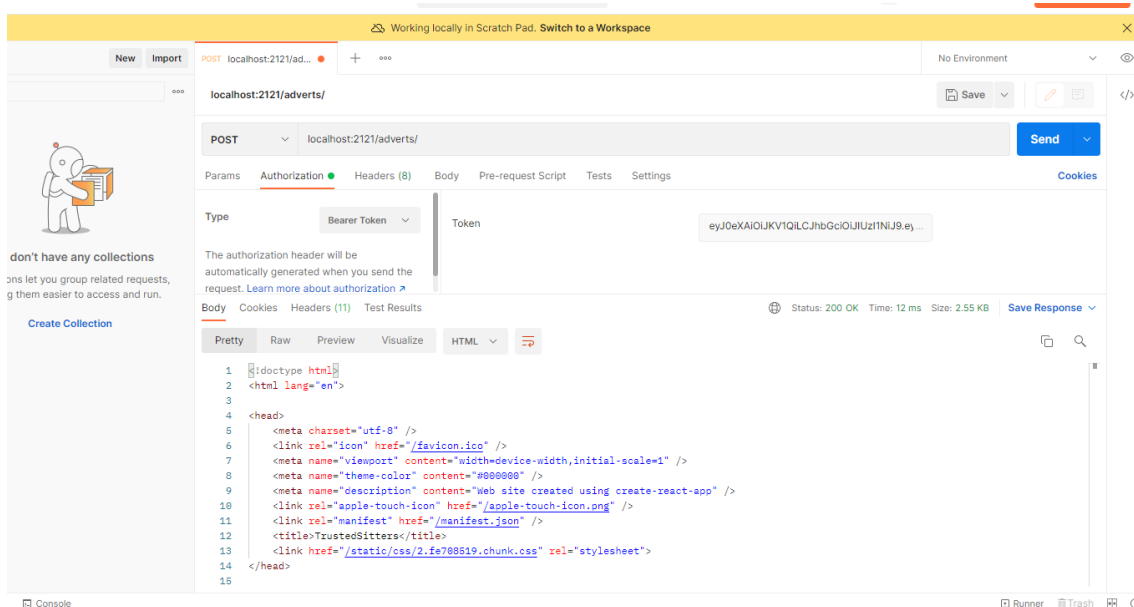


Figure 10: Postman showing the same access token used to log into the same account

## 11 WSTG-ERRH-01

### Testing for Improper Error Handling:

Errors can arise as stack traces, network timeouts, input mismatch, and memory dumps. Stack traces are not vulnerable by themselves but can disclose relevant information for an attacker.

#### 11.1 White-Box

In settings.py line 49, we see that the debugging mode is on. This is not recommended when deploying an application.

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

```
DEBUG = True
```

## 11.2 Black-Box

As we can observe in Figure 11, enabling the debug mode in Django leaks a lot of valuable data to a malicious actor. This data can be used to map out the whole website or even find vulnerabilities in the code.

### Page not found (404)

Request Method: GET

Request URL: http://localhost:2121/api/login

Using the URLconf defined in `trustedsitters.urls`, Django tried these URL patterns, in this order:

```
1. admin/
2. ^api/users/$ [name='users-list']
3. ^api/users/(?P<format>[a-z0-9]+)/?$ [name='users-list']
4. ^api/users/(?P<pk>[^.]+)/$ [name='users-detail']
5. ^api/users/(?P<pk>[^.]+)\.(?P<format>[a-z0-9]+)/?$ [name='users-detail']
6. ^api/register/$ [name='register-list']
7. ^api/register\.(?P<format>[a-z0-9]+)/?$ [name='register-list']
8. ^api/register/(?P<pk>[^.]+)/$ [name='register-detail']
9. ^api/register/(?P<pk>[^.]+)\.(?P<format>[a-z0-9]+)/?$ [name='register-detail']
10. ^api/login/$ [name='login-list']
11. ^api/login\.(?P<format>[a-z0-9]+)/?$ [name='login-list']
12. ^api/login/(?P<pk>[^.]+)/$ [name='login-detail']
13. ^api/login/(?P<pk>[^.]+)\.(?P<format>[a-z0-9]+)/?$ [name='login-detail']
14. ^api/refresh/$ [name='refresh-list']
15. ^api/refresh\.(?P<format>[a-z0-9]+)/?$ [name='refresh-list']
16. ^$ [name='api-root']
17. ^\.(?P<format>[a-z0-9]+)/?$ [name='api-root']
18. api/verify-email/<uid><status> [name='verify-email']
19. api/reset-password/<uidb64><token>/ [name='password-reset']
20. api/request-reset-password/ [name='password-reset-email']
21. api/reset-password-validate/ [name='password-reset-valid']
22. api/mfa/ [name='mfa']
23. ^api/adverts/needsitter/$ [name='needsitter-list']
24. ^api/adverts/needsitter\.(?P<format>[a-z0-9]+)/?$ [name='needsitter-list']
25. ^api/adverts/needsitter/(?P<pk>[^.]+)/$ [name='needsitter-detail']
26. ^api/adverts/needsitter/(?P<pk>[^.]+)\.(?P<format>[a-z0-9]+)/?$ [name='needsitter-detail']
27. ^api/adverts/issitter/$ [name='issitter-list']
28. ^api/adverts/issitter\.(?P<format>[a-z0-9]+)/?$ [name='issitter-list']
29. ^api/adverts/issitter/(?P<pk>[^.]+)/$ [name='issitter-detail']
30. ^api/adverts/issitter/(?P<pk>[^.]+)\.(?P<format>[a-z0-9]+)/?$ [name='issitter-detail']
31. ^api/adverts/$ [name='adverts-list']
32. ^api/adverts\.(?P<format>[a-z0-9]+)/?$ [name='adverts-list']
33. ^api/adverts/(?P<pk>[^.]+)/$ [name='adverts-detail']
34. ^api/adverts/(?P<pk>[^.]+)\.(?P<format>[a-z0-9]+)/?$ [name='adverts-detail']
35. ^$ [name='api-root']
36. ^\.(?P<format>[a-z0-9]+)/?$ [name='api-root']
37. ^api/children/$ [name='children-list']
38. ^api/children\.(?P<format>[a-z0-9]+)/?$ [name='children-list']
39. ^api/children/(?P<pk>[^.]+)/$ [name='children-detail']
40. ^api/children/(?P<pk>[^.]+)\.(?P<format>[a-z0-9]+)/?$ [name='children-detail']
41. ^api/child-file/$ [name='child-file-list']
42. ^api/child-file\.(?P<format>[a-z0-9]+)/?$ [name='child-file-list']
43. ^api/child-file/(?P<pk>[^.]+)/$ [name='child-file-detail']
44. ^api/child-file/(?P<pk>[^.]+)\.(?P<format>[a-z0-9]+)/?$ [name='child-file-detail']
45. ^$ [name='api-root']
46. ^\.(?P<format>[a-z0-9]+)/?$ [name='api-root']
47. api/child-file-download/<int:pk>/ [name='child-file-download']
48. api/remove-guardian/ [name='remove-guardian']
49. api/active-contract-children/ [name='active-contract']
50. ^api/offers/$ [name='offers-list']
51. ^api/offers\.(?P<format>[a-z0-9]+)/?$ [name='offers-list']
52. ^api/offers/(?P<pk>[^.]+)/$ [name='offers-detail']
53. ^api/offers/(?P<pk>[^.]+)\.(?P<format>[a-z0-9]+)/?$ [name='offers-detail']
54. ^api/contracts/$ [name='contracts-list']
55. ^api/contracts\.(?P<format>[a-z0-9]+)/?$ [name='contracts-list']
56. ^api/contracts/(?P<pk>[^.]+)/$ [name='contracts-detail']
57. ^api/contracts/(?P<pk>[^.]+)\.(?P<format>[a-z0-9]+)/?$ [name='contracts-detail']
58. ^$ [name='api-root']
59. ^\.(?P<format>[a-z0-9]+)/?$ [name='api-root']
60. api/offer_answer/ [name='offer_answer']
61. api/finish_contract/ [name='finish_contract']
```

The current path, `api/login`, didn't match any of these.

You're seeing this error because you have `DEBUG = True` in your Django settings file. Change that to `False` and Django will display a standard 404 page

Figure 11: Error message from Django when requesting a page that does not exist

---

## 12 WSTG-ATHN-09

### Testing for Weak Password Change or Reset Functionalities:

The password change and reset functionality allow a user to change their password without the help of an administrator. If these services are not secure, malicious actors can take unauthorized control over the user's account.

#### 12.1 White-Box

In views.py line 155-181, we can see that creating the uid and the token in the password reset URL sent by mail. We can see that the uid results from the primary key for the user being base64 encoded. And the primary key for the user is not randomly generated; it is incremented sequentially for each user registering. This presents an easy task for a malicious actor to guess the primary key.

---

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

class PasswordResetEmailView(generics.GenericAPIView):
    serializer_class = ResetPasswordSerializer
    def post(self, request):
        if request.data.get("email") and request.data.get("username"):
            email = request.data["email"]
            username = request.data["username"]

            if get_user_model().objects.filter(email=email, username=username).exists():
                user = get_user_model().objects.get(email=email, username=username)

                uid = urlsafe_base64_encode(force_bytes(user.pk))
                domain = get_current_site(request).domain
                token = urlsafe_base64_encode(force_bytes(user.username))
                link = reverse(
                    'password-reset', kwargs={"uidb64": uid, "token": token})

                url = f"{settings.PROTOCOL}://{domain}{link}"
                email_subject = "Password reset"
                mail = EmailMessage(
                    email_subject,
                    url,
                    None,
                    [email],
                )
                mail.send(fail_silently=False)
            return Response({'success': "If the user exists, you will shortly receive a
                link to reset your password."}, status=status.HTTP_200_OK)
```

---

#### 12.2 Black-Box

We created a new user with primary key 22, and we base64 encode with a website used for encoding and decoding base64 [4]. From this, we get MjI illustrated in Figure 12. This is our uid in our link. Then we base64 encode the username of the user, which in this case is "harry" and we base64 encode that too, and this gives us aGFycnk as shown in Figure 13. From this, we know the reset password URL for this user, which is localhost:2121/api/reset-password/MjI/aGFycnk/ illustrated in Figure 14.

---

22

**i** To encode binaries (like images, documents, etc.) use the file upload form a little further down on this page.

UTF-8

▼

Destination character set.

LF (Unix)


▼

Destination newline separator.

☐ Encode each line separately (useful for when you have multiple entries).

☐ Split lines into 76 character wide chunks (useful for MIME).

☐ Perform URL-safe encoding (uses Base64URL format).

 Live mode OFF

Encodes in real-time as you type or paste (supports only the UTF-8 character set).

> ENCODE <

Encodes your data into the area below.

MjI=

Figure 12: Base64 encode 22 that gives MjI as result

harry

**i** To encode binaries (like images, documents, etc.) use the file upload form a little further down on this page.

UTF-8

▼

Destination character set.

LF (Unix)


▼

Destination newline separator.

☐ Encode each line separately (useful for when you have multiple entries).

☐ Split lines into 76 character wide chunks (useful for MIME).

☐ Perform URL-safe encoding (uses Base64URL format).

 Live mode OFF

Encodes in real-time as you type or paste (supports only the UTF-8 character set).

> ENCODE <

Encodes your data into the area below.

aGFycnk=

Figure 13: Base64 encode harry that gives aGFycnk as result

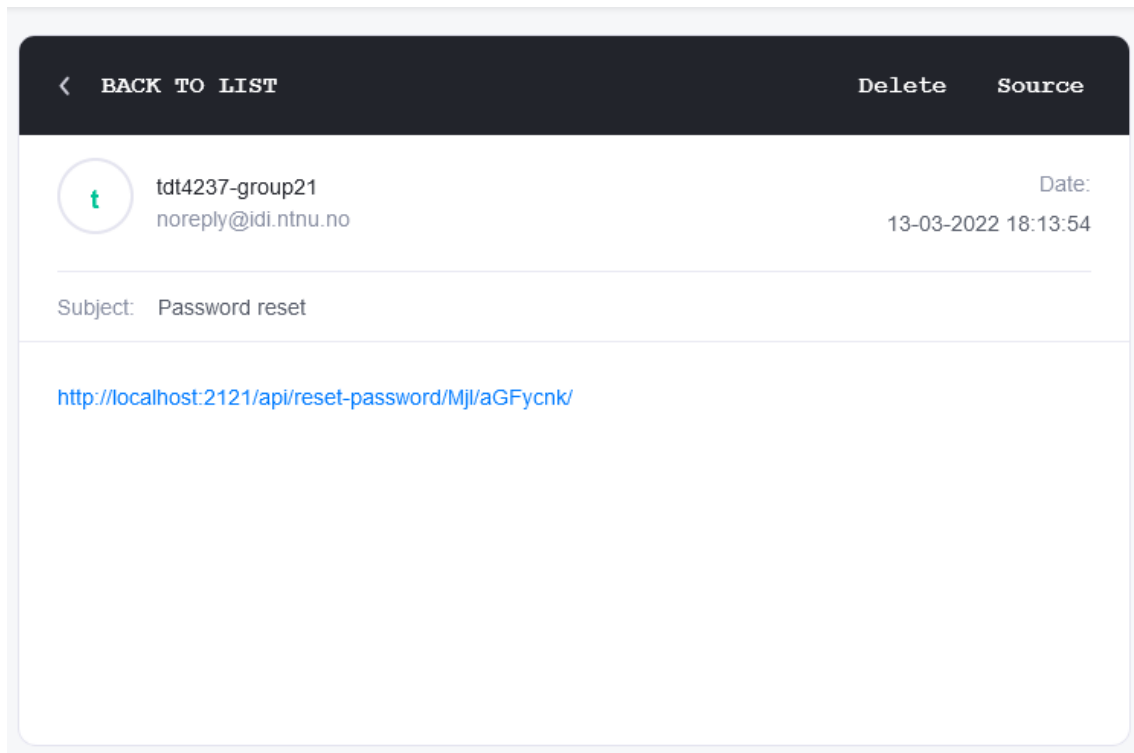


Figure 14: Password reset link send by email

## 13 WSTG-SESS-06

### Testing for Logout Functionality:

Logout functionality is an integral tool for reducing the lifetime of session tokens. Without a logout functionality, the session tokens are alive for the full duration of the specified token lifetime.

### 13.1 White-Box

In App.jsx in line 28-45, we see that the only log-out functionality is implemented in the frontend. This is not recommended because a malicious actor can bypass the authentication process in the backend. The program only clears the localStorage when logging out in the frontend of the application, meaning it does not log out the user. This leads to the possibility to log in to the same account on two separate browser sessions without the legitimate user being aware of it. The website should log in the database if a user is logged in or not, so only one user on session can use the website.

```
// /trustedsitters/frontend/src/App.jsx
```

```
const App = () => {
  const [user, setUser] = useState(null);
  const [snackbarOpen, setSnackbarOpen] = useState(false);
  const [snackbarText, setSnackbarText] = useState("");

  const signOut = () => {
    window.localStorage.removeItem("user");
    window.localStorage.removeItem("refresh_token");
    window.localStorage.removeItem("access_token");
    setUser(null);
  };
};
```

---

```
const handleClose = (event, reason) => {  
  if (reason === "clickaway") {  
    return;  
  }  
  setSnackbarOpen(false);  
};
```

---

## 13.2 Black-Box

We can exploit this vulnerability by logging into one browser session with one account as seen in Figure 15. Then a malicious user can log in with the same credentials and access the website at the same time as the legitimate user in Figure 16.

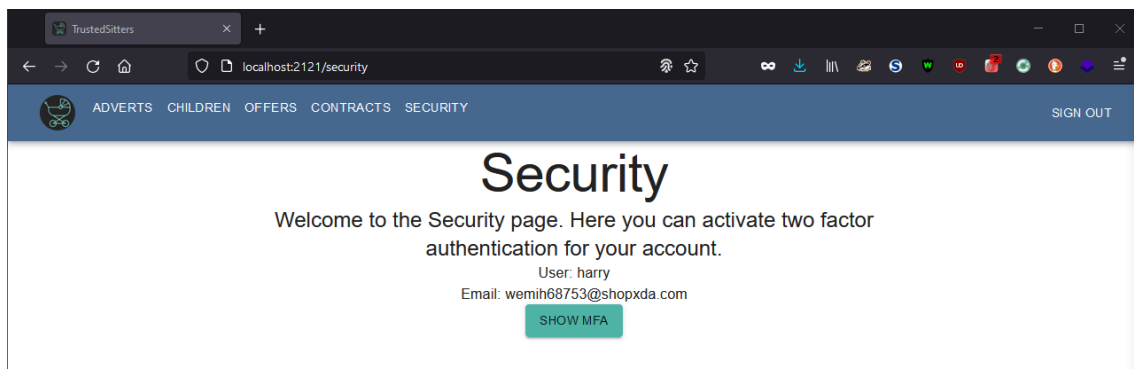


Figure 15: harry account logged in first session

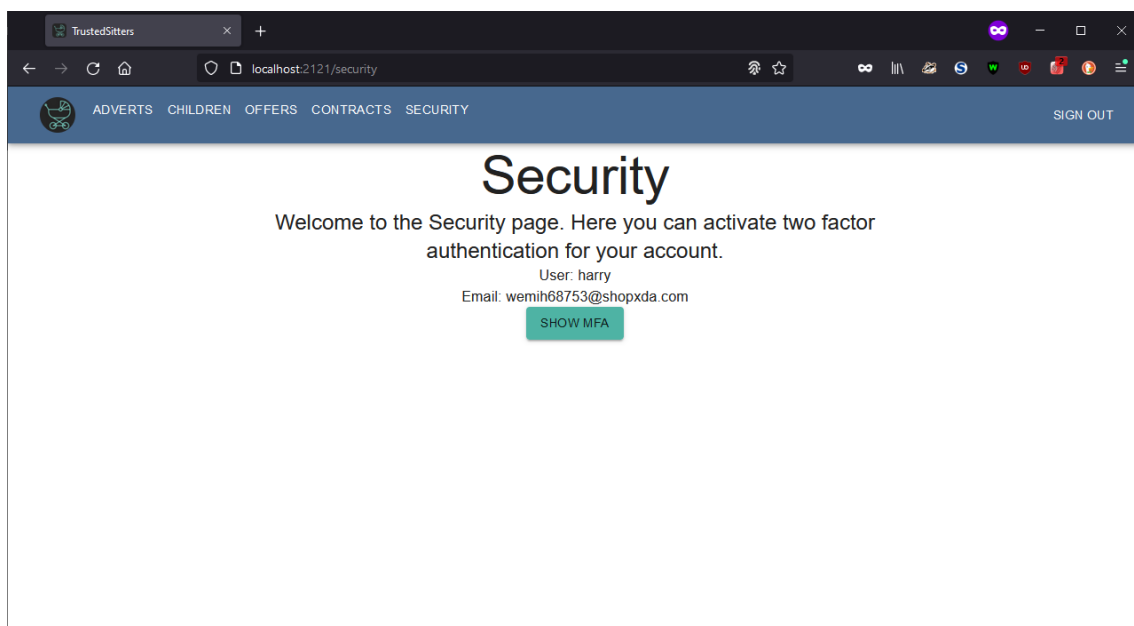


Figure 16: harry account logged in a second session

## 14 WSTG-SESS-09

### Testing for Session Hijacking:

Session hijacking is when malicious actors impersonate other users by getting control of their cookies



---

and presenting them to the website. This can occur when the user's cookies are not transported over HTTPS.

## 14.1 White-Box

In App.jsx lines 28-45, we can see the lack of code that handles the sign-in and sign-out process an attacker could hijack the session. In UserSerializer lines 18-41, we can see that there are no checks implemented to see if a token is already in use by a user.

---

```
// /trustedsitters/frontend/src/App.jsx
```

```
const App = () => {
  const [user, setUser] = useState(null);
  const [snackbarOpen, setSnackbarOpen] = useState(false);
  const [snackbarText, setSnackbarText] = useState("");

  const signOut = () => {
    window.localStorage.removeItem("user");
    window.localStorage.removeItem("refresh_token");
    window.localStorage.removeItem("access_token");
    setUser(null);
  };

  const handleClose = (event, reason) => {
    if (reason === "clickaway") {
      return;
    }
    setSnackbarOpen(false);
  };
};
```

---

```
// /trustedsitters/backend/apps/users/serializer.py
```

```
class UserSerializer(serializers.ModelSerializer):

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

class LoginSerializer(TokenObtainPairSerializer):

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

        refresh = self.get_token(self.user)

        data['user'] = UserSerializer(self.user).data
        data['refresh'] = str(refresh)
        data['access'] = str(refresh.access_token)
        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
```

---

## 14.2 Black-Box

First by looking at the general post request we need to login to the Trustedsitters, in the request we find out we only need 1 request shown in Figure 17.



67	Proxy	13.03.2022, 20:35:34	POST	http://localhost:2121/api/login/	200	OK	124 ms	531 bytes	JSON
70	Proxy	13.03.2022, 20:35:34	GET	http://localhost:2121/api/adverts/	200	OK	17 ms	1 025 bytes	JSON
73	Proxy	13.03.2022, 20:41:41	GET	https://brave.safesite.com/oc/mozilla.com/vt/brave/	200	OK	67 ms	50 406 bytes	HTML

Figure 17: Log in post request

And we need to provide in the request both the password and the username as shown in Figure 18.

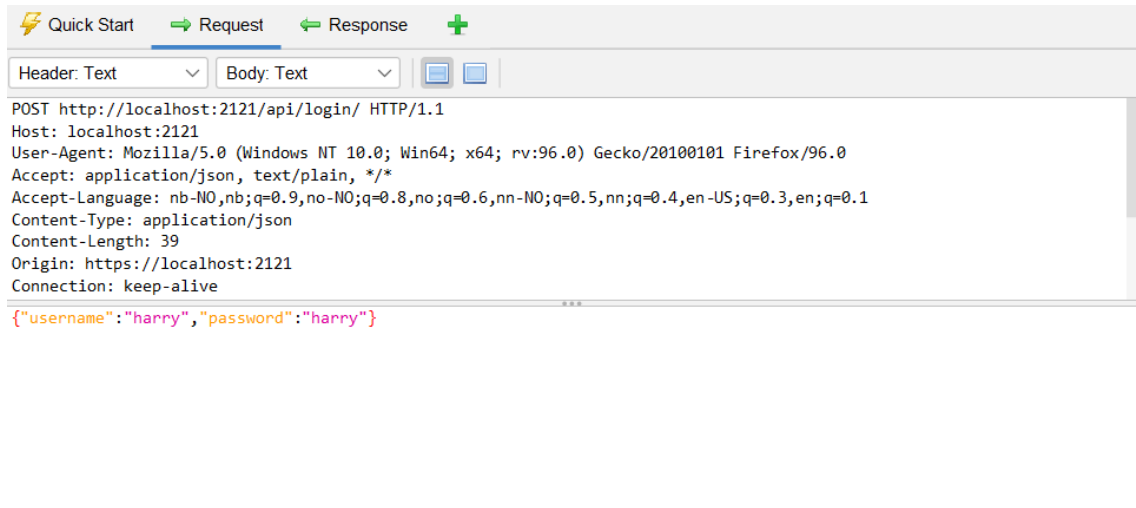


Figure 18: Post request that supplies username and password

In the Figure 19 we receive the access token and because it does not verify that this token is already in use by a logged user. It does not have sufficient log in mechanism to check if a user is logged in.

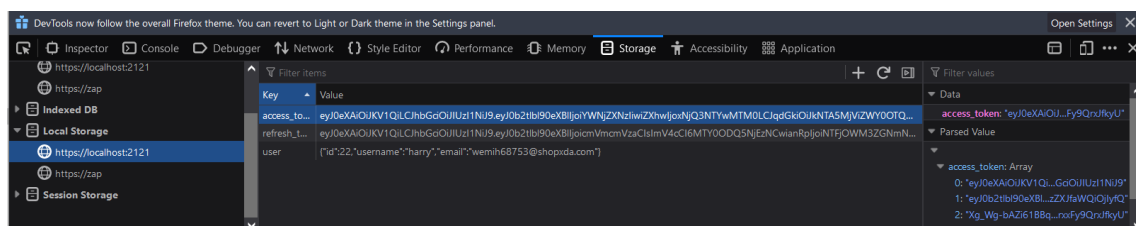


Figure 19: Browser Localstorage session token

If a malicious user manages to the localstorage of the browser by a phishing attack it can use the same token of our user to login to the system as our other user. See figure 20.

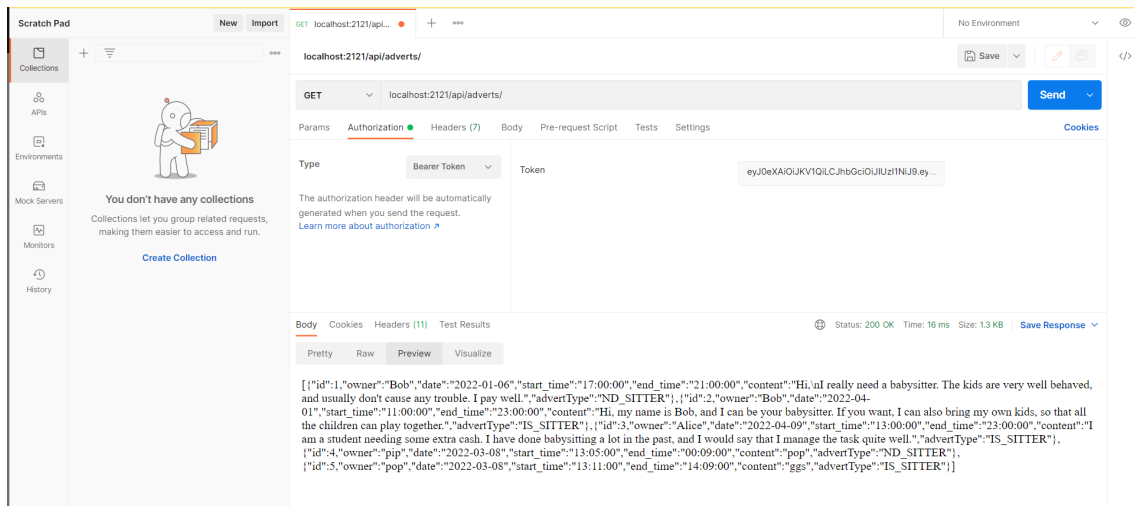


Figure 20: Postman Logging in with the same token that our other user is logged in with

## 15 WSTG-BUSL-02

### Test Ability to Forge Requests:

Malicious actors can forge requests by sending information directly to the backend processing and thereby circumventing the frontend GUI. This is achieved by using HTTP GET / POST requests, which contains data that should not be allowed by the applications business logic implemented in the frontend. TrustedSitters does not guard itself against this and leaves itself vulnerable to users impersonating each other.

### 15.1 White-Box

In `/offers/views.py` lines 79-97, there is not check whether the request is sent from the user that has received the offer. Hence, an actor with malicious intents can answer the offer on behalf of other users.

---

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

```
class AnswerOfferView(generics.GenericAPIView):

    def post(self, request):

        if request.data.get('offerId') and request.data.get('status'):

            offer = Offer.objects.get(id=request.data['offerId'])
            state = request.data['status']

            if offer.status != OfferStatus.PENDING:
                return Response({'success': False, 'message': 'Offer already answered'},
                                status=status.HTTP_400_BAD_REQUEST)

            if state == 'D': # Declined offer
                offer.status = OfferStatus.DECLINED
                offer.save()
                return Response({'success': True, 'message': 'Offer declined'},
                                status=status.HTTP_200_OK)

            if state == 'A': # Accepted offer
                offer.status = OfferStatus.ACCEPTED
```

---

---

## 15.2 Black-Box

To exploit this vulnerability we sent a POST request to `/api/offer_answer/` using Postman. A user can accept offers on behalf of other using as shown in Figure 21.

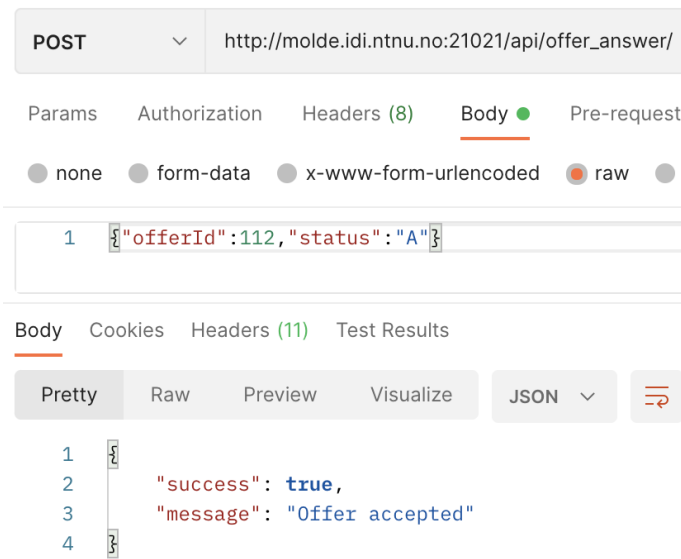


Figure 21: Using Postman to accept an offer.

## 16 WSTG-BUSL-05

### Test Number of Times a Function Can Be Used Limits:

If functions are allowed to be used more times than intended, malicious actors may be able to circumvent the application's business logic. TrustedSitters does not adequately guard against this exploit, e.g., users can send multiple job offers for the same advert.

### 16.1 White-Box

In `/offers/views.py` lines 24-37, the function creating offers does not check whether or not an offer from the user is already created. This lets user spam job offers for the same advert, as well as spamming guardian offers to the same user.

---

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

def perform_create(self, serializer):
    if self.request.data.get('recipient'):
        result = get_user_model().objects.raw(
            "SELECT * from users_user WHERE username = '%s'" %
            self.request.data['recipient'])

        if self.request.user.username == self.request.data['recipient'].strip():
            raise ValidationError("Cannot send offer to yourself")

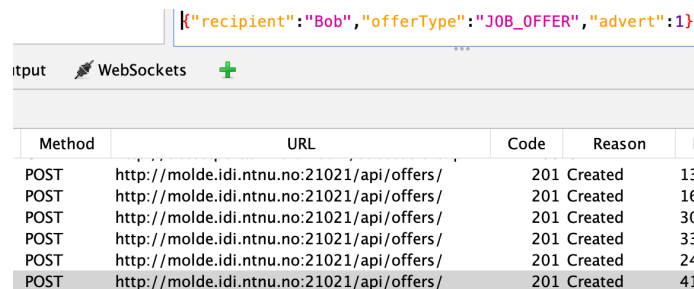
        if len(result) > 0: # Check if a user exist with the given username
            r = ""
            for u in result:
                r += u.username + " "
            serializer.save(
                recipient=r.strip(), sender=self.request.user.username)
```

---

---

## 16.2 Black-Box

A user can send job offers for the same advert an unlimited number of times as shown in Figure 22.



Method	URL	Code	Reason	
POST	http://molde.idi.ntnu.no:21021/api/offers/	201	Created	13
POST	http://molde.idi.ntnu.no:21021/api/offers/	201	Created	16
POST	http://molde.idi.ntnu.no:21021/api/offers/	201	Created	30
POST	http://molde.idi.ntnu.no:21021/api/offers/	201	Created	33
POST	http://molde.idi.ntnu.no:21021/api/offers/	201	Created	24
POST	http://molde.idi.ntnu.no:21021/api/offers/	201	Created	41

Figure 22: Sending job offers for the same advert multiple times with no errors.

## 17 WSTG-BUSL-08

### Test Upload of Unexpected File Types:

Many applications allow files to be uploaded and to process and act upon the files uploaded. Therefore, it is essential to check that the files uploaded are in the expected format to prevent errors.

### 17.1 White-Box

In models.py line 31-39. In the creation of the validator class, we can see that the allowed mimetypes is equal to "", allowed extensions is equal to "" and max size is equal to infinity. This should not be the case, instead it should have included the files types that should be rejected and a non max size which is not equal to infinity.

---

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

class ChildFile(models.Model):

    child = models.ForeignKey(
        Child, on_delete=models.CASCADE, blank=False, related_name='children_files')

    file = models.FileField(upload_to=child_directory_path,
                           blank=False, validators=[FileValidator(allowed_mimetypes='',
                           allowed_extensions='', max_size=math.inf)])

    content_type = models.CharField(max_length=64)

// /trustedsitters/backend/apps/children/validators.py

def __init__(self, *args, **kwargs):
    self.allowed_extensions = kwargs.pop('allowed_extensions', None)
    self.allowed_mimetypes = kwargs.pop('allowed_mimetypes', None)
    self.min_size = kwargs.pop('min_size', 0)
    self.max_size = kwargs.pop('max_size', None)
```

---

In the description of the site, it says that parents can upload pdf and png files for their childs. However, the site does not restrict which the type of files a user can upload.

---

## 17.2 Black-Box

In the description of the website, it says that parents can upload pdf and png files for their children. However, we can upload other file types as shown in Figure 23.

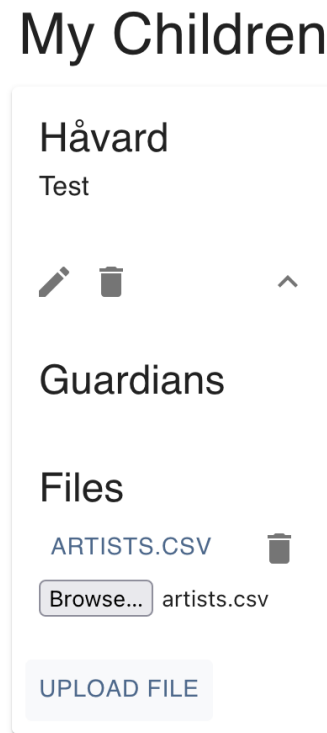


Figure 23: Uploading a .csv file.

## 18 WSTG-BUSL-09

### Test Upload of Malicious Files:

Many applications allow files to be uploaded and for the application to act upon the contents of the file. The problem arises when malicious actors use these file uploads to achieve their goals. Trustedsitters does not guard against this and allows the user to for example upload a webshell to compromise the website.

### 18.1 White-Box

In models.py line 31-39. There is no check in the creation of the validator, to check for file extensions, to check the name of the file and there is not limit on the size of file. This can be exploited to upload malware and even manage to execute it on the webserver.

---

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

```
class ChildFile(models.Model):

    child = models.ForeignKey(
        Child, on_delete=models.CASCADE, blank=False, related_name='children_files')

    file = models.FileField(upload_to=child_directory_path,
                           blank=False, validators=[FileValidator(allowed_mimetypes='',
                           allowed_extensions='', max_size=math.inf)])
```

---

```
content_type = models.CharField(max_length=64)

// /trustedsitters/backend/apps/children/validators.py

def __init__(self, *args, **kwargs):
    self.allowed_extensions = kwargs.pop('allowed_extensions', None)
    self.allowed_mimetypes = kwargs.pop('allowed_mimetypes', None)
    self.min_size = kwargs.pop('min_size', 0)
    self.max_size = kwargs.pop('max_size', None)
```

---

## 18.2 Black-Box

To try to test the upload of malicious file exploit, the group chose to upload a really common php webshell called c99 [5]. We managed to upload it under a children. See Figure 24.

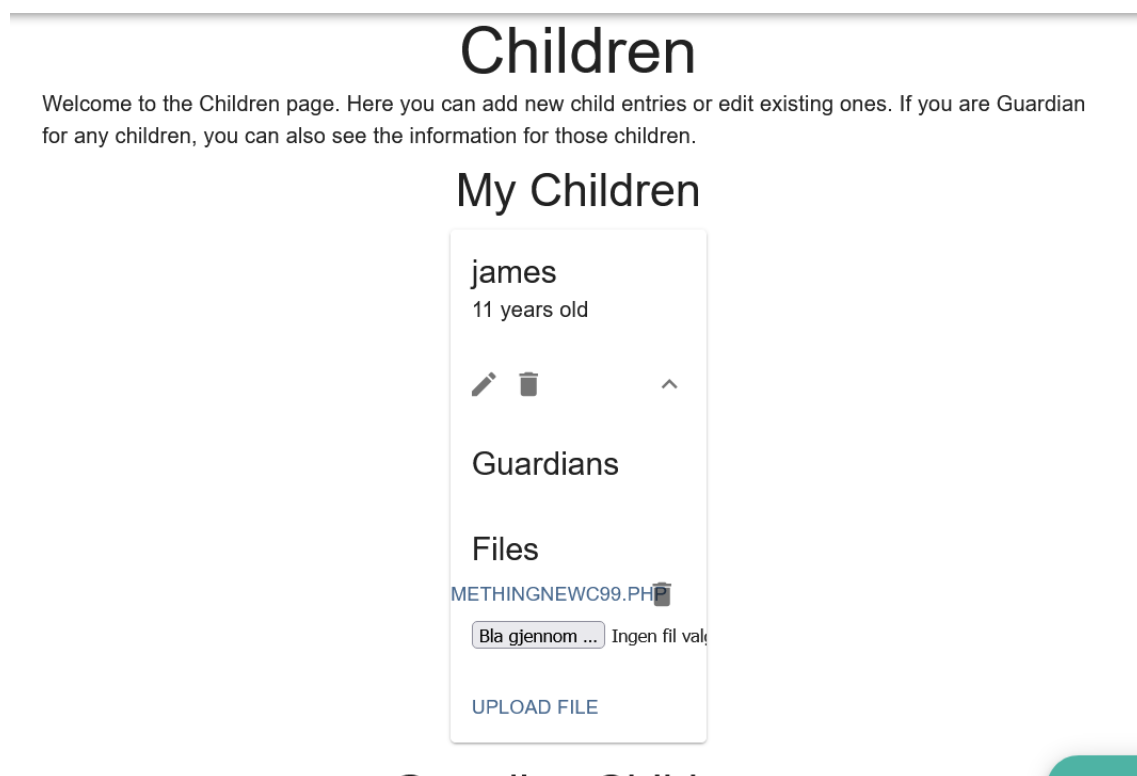


Figure 24: Uploaded c99 php webshell

We can see all the files that has been uploaded in the `/api/childfile/` request. We can see that our last file has our c99 php webshell in Figure 25.

ID	Source	Req. Timestamp	Method	URL	Code	Reason	RTT	Size Resp. Body	Highest Alert	Note	Tags
77	Proxy	14.03.2022, 13:44:56	GET	http://localhost:2121/api/child-file-download/24/	401	Unauthorized	9 ms	58 bytes		JSON	
80	Proxy	14.03.2022, 13:46:18	GET	https://contile.services.mozilla.com/v1/files	200	OK	268 ms	12 bytes	Low	JSON	
83	Proxy	14.03.2022, 13:48:32	GET	https://firefox.settings.services.mozilla.com/v1/buckets/m	200	OK	105 ms	223 bytes	Medium	JSON	
90	Proxy	14.03.2022, 13:48:32	GET	https://firefox.settings.services.mozilla.com/v1/buckets/m	200	OK	33 ms	683 bytes	Medium	JSON	
94	Proxy	14.03.2022, 13:48:32	GET	http://localhost:2121/	200	OK	6 ms	2 240 bytes	Medium	Script	
95	Proxy	14.03.2022, 13:48:32	GET	http://localhost:2121/static/css/2.7e708519.chunk.css	200	OK	5 ms	1 410 bytes		Comment	
96	Proxy	14.03.2022, 13:48:32	GET	http://localhost:2121/static/js/main.db74762f.chunk.js	200	OK	10 ms	49 548 bytes	Informational		
97	Proxy	14.03.2022, 13:48:32	GET	https://firefox.settings.services.mozilla.com/v1/buckets/m	200	OK	33 ms	232 bytes	Medium	JSON	
98	Proxy	14.03.2022, 13:48:32	GET	http://localhost:2121/static/js/2.5395af96.chunk.js	200	OK	27 ms	605 590 bytes	Low	Script	Comment
99	Proxy	14.03.2022, 13:48:32	GET	https://firefox.settings.services.mozilla.com/v1/buckets/m	200	OK	32 ms	2 105 bytes	Medium	JSON	
100	Proxy	14.03.2022, 13:48:32	GET	https://content-signature-2.cdn.mozilla.net/chains/remotes	200	OK	162 ms	5 348 bytes	Low		
106	Proxy	14.03.2022, 13:48:42	POST	http://localhost:2121/api/login	200	OK	12 ms	531 bytes			
107	Proxy	14.03.2022, 13:48:42	GET	http://localhost:2121/api/adverts/	200	OK	15 ms	1 026 bytes		JSON	
108	Proxy	14.03.2022, 13:48:45	GET	http://localhost:2121/api/child-file/	200	OK	10 ms	780 bytes		JSON	
109	Proxy	14.03.2022, 13:48:45	GET	http://localhost:2121/api/children/	200	OK	23 ms	525 bytes		JSON	
110	Proxy	14.03.2022, 13:49:10	GET	http://localhost:2121/api/child-file-download/25/	401	Unauthorized	6 ms	58 bytes		JSON	

Figure 25: Zap showing the list of all files uploaded with our c99 php webshell highlighted

To run the php webshell we have to make a get request to localhost:2121/api/child-file-download/25/ this is going to run the file on our server. If we make this get request as we can see in Figure 26, Trustedstitters is now compromised with c99 php webshell.

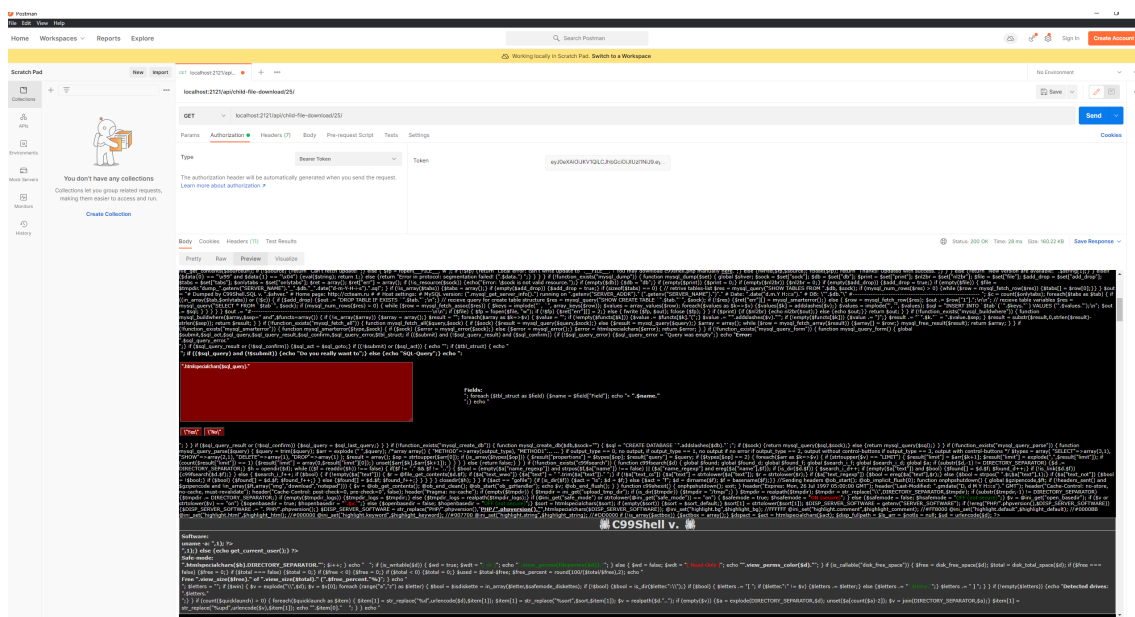


Figure 26: Postman showing the website compromised with the c99 php webshell

## 19 WSTG-ATHN-04

### Testing for Bypassing Authentication Schema:

Negligence or ignorance can, among others, lead to an authentication schema that can be bypassed just by skipping the log-in page and move on to an internal page meant for authenticated users.



---

## 19.1 White-Box

In `/offers/views.py` line 22 and `/offers/permissions.py` lines 4-8, there is no check for whether or not the user is authenticated. Hence, we can send a POST request to `/api/offers/` while not being authenticated. This will result in the creation of offers with the "sender" field being empty.

---

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

permission_classes = [IsSenderOrReceiver]

// /trustedsitters/backend/apps/offers/permissions.py

class IsSenderOrReceiver(permissions.BasePermission):

    def has_object_permission(self, request, view, obj):

        return obj.recipient == request.user.username or obj.sender ==
            request.user.username
```

---

## 19.2 Black-Box

We make a guardian offer from the `/offers` page while not being authenticated. See Figure 27. This action sends the POST request shown in Figure 28 and receives the 201 Created response in Figure 29.

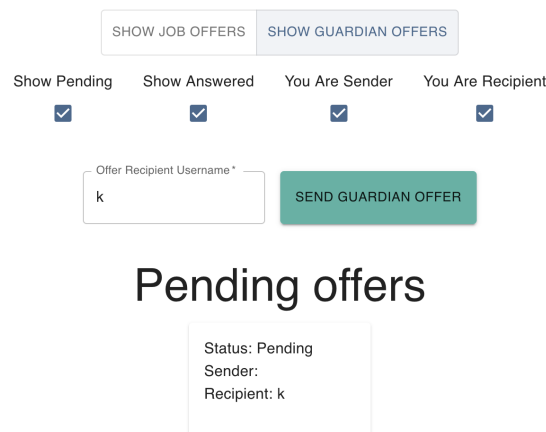


Figure 27: Creating a guardian offer while being logged out.

```
POST http://molde.idi.ntnu.no:21021/api/offers/ HTTP/1.1
Host: molde.idi.ntnu.no:21021
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:98.0) Gecko/20100101 Firefox/98.0
Accept: application/json, text/plain, */*
Accept-Language: en-GB,en;q=0.5
Content-Type: application/json
Content-Length: 46
Origin: https://molde.idi.ntnu.no:21021
Connection: keep-alive
Referer: https://molde.idi.ntnu.no:21021/offers
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin
{"recipient":"k","offerType":"GUARDIAN_OFFER"}
```

Figure 28: Request sent to the server without a sender specified.

---

```
HTTP/1.1 201 Created
Server: nginx/1.21.4
Date: Wed, 16 Mar 2022 08:59:33 GMT
Content-Type: application/json
Content-Length: 94
Connection: keep-alive
Allow: GET, HEAD, POST
X-Frame-Options: DENY
Vary: Origin
X-Content-Type-Options: nosniff
Referrer-Policy: same-origin
X-Content-Type-Options: nosniff

{"id":146,"offerType":"GUARDIAN_OFFER","sender":"","recipient":"k","status":"P","advert":null}
```

Figure 29: 201 Created response with no sender.

## 20 WSTG-ATHZ-02

### Testing for Bypassing Authorization Schema:

TrustedSitters's application designates users different roles to control what actions they can execute within the application. This is crucial in order to prevent users from performing actions they shouldn't be allowed to. Bypassing the authorization schema allows users to perform functions without having the necessary privileges to execute them.

### 20.1 White-Box

One of many permissions files where it lacks authentication checks. This means that another user can access most of endpoints without being logged in.

---

```
// /trustedsitters/backend/apps/adverts/permissions.py

from rest_framework import permissions

class IsOwnerOrReadOnly(permissions.BasePermission):
    """
    Custom permission to only allow owners of an object to edit it.
    """

    def has_object_permission(self, request, view, obj):
        # Read permissions are allowed to any request,
        # so we'll always allow GET, HEAD or OPTIONS requests.
        if request.method in permissions.SAFE_METHODS:
            return True

        # Write permissions are only allowed to the owner of the advert.
        return obj.owner == request.user
```

---

### 20.2 Black-Box

In Figure 30 we can see a non logged in user has access to the adverts page.

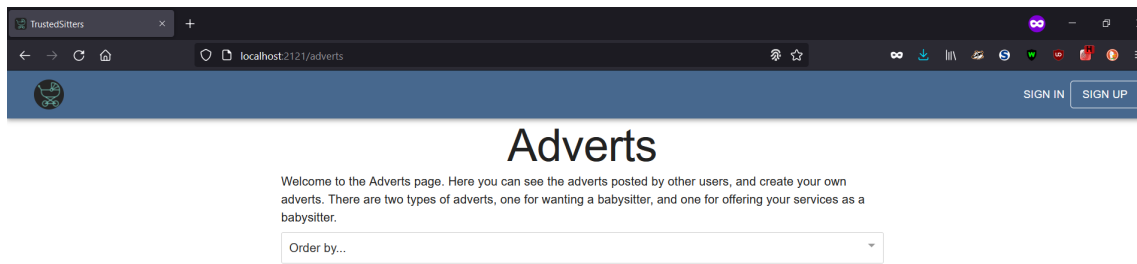


Figure 30: Trustedsitters advert page without being logged in

## 21 Conclusion

We have exposed several security vulnerabilities malicious actors could abuse. Among others, the application has weak username and password policy, no lockout mechanism, allows clickjacking, and discloses sensitive information. Trustedsitters need to fix these highlighted issues to provide their babysitting services securely.

---

## References

- [1] Daniel Miessler. *Password database*. 2020. URL: <https://github.com/danielmiessler/SecLists/blob/master/Passwords/Leaked-Databases/rockyou-75.txt> (visited on 03/15/2022).
- [2] Multiple. *Django library*. 2022. URL: <https://github.com/jazzband/django-axes> (visited on 03/15/2022).
- [3] *Md5 Decrypt Encrypt*. URL: <https://md5decrypt.net/en/> (visited on 03/15/2022).
- [4] *Base64 encode and decoder*. 2022. URL: <https://www.base64encode.org/> (visited on 03/15/2022).
- [5] Multiple. *C99 php webshell*. 2013. URL: <https://github.com/tennc/webshell/blob/master/php/PHPshell/c99/c99.php> (visited on 03/15/2022).