

**Politechnika Świętokrzyska w Kielcach Wydział Elektrotechniki,
Automatyki i Informatyki**

PROJEKT : Bezpieczeństwo aplikacji internetowych

Temat:

Aplikacja webowa do
zarządzania sklepem z
grami

Zespół:

David Salwa, Adam Rzepka,
Jakub Sadza, Bartłomiej Tokar

Grupa: 1ID24B

**Data oddania
sprawozdania:**

17.01.2024 r.

Spis treści

1. Wstęp

1.1 Wykorzystane technologie oraz narzędzia

1.2 Krótki opis uruchomienia aplikacji dla developera

2. Implementacja

2.1 Struktura projektu

2.2 Najważniejsze funkcjonalności

2.3 Zrzuty ekranu aplikacji

3. Testy

3.1 SonarCloud

3.2 Selenium ChromeDriver

4. Podział pracy

5. Podatności

5.1 Wybrane podatności

5.1.1 XSS

5.1.2 SQL INJECTION

5.1.3 Niezabezpieczone API

5.1.4 Brak walidacji danych

1. Wstęp

Tematem projektu było stworzenie aplikacji webowej do zarządzania sklepem z gramami. Aplikacja została zaimplementowana przy pomocy frameworka Django.

Implementacja Frontendu: podział aplikacji na pożądane odrębne widoki tj.:

- Strona główna,
- Produkty,
- Regulamin,
- Pomoc,
- Logowanie i Rejestracja.

Dołączona do projektu została baza danych SQLite, gdzie przechowywane są dane odnośnie produktów w sklepie (każdy produkt ma swoją podstronę do podglądu informacji), oraz zagnieżdżone zostały informacje o zarejestrowanych użytkownikach sklepu.

Projekt został odpowiednio zabezpieczony przy pomocy protokołu OFTP. Po zalogowaniu i rejestracji na konto email użytkownika wysyłany jest email z kodem. Użytkownik zostaje przekierowany na formularz do którego należy wpisać wysłany kod. Poprawny kod zapisany w sesji jest porównany z kodem wpisanym przez użytkownika. Po udanej autoryzacji użytkownik może dalej korzystać ze strony. Następnie zostały wykonane testy aplikacji przy pomocy Selenium. Przeprowadzono analizę statyczną kodu po której luki bezpieczeństwa zostały przeanalizowane i naprawione. Wykonano dokumentację API dostępną pod adresem /swagger/schema. Dodano REST API, zabezpieczone za pomocą autentykacji użytkownika oraz tokenów JWT. Projekt nieustannie wersjonowany przy użyciu Git.

1.1 Wykorzystane technologie oraz narzędzia

Spis najważniejszych bibliotek dla aplikacji: (plik requirements.txt)

Nazwa biblioteki	Wersja	Opis
asgiref	3.6.0	implementuje specyfikację ASGI (Asynchronous Server Gateway Interface) w Pythonie
Django	4.1.7	framework webowy napisany w języku Python. Umożliwia szybkie i efektywne tworzenie aplikacji internetowych
python-decouple	3.8	biblioteka umożliwiająca separację konfiguracji od kodu.

sqlparse	0.4.3	narzędzie do parsowania i formatowania zapytań SQL w języku Python
tzdata	2023.3	biblioteka zawierająca informacje o strefach czasowych (Time Zone Database)
pillow	9.5.0	używana do manipulacji obrazami w języku Python
django-crispy-forms	2.0	biblioteka ułatwiająca tworzenie formularzy w Django poprzez dostarczenie gotowych stylów CSS
django_otp	1.2.2	biblioteka wspierająca autentykację dwuetapową (OTP) w aplikacjach Django
selenium	4.10.0	narzędzie automatyzujące przeglądarki internetowe. Jest powszechnie używane do testowania aplikacji internetowych i skryptowania interakcji z przeglądarką
django-rest-framework	3.14.0	rozszerzenie frameworka Django, które ułatwia tworzenie interfejsów API (RESTful) dla aplikacji Django
django-rest-framework-simplejwt	5.2.2	biblioteka dostarczająca prosty sposób na implementację uwierzytelniania opartego na tokenach JWT
drf-yasg		narzędzie, które generuje dokumentację interaktywną dla API Django REST Framework
django-rest-swagger		narzędzie do generowania dokumentacji interaktywnej dla API Django REST Framework

Obsługa projektu

Narzędzia:

1. IDE:
 - a. Visual Studio Code (v. 1.84)
 - b. PyCharm (v. 2023.3.2)
2. Zarządzanie kontrolą wersji - GitHub

Środowisko:

1. System operacyjny - Windows 10 lub nowszy
2. Języki programowania:
 - a. Python (v. 3.10)

1.2 Krótki opis uruchomienia aplikacji dla developera

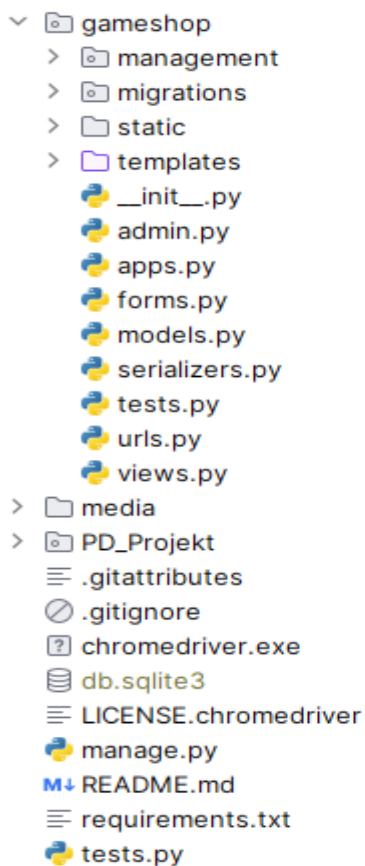
- Po pobraniu projektu, należy zainstalować wszystkie wymagane biblioteki za pomocą `pip install -r requirements.txt`
- Następnie, w głównym folderze z projektem należy wykonać komendę `python manage.py makemigrations`, następnie `python manage.py migrate`
- Projekt powinien być gotowy do uruchomienia. Do uruchomienia należy wykorzystać komendę `python manage.py runserver`, strona www dostępna będzie pod adresem `127.0.0.1:8000`

Panel administratora dostępny jest pod linkiem `/admin`, dostępny jedynie z maszyny lokalnej.

2. Implementacja

2.1 Struktura projektu

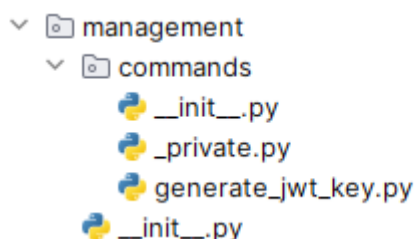
Struktura katalogów oraz plików aplikacji odpowiedzialna za jej funkcjonowanie oraz wygląd jest umieszczona w katalogu głównym o nazwie „gameshop”. (Kody źródłowe plików z poszczególnych katalogów znajdują się w załączniku / repozytorium GitHub)



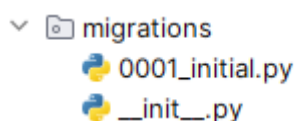
Projekt został podzielony na kilka ważnych sekcji jeśli chodzi o pliki. W głównym directory znajdują się: manage.py - plik wykonywalny, który służy do rozruchu projektu, requirements w których mieszczą się wszystkie biblioteki wykorzystywane oraz plik zawierający testy napisane przy pomocy Selenium dla frontu aplikacji.

Opis poszczególnych podkatalogów:

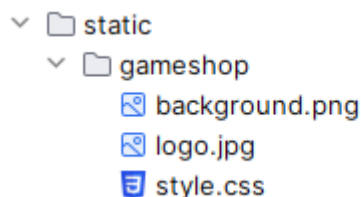
Katalog **management** - zawiera pliki odpowiedzialne do generowania par kluczy JWT dla użytkownika, gdy podana jest nazwa użytkownika jako argument polecenia. Kod w pliku generate_jwt_key przyjmuje nazwę użytkownika jako argument, znajduje tego użytkownika w bazie danych, a następnie generuje i wypisuje tokeny odświeżania i dostępu dla tego użytkownika.



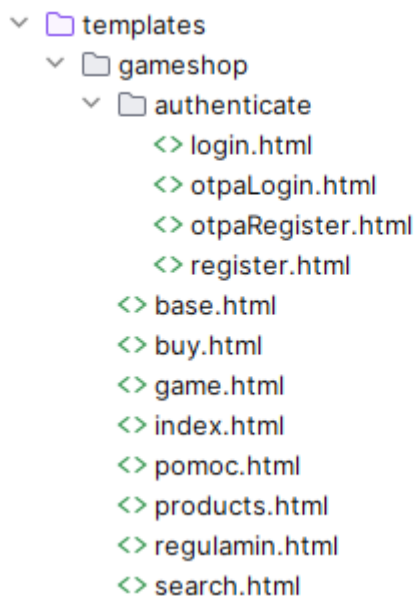
Katalog **migrations** jest kluczowym elementem, który ułatwia deweloperom zarządzanie ewolucją struktury bazy danych w trakcie rozwoju projektu Django. Automatyzuje proces tworzenia, zastosowywania i cofania zmian w bazie danych, co znacznie ułatwia utrzymanie spójności między modelem a rzeczywistą bazą danych. Tworzy chronologiczną historię zmian w strukturze bazy danych. Każda migracja ma przypisany numer w celu utrzymania porządku.



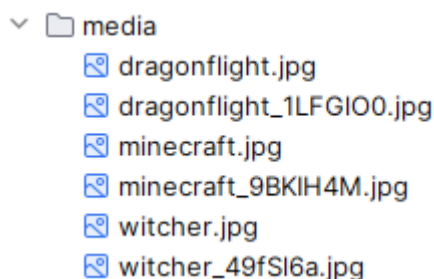
Katalog **static** odpowiedzialny jest za przetrzymywanie statycznych plików do działania frontendu aplikacji takich jak tło, logo czy arkusz styli.



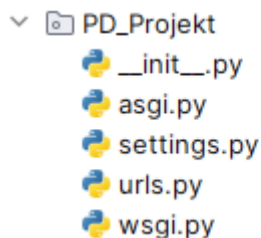
Katalog **templates** zawiera pliki odpowiedzialne za struktury widoków aplikacji. Są to kody języka HTML, które odpowiadają za wygląd poszczególnych widoków. Aplikacja posiada widok bazowy base.html, który jest dziedziczony przez wszystkie inne widoki.



Katalog **media** - przechowywane są w nim zdjęcia służące jako etykiety dla gier.



Katalog PD_projekt - pliki asgi.py, settings.py, urls.py, i wsgi.py pełnią kluczowe role w konfiguracji i obszarze działania aplikacji.



asgi.py jest plikiem konfiguracyjnym służącym do obsługi aplikacji Django w środowisku asynchronicznym.

- ASGI to interfejs umożliwiający obsługę żądań asynchronicznie, co jest szczególnie przydatne w przypadku obsługi wielu jednoczesnych połączeń w czasie rzeczywistym, takich jak aplikacje obsługujące WebSocket czy Server-Sent Events (SSE).

settings.py zawiera konfigurację projektu Django.

- W tym pliku znajdują się ustawienia dotyczące bazy danych, plików statycznych, middleware, aplikacji, języka, strefy czasowej, bezpieczeństwa i wielu innych aspektów konfiguracyjnych projektu.

urls.py definiuje zestaw wzorców adresów URL dla aplikacji Django.

- Każdy wzorzec URL określa, jakie widoki (views) powinny obsłużyć konkretne żądanie HTTP.
- Wzorce te są używane do mapowania adresów URL na konkretne widoki, co umożliwia Django odpowiednie przekierowanie żądań do odpowiednich części kodu.

wsgi.py jest plikiem konfiguracyjnym, który definiuje interfejs WSGI dla aplikacji Django.

- WSGI to standardowy interfejs komunikacyjny między aplikacją webową napisaną w języku Python, a serwerem WWW (np. Apache, Nginx).

2.2. Najważniejsze funkcjonalności

Generowanie kluczy JWT

Funkcja generuje parę kluczy JWT (JSON Web Token) dla określonego użytkownika.

- Pobiera użytkownika (user) z bazy danych Django na podstawie podanego argumentu "user".
- Następnie tworzy nowy token odświeżenia (RefreshToken) dla tego użytkownika.
- Wypisuje nazwę użytkownika na konsolę.
- Zwraca słownik zawierający klucze "refresh" i "access", z których każdy jest przekształcony na ciąg znaków (str(refresh) i str(refresh.access_token)).


```

class Command(BaseCommand):
    help = "Generates pair of JWT keys for user manually"

    def add_arguments(self, parser):
        parser.add_argument("user", nargs="+", type=str)

    def handle(self, *args, **options):
        user =
User.objects.filter(username=options["user"]).values_list("username", flat=True)
        refresh = RefreshToken.for_user(user)
        print(user)

        return {
            'refresh': str(refresh),
            'access': str(refresh.access_token),
        }

```

Weryfikacja kodu OTP podczas rejestracji:

Funkcja ta obsługuje proces weryfikacji kodu OTP podczas rejestracji użytkownika, przy czym zakłada, że kod OTP jest przechowywany w sesji. Jeśli weryfikacja przebiegnie pomyślnie, użytkownik zostaje oznaczony jako zweryfikowany, a następnie przekierowany do strony logowania.

```

def otpaRegister(request):
    if request.method == 'POST':
        form = OTPVerificationForm(request.POST)
        if form.is_valid():
            otp_code = form.cleaned_data['otp_code']
            stored_otp_code = request.session.get('otp_code')
            if otp_code == stored_otp_code:
                # Kod OTP jest poprawny
                del request.session['otp_code'] # Usuń kod OTP z
sesji po weryfikacji

                user_id = request.session.get('new_user_id')
                if user_id:
                    user = User.objects.get(id=user_id)
                    user.email_verified = True
                    user.save()

                    messages.success(request, 'Konto zostało
założone pomyślnie. Możesz się teraz zalogować.')
                    return redirect('gameshop:login')
                else:
                    messages.error(request, 'Wystąpił błąd podczas

```

```
rejestracji. Spróbuj ponownie.')
```

```
    else:
```

```
        # Kod OTP jest niepoprawny
```

```
        messages.error(request, 'Podany kod OTP jest
```

```
niepoprawny.')
```

```
    else:
```

```
        form = OTPVerificationForm()
```

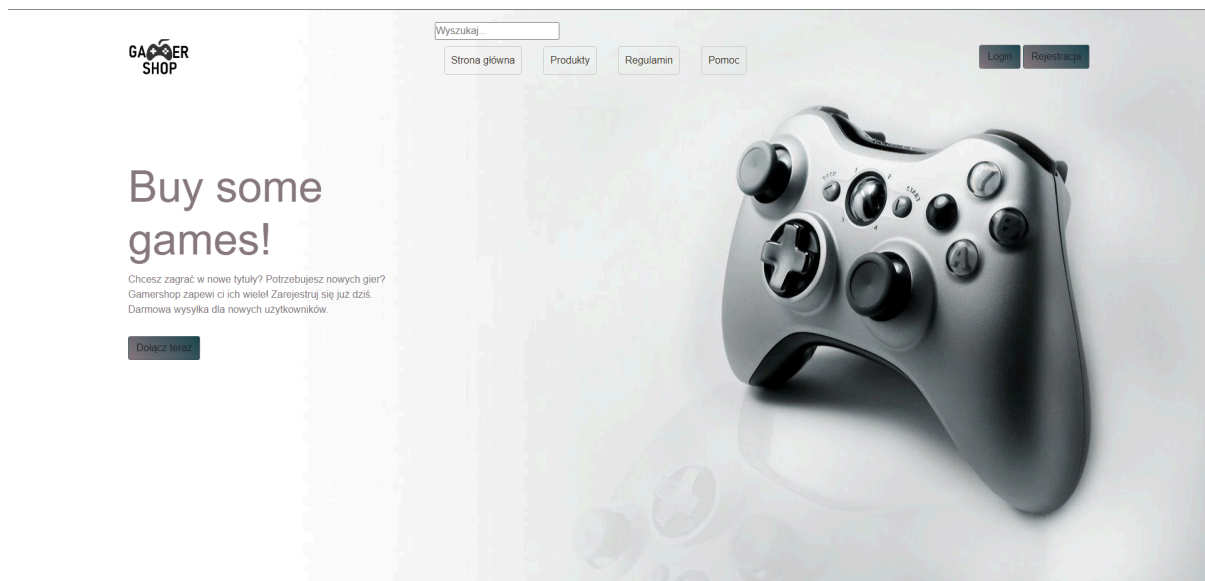
```
    return render(request,
```

```
'gameshop/authenticate/otpRegister.html', {'form': form})
```

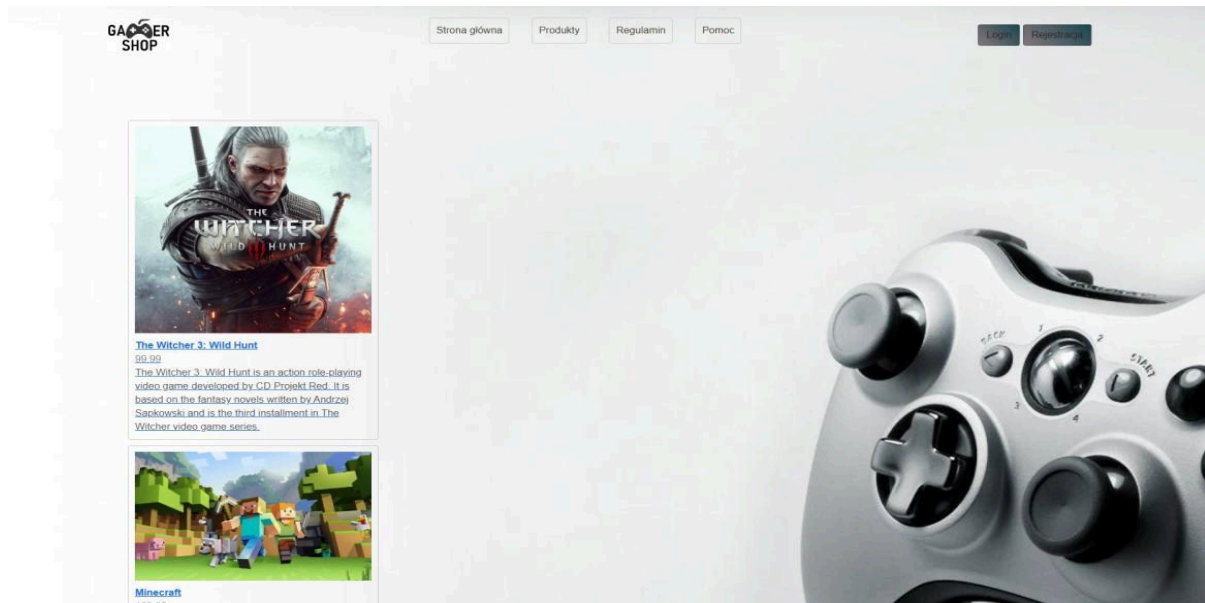
2.3. Zrzuty ekranu aplikacji

Działanie aplikacji, najważniejsze komponenty:

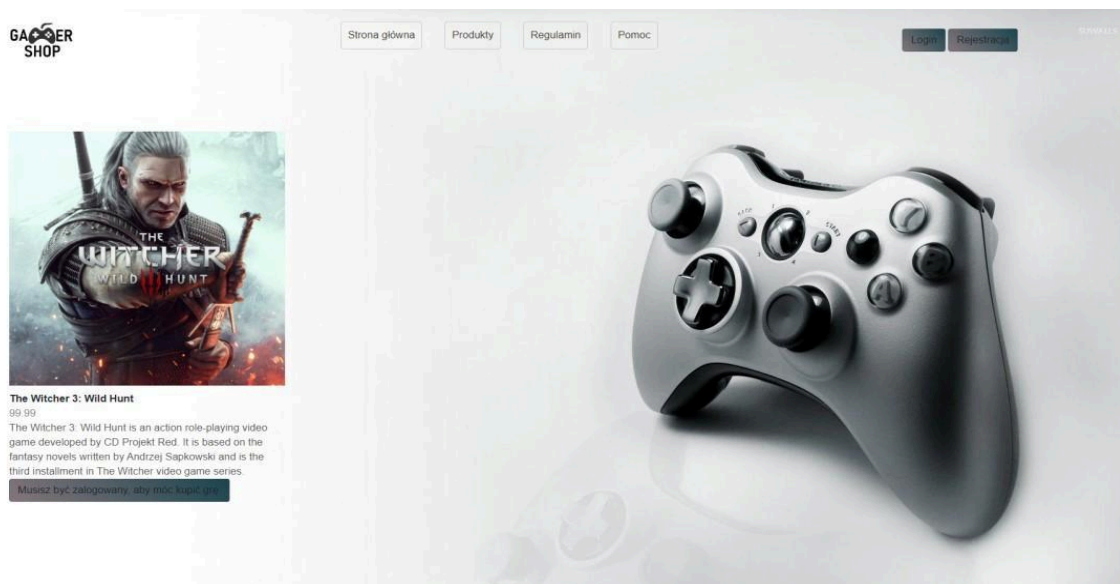
- Strona startowa



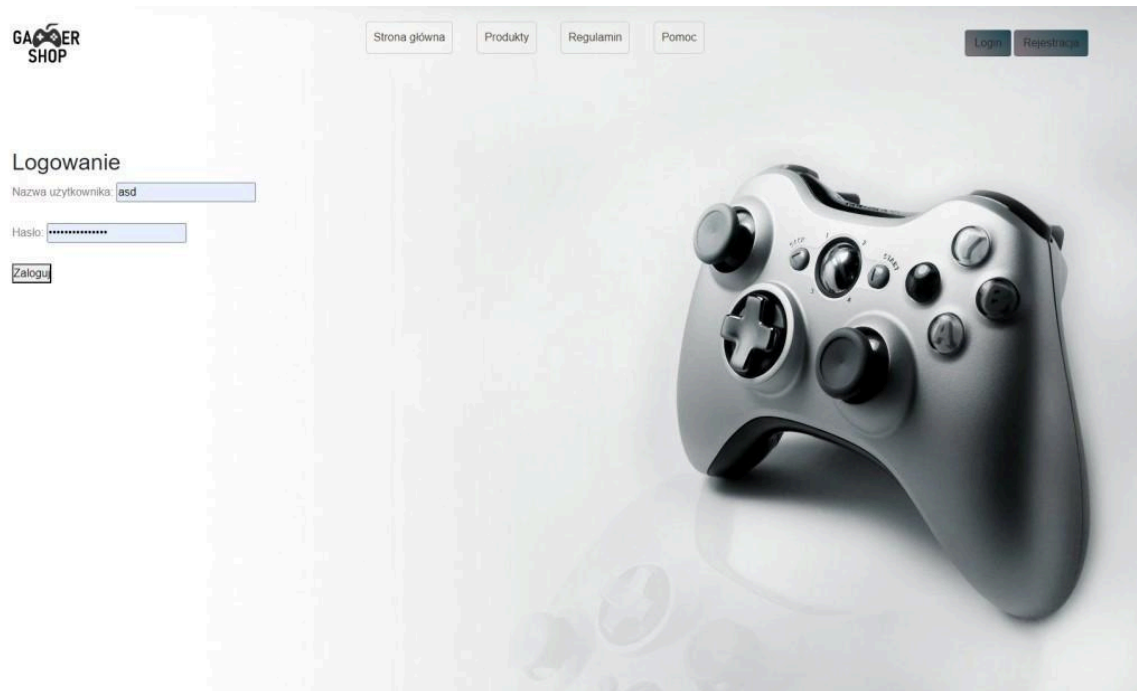
- Lista produktów, dynamicznie pobierana z bazy danych



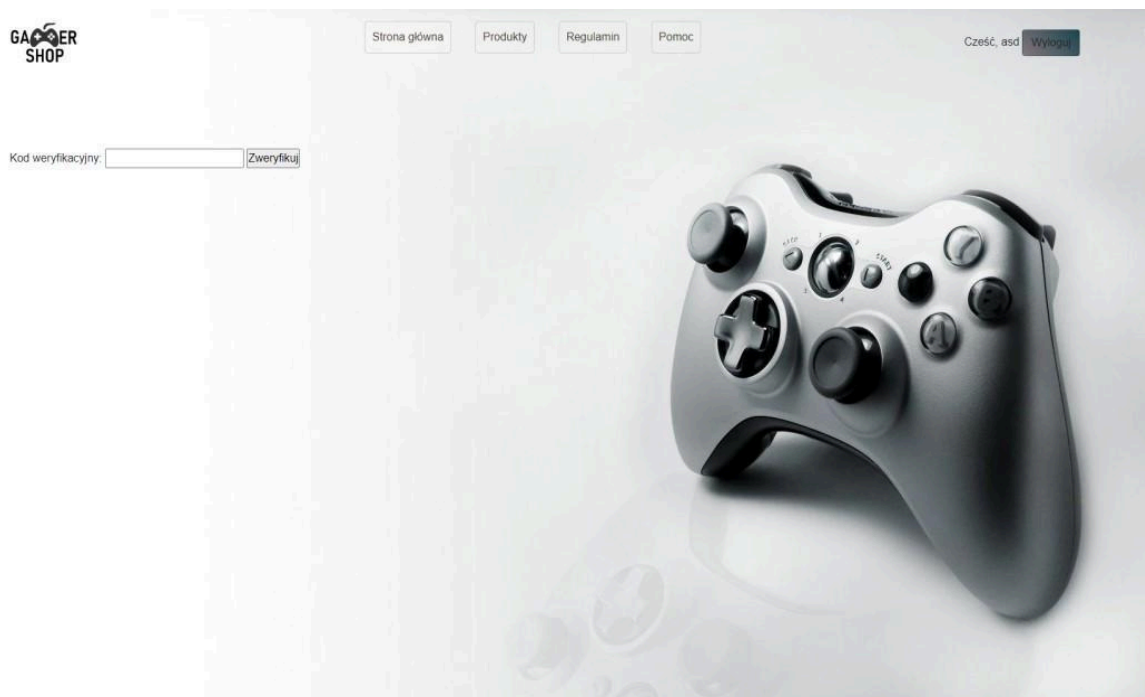
- Osobna strona dla każdego produktu dostępnego na stronie. Możliwość zakupu gry dopiero po zalogowaniu.



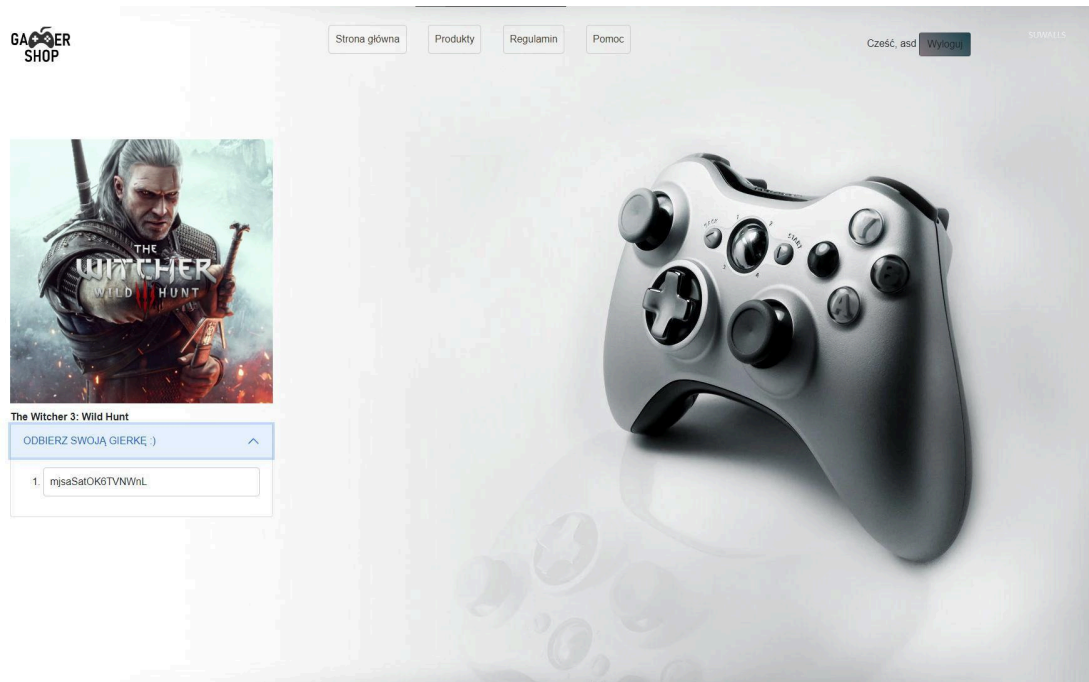
- Strona logowania do aplikacji



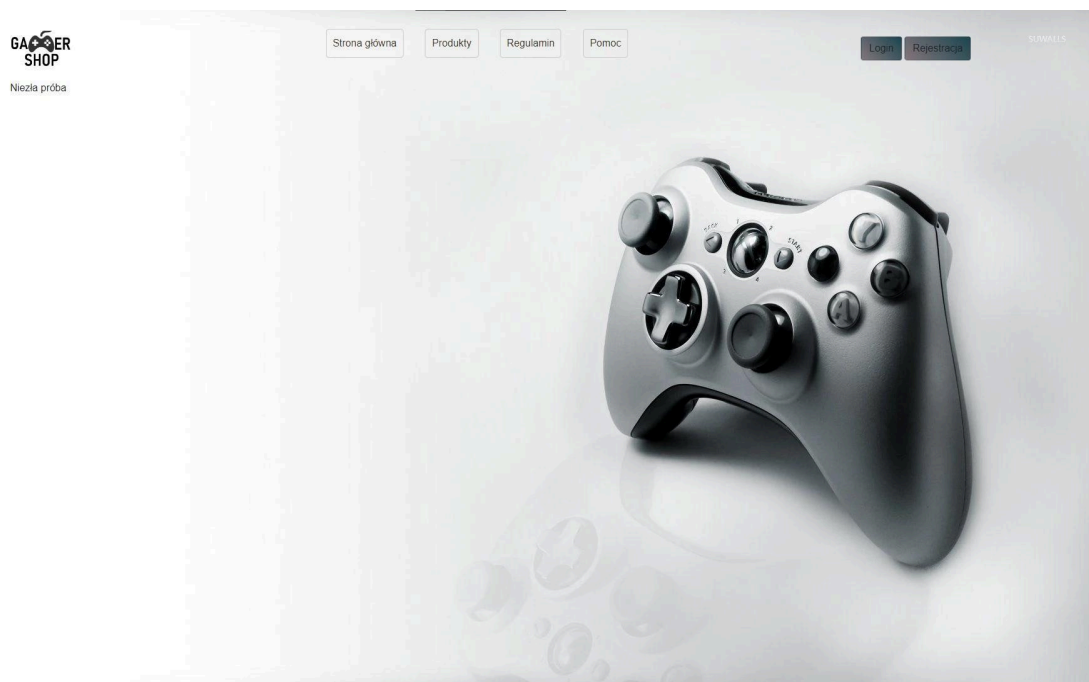
- Strona uwierzytelniania dwuetapowego za pomocą OTP



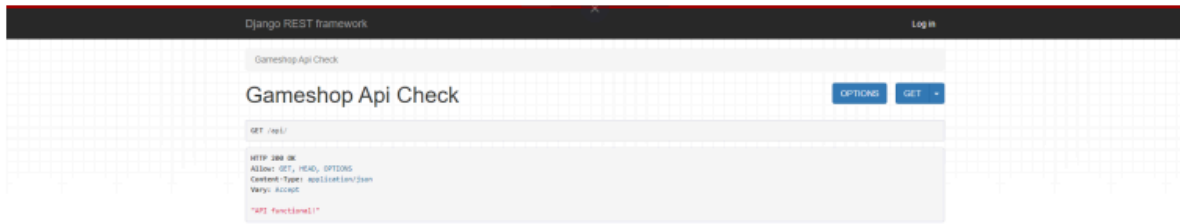
- Strona po uwierzytelnieniu generuje kody do gier



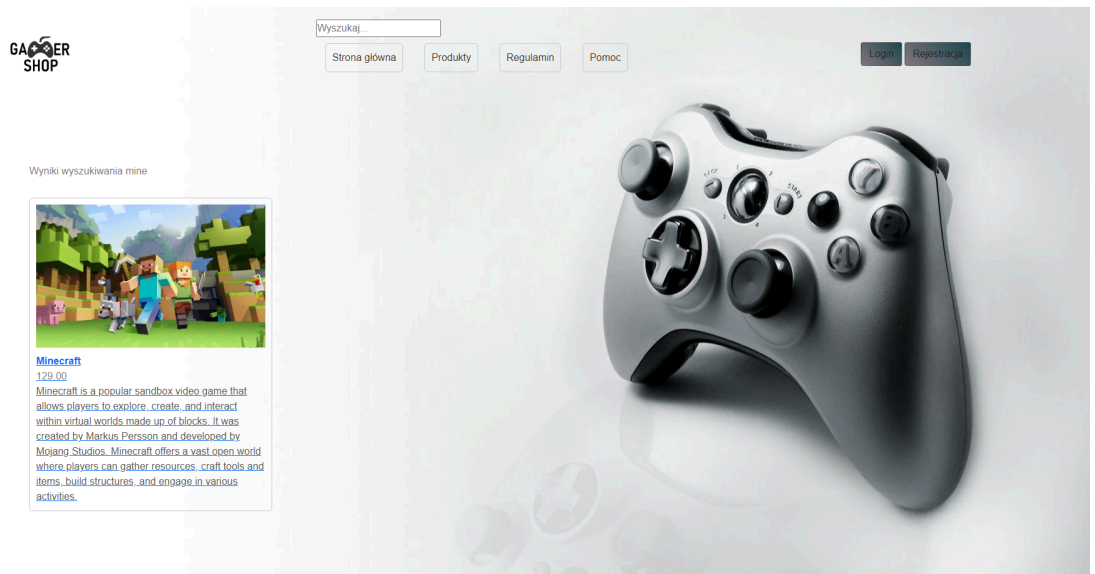
- Strona również sprawdza czy użytkownik jest zalogowany w trakcie “kupowania gry”, więc jeśli będzie próbował obejść nasze ✨świetne✨ zabezpieczenia spotka się z informacją



- strona zawierająca dostęp do api



- Na stronie dostępna jest również wyszukiwarka, dzięki której możemy szybciej znaleźć tytuł, który nas interesuje:



3. Testy

3.1 SonarCloud

Wynik ewaluacji kodu z wykorzystaniem narzędzia SonarCloud. Uwagi dotyczące audytu bezpieczeństwa: SonarCloud dokonuje analizy kodu, korzystając z dokumentacji Django 3.1. Do projektu użyto Django w wersji 4.2. Niektóre funkcje wykorzystywane w projekcie są bezpieczne używając wersji Django 4.2, natomiast w wersji Django 3.1 były przyczyną niektórych podatności.

adamulll > PD_Projekt > main

Summary Issues Security Hotspots Measures Code Activity

The last analysis has a warning.

Quality Gate **Passed** New Code Overall Code

Reliability

0 Bugs

Maintainability

4 Code Smells

Security

0 Vulnerabilities

Security Review

6 Security Hotspots 0.0% Reviewed

Coverage

A few extra steps are needed for SonarCloud to analyze your code coverage

[Setup coverage analysis](#)

Duplications

0.0% Duplications

Raport z testów aplikacji, wygenerowany przy pomocy narzędzia coverage.py:

Coverage report: 73%				
coverage.py v7.2.7, created at 2023-06-27 22:38 +0200				
Module	statements	missing	excluded	coverage
PD_Projekt__init__.py	0	0	0	100%
PD_Projekt\settings.py	30	0	0	100%
PD_Projekt"urls.py	11	1	0	91%
gameshop__init__.py	0	0	0	100%
gameshop\admin.py	5	0	0	100%
gameshop\apps.py	4	0	0	100%
gameshop\forms.py	13	0	0	100%
gameshop\management__init__.py	0	0	0	100%
gameshop\management\commands__init__.py	0	0	0	100%
gameshop\migrations\0001_initial.py	6	0	0	100%
gameshop\migrations__init__.py	0	0	0	100%
gameshop\models.py	33	4	0	88%
gameshop\serializers.py	14	0	0	100%
gameshop\tests.py	1	0	0	100%
gameshop"urls.py	5	0	0	100%
gameshop\views.py	160	93	52	42%
manage.py	12	2	0	83%
tests.py	79	0	0	100%
Total	373	100	52	73%
coverage.py v7.2.7, created at 2023-06-27 22:38 +0200				

3.1 Selenium ChromeDriver

ChromeDriver to samodzielny serwer lub oddzielny plik wykonywalny używany przez Selenium WebDriver do kontrolowania przeglądarki Chrome.

Aby uruchomić Live Testy Selenium w Chrome należy:

Dodać w systemowych PATH ścieżkę do chromedriver.exe:

można to zrobić za pomocą CMD: setx PATH "%PATH%;waszaSciezkaDoPlikuExe"
np. setx PATH "%PATH%;C:\Users\Kowalski\Desktop\PD_Projekt\chromedriver.exe",

a następnie za pomocą komendy w terminalu: "python manage.py test" - rozpoczyna się proces testowania (Należy mieć odpalony projekt przed wystartowaniem testów).

Przykład kodu z testem: (plik tests.py)

```
class LoginFormTest(LiveServerTestCase):

    def testform(self):
        """
        Testuje formularz logowania.
        Otwiera przeglądarkę Chrome, przechodzi do strony
logowania,
        oczekuje na załadowanie się strony, wprowadza nazwę
użytkownika i hasło,
        a następnie wysyła formularz.
        """
        driver = webdriver.Chrome()

        driver.get('http://127.0.0.1:8000/login/')
        time.sleep(3)
        user_name = driver.find_element(By.NAME, 'username')
        user_password = driver.find_element(By.NAME,
'password')
        submit = driver.find_element(By.ID, 'submit')

        user_name.send_keys('admin1')
        user_password.send_keys('admin')
        time.sleep(3)
        submit.send_keys(Keys.RETURN)
```



```
System check identified no issues (0 silenced).  
  
DevTools listening on ws://127.0.0.1:60276/devtools/browser/97b5b756-1b7c-4f73-ba5f-1f4d002a3c8d  
.  
-----  
Ran 1 test in 3.596s  
  
OK  
Destroying test database for alias 'default'...  
PS C:\Users\Jakub Sadza\Desktop\PD_Projekt> |
```

4. Podział pracy

Zaangażowanie w zespole oraz role:

- **David Salwa:** konfiguracja bazy, utworzenie formularzu logowania oraz rejestracji, konfiguracja stron do logowania oraz rejestracji, stworzenie poprawa funkcji wysyłania kodu OTP na maila podczas logowania oraz implementacja weryfikacji OTP podczas rejestracji.
- **Adam Rzepka:** Zaprojektowanie UI aplikacji, implementacja frontendu aplikacji, stworzenie widoków strony głównej oraz produktów. Uzupełnienie bazy danymi. Dynamiczne wyświetlanie zawartości bazy na stronie. Stworzenie dynamicznych linków dla każdego z produktów. Zaprojektowanie oraz dodanie protokołu OFTP. Wysłanie maila z kodem do użytkownika. Przeprowadzenie analizy statycznej. Kod coverage
- **Jakub Sadza:** Rozbudowa frontendu aplikacji, stworzenie nowych widoków regulaminu, pomocy oraz ich konfiguracja. Dodanie widoku stron logowania i rejestracji. Testy aplikacji. Przeprowadzenie testów jednostkowych aplikacji dla FrontEnd przy pomocy Selenium, stworzenie dokumentacji API dla projektu przy pomocy Django REST Swagger. Tworzenie dokumentacji kodu dla plików projektowych.
- **Bartłomiej Tokar:** przygotowanie backendowej części aplikacji - utworzenie i konfiguracja nowej aplikacji django, utworzenie oraz konfiguracja bazy danych, konfiguracja adresów url strony, projekt i implementacja REST API strony, usprawnienia do plików html, opcja kupowania gier oraz generowanie kluczy gier

5. Podatności

5.1 Wybrane podatności

5.1.1 XSS

Podatność **XSS** (Cross-Site Scripting) to typ ataku na aplikacje internetowe, który pozwala atakującemu na wstrzyknięcie złośliwego kodu do strony internetowej wyświetlanej przez inne osoby. Atak XSS występuje, gdy atakujący wykorzystuje aplikację internetową do wysłania złośliwego kodu, zazwyczaj w formie skryptu po stronie przeglądarki, do innego użytkownika końcowego.

Podatność ta jest powszechna tam, gdzie aplikacja internetowa używa danych wejściowych od użytkownika w wygenerowanym przez siebie wyjściu bez ich walidacji lub kodowania. Złośliwy skrypt może uzyskać dostęp do wszelkich ciasteczek, tokenów sesji lub innych wrażliwych informacji przechowywanych przez przeglądarkę i używanych na danej stronie. Te skrypty mogą nawet przepisać zawartość strony HTML.

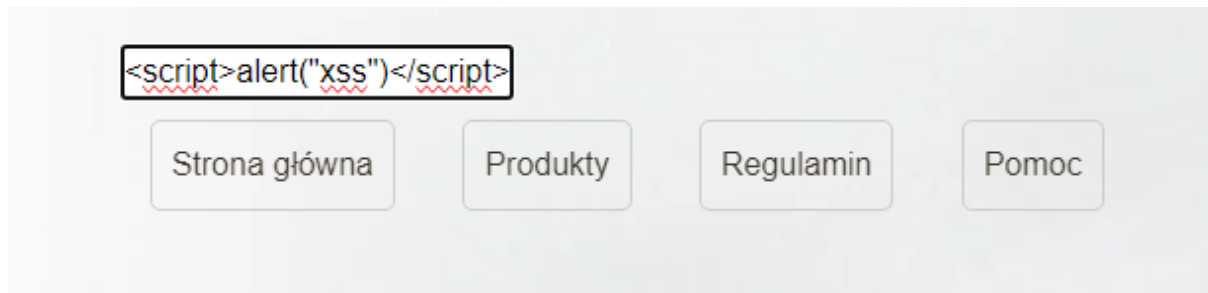
Ataki XSS można generalnie podzielić na dwie kategorie: odbite i przechowywane. Istnieje również trzeci, znacznie mniej znany typ ataku XSS, nazywany XSS opartym na DOM.

1. Reflected XSS: Ataki odbite to te, w których wstrzyknięty skrypt jest odbijany od serwera internetowego, na przykład w komunikacie o błędzie, wyniku wyszukiwania lub innym odpowiedzi, która zawiera niektóre lub wszystkie dane wejściowe wysłane do serwera jako część żądania.
2. Stored XSS: Ataki przechowywane są bardziej niebezpieczne. W tym przypadku złośliwy skrypt jest trwale przechowywany na celach ataku, na przykład w bazie danych, na forum dyskusyjnym, w komentarzu do wiadomości, polu informacji o użytkowniku itp.
3. DOM-Based XSS: Jest to typ ataku XSS, który występuje, gdy złośliwy skrypt jest w stanie wpłynąć na strukturę HTML lub wykonanie skryptu na stronie za pomocą manipulacji DOM (Document Object Model) strony.

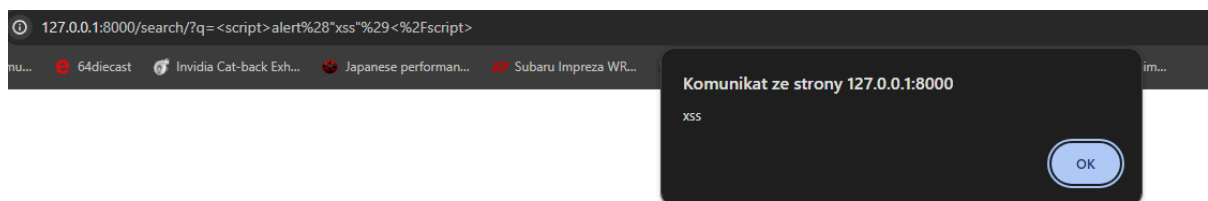
Podatność XSS w naszej aplikacji:

Miejscem występowania podatności XSS w naszej aplikacji występuje w wyszukiwarce. Dane wejściowe nie są filtrowane w żaden sposób, a następnie dane, które użytkownik wprowadził, wyświetlane są w banerze "wynik wyszukiwania *dane_użytkownika*" na stronie, co umożliwia uruchomienie złośliwego kodu bezpośrednio na stronie.

Można to sprawdzić wprowadzając prosty blok skryptu Javascript, np. `alert('wiadomość')`:



Próba wyszukania daje następujący efekt:



Powód podatności:

Wszystkie dane wejściowe w Django są domyślnie uciekane, o ile pole wyświetlania danych nie jest oznaczone flagą `safe` - jak w tym przypadku:

```
<p> Wyniki wyszukiwania {{ search | safe }} </p>
```

Sposób zabezpieczenia:

W frameworku Django należy unikać flagi `safe`, w szczególności w przypadku danych wejściowych podawanych przez użytkowników strony. Wszystkie pola tekstowe w każdym innym przypadku są uciekane domyślnie.

5.1.2 SQL INJECTION

Atak **SQL Injection** (SQLi): to podatność bezpieczeństwa aplikacji internetowych, która pozwala atakującemu na ingerowanie w zapytania, które aplikacja wysyła do swojej bazy danych. Atakujący może dzięki temu przeglądać dane, do których normalnie nie ma dostępu. Może to obejmować dane należące do innych użytkowników lub dowolne inne dane, do których aplikacja ma dostęp. W wielu przypadkach atakujący może modyfikować lub usuwać te dane, powodując trwałe zmiany w treści lub zachowaniu aplikacji.

Podatność ta występuje najczęściej tam, gdzie aplikacja internetowa używa danych wejściowych od użytkownika w wygenerowanym przez siebie wyjściu bez ich walidacji lub kodowania. Atak SQL Injection może prowadzić do kradzieży wrażliwych informacji, instalacji złośliwego oprogramowania lub przejęcia kont użytkowników.

Podatność SQL Injection w naszej aplikacji:

Podatność SQL Injection występuje w tym samym punkcie aplikacji, co podatność XSS - w pasku wyszukiwania. Dane nie są walidowane w żaden sposób, a zapytanie do bazy SQL jest źle sformatowane, co umożliwia na przeprowadzenie ataku SQL Injection:

```
@require_GET
def search(request):
    query = request.GET.get("q")
    sql = "SELECT * FROM gameshop_game WHERE title LIKE '%%%s%%'" %
    query
    products = Game.objects.raw(sql)
    return render(request, "gameshop/search.html", {'products':
    products, 'search': query})
```

Przez użycie funkcji *raw* na bazie wykonywane jest “surowe” zapytanie SQL, pomijając domyślne uciekanie. Pozwala to na wykonanie dowolnego zapytania SQL wprowadzonego w pasek wyszukiwarki przez użytkownika.

Sposób zabezpieczenia podatności

Należy używać wbudowanych w Django funkcji *Query*, która automatycznie ucieka nawet niezaweryfikowane wcześniej dane oraz formuje zapytania SQL automatycznie. Zabezpieczony kod wyglądałby następująco:

```
@require_GET
def search(request):
    query = request.GET.get("q")
    products = Game.objects.filter(title__icontains=query)
    return render(request, "gameshop/search.html", {'products':
products, 'search': query})
```

Funkcja `.filter(title__icontains=query)` tworzy dokładnie takie samo zapytanie do bazy, jak to, które jest zadeklarowane w wersji podatnej na ataki SQL.

5.1.3 Niezabezpieczone API

Podatność “Niezabezpieczone API” w aplikacjach internetowych odnosi się do sytuacji, gdy interfejsy programowania aplikacji (API), które są kluczowym elementem współczesnych aplikacji internetowych, nie są odpowiednio zabezpieczone. Niezabezpieczone API mogą prowadzić do różnych rodzajów ataków, w tym do naruszenia danych, ataków DDoS, utraty danych i ostatecznie do strat ekonomicznych.

Oto kilka typowych podatności związanych z niezabezpieczonymi API¹:

- Common Vulnerabilities and Exposures (CVEs): To są znane podatności, które zostały zidentyfikowane i udokumentowane w publicznej bazie danych.
- Ataki typu Denial of Service (DDoS): Atakujący może wykorzystać API do przeciążenia serwera, co prowadzi do niedostępności usługi.
- Ataki przez wstrzyknięcie danych: Atakujący może wstrzyknąć złośliwe dane do API, co może prowadzić do nieautoryzowanego dostępu do danych lub funkcji.

Aby zabezpieczyć się przed tymi zagrożeniami, deweloperzy powinni wdrożyć odpowiednie zabezpieczenia w warstwie API, takie jak autentykacja oparta na

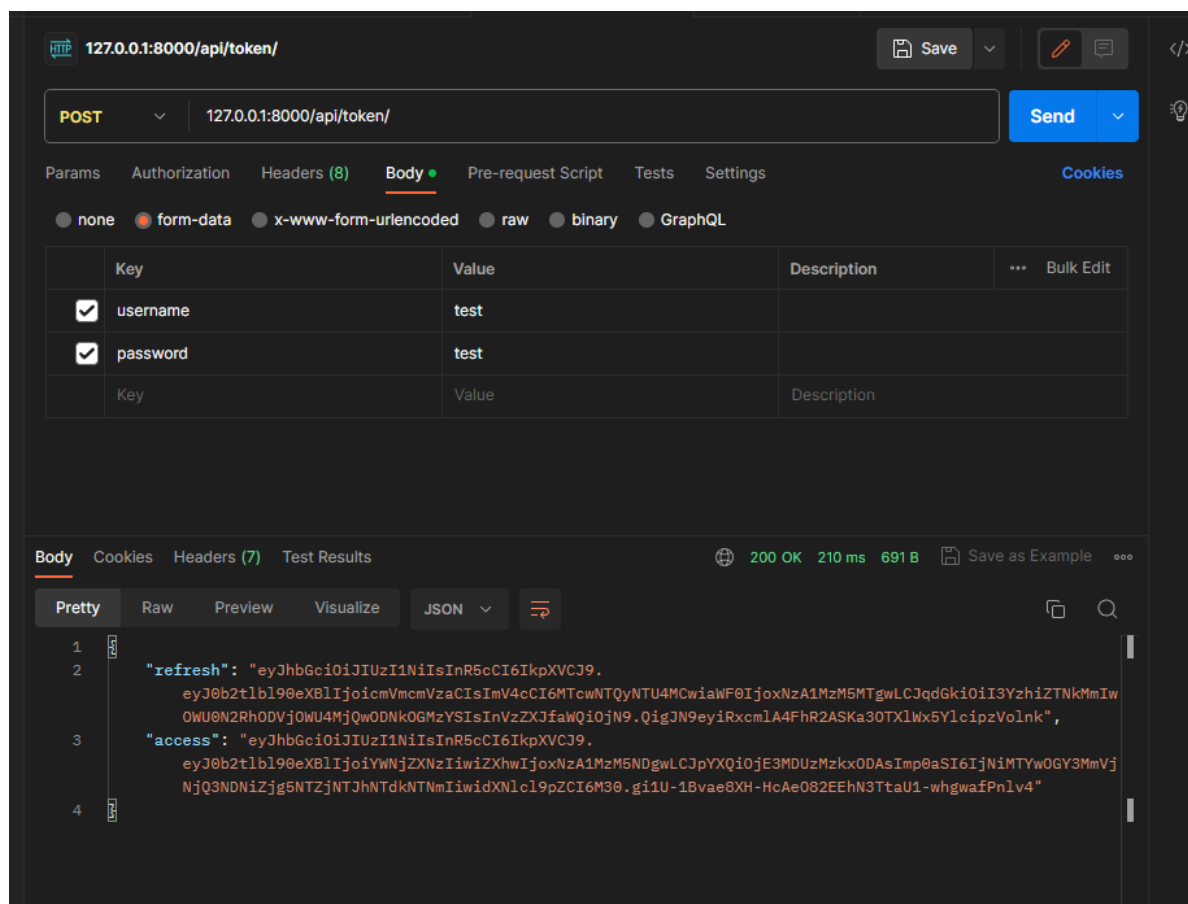
tokenach, szyfrowanie komunikacji, kontrole dostępu, zabezpieczenia przed atakami CSRF, monitorowanie i logowanie zdarzeń, oraz ochrona przed atakami DDoS. Regularne audyty bezpieczeństwa oraz ścisła polityka zarządzania dostępem do API również są kluczowe dla minimalizowania ryzyka związanego z niezabezpieczonymi interfejsami programowania aplikacji.

Niezabezpieczone API w naszej aplikacji:

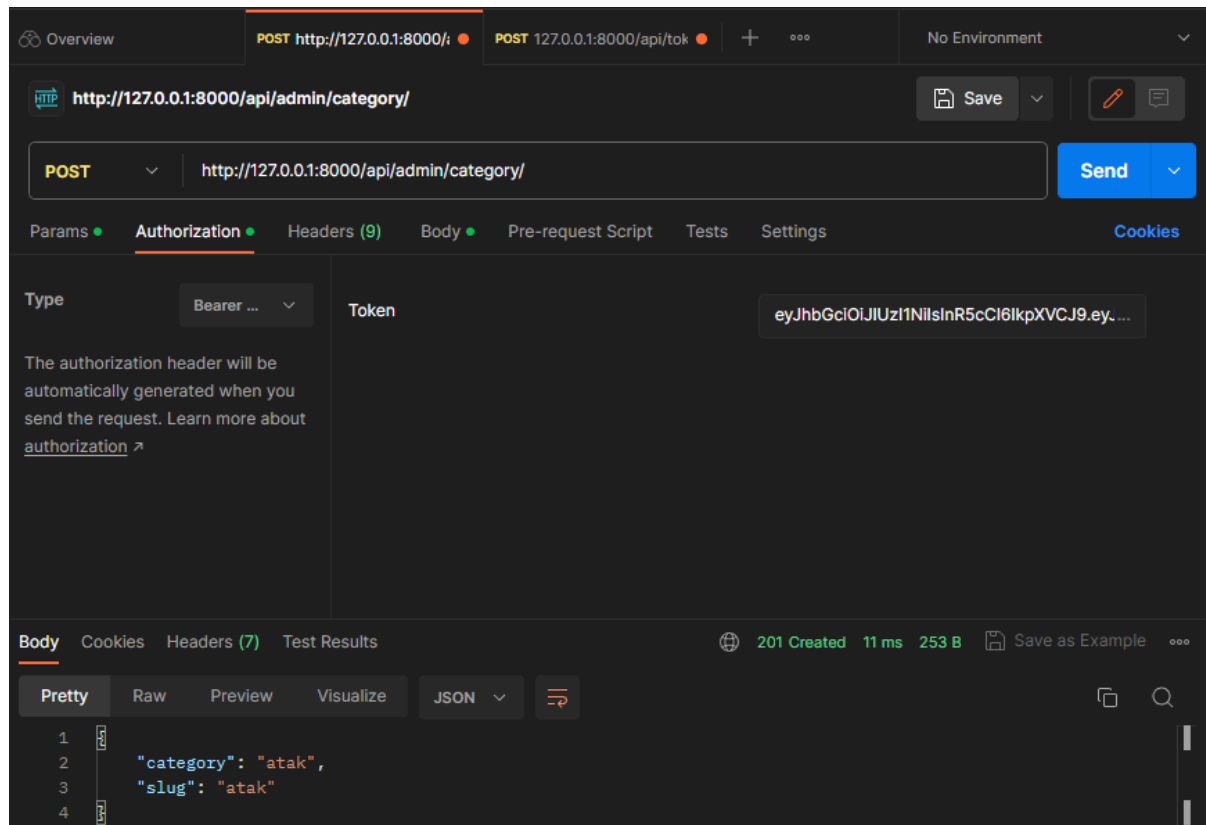
W aplikacji zaimplementowane jest proste API administracyjne, które pozwala na zarządzanie bazą danych - umożliwia ono dodawanie nowych danych do bazy, aktualizację oraz kasowanie wybranych rekordów. Dostęp do API jest zabezpieczony za pomocą tokenu JWT, aby uniknąć nieautoryzowanego dostępu.

Problem polega na tym, że dostęp do generowania tokenów JWT ma każdy, kto zna adres endpointu oraz posiada zarejestrowane konto w aplikacji. Niepotrzebne są żadne dodatkowe uprawnienia.

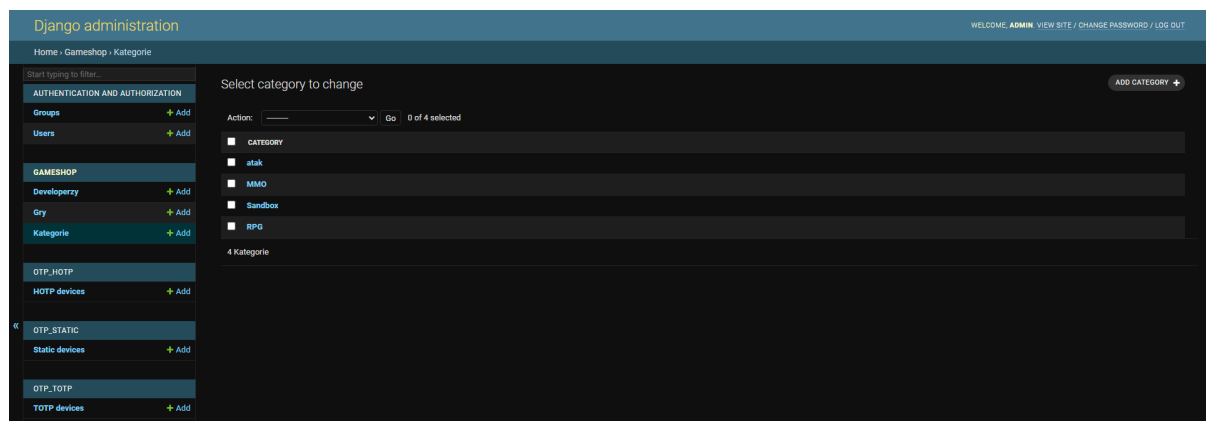
Dla przykładu, wygenerowany został token JWT dla użytkownika bez uprawnień:



Następnie, wygenerowany token został użyty do dodania nowej kategorii:



W panelu administracyjnym można zauważyć, że dodana została kategoria “atak”:



Sposób zabezpieczenia podatności:

Biblioteka Django API pozwala na proste zarządzanie permisjami dostępu do każdego punktu API. Do każdego endpointa należy przypisać odpowiednie permisje - w obecnym kodzie wszystkie endpointy mają źle zdefiniowane permisje:

```
class GameshopApiAddDeveloper(APIView):
    def post(self, request, *args, **kwargs):
        serializer = DeveloperSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data,
status=status.HTTP_201_CREATED)

        return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)
```

Poprawnie zabezpieczony endpoint powinien zawierać tablicę *permission_classes*, która jest wypełniona grupami, które mają dostęp do danego endpointu. Aby zapewnić dostęp tylko dla osób z uprawnieniami administratora, należy uzupełnić powyższą funkcję w następujący sposób:

```
class GameshopApiAddDeveloper(APIView):

    permission_classes = [permissions.IsAdminUser]

    def post(self, request, *args, **kwargs):
        serializer = DeveloperSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data,
status=status.HTTP_201_CREATED)

        return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)
```


5.1.4 Brak Walidacji danych

Brak walidacji danych to podatność bezpieczeństwa aplikacji internetowych, która pojawia się, gdy aplikacja nie sprawdza poprawnie danych wejściowych od użytkownika. Oznacza to, że aplikacja nie sprawdza dokładnie danych dostarczanych przez użytkownika pod kątem potencjalnie złośliwej zawartości, takiej jak ataki SQL Injection, Cross-Site Scripting (XSS) czy przepełnienia bufora.

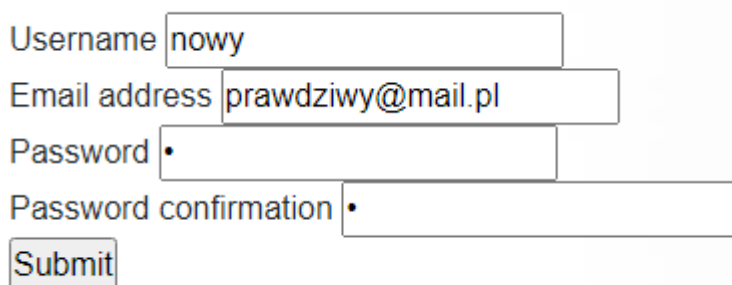
Podatność ta występuje najczęściej tam, gdzie aplikacja internetowa używa danych wejściowych od użytkownika w wygenerowanym przez siebie wyjściu bez ich walidacji lub kodowania. Brak walidacji danych może prowadzić do kradzieży wrażliwych informacji, instalacji złośliwego oprogramowania lub przejęcia kont użytkowników.

Przykład braku walidacji danych w naszej aplikacji:

Głównym miejscem, gdzie występuje brak walidacji danych, jest opcja wyszukiwania. W projekcie występują jeszcze dwa problemy z walidacją danych:

Brak walidacji haseł

Strona nie waliduje wprowadzanych haseł - nie sprawdza ich pod względem długości, złożoności i skomplikowania. Powoduje to, że użytkownicy mogą używać bardzo prostych haseł, zmniejszając bezpieczeństwo ich kont. Przy rejestracji nowego użytkownika można użyć pustego znaku jako hasło:



The image shows a registration form with the following fields and values:

- Username: nowy
- Email address: prawdziwy@mail.pl
- Password: (empty, indicated by a red dot)
- Password confirmation: (empty, indicated by a red dot)
- Submit button

Po kliknięciu Submit, użytkownik zostaje przeniesiony do weryfikacji OTP, co oznacza, że hasło zostało przyjęte:

Kod weryfikacyjny:

Aby zabezpieczyć użytkowników, wystarczy włączyć walidację haseł w ustawieniach Django - domyślnie funkcje walidacji są włączone. Dzięki walidacji haseł dopuszczane są tylko hasła spełniające określone warunki:

- odpowiednia długość (min. 8 znaków),
- nie pokrywa się z nazwą użytkownika / mailem,
- zawiera liczbę i znak specjalny,
- nie znajduje się na zdefiniowanej przez Django liście haseł prostych (np. admin123).

Domyślnie, wszystkie te walidatory są aktywne, jeżeli używany jest model autentykacji zaimplementowany w Django:

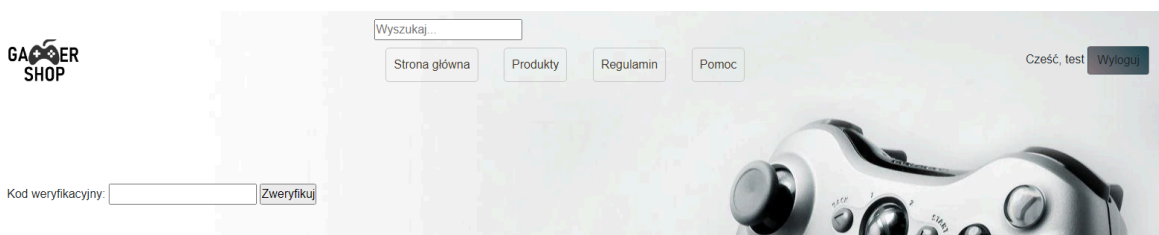
```
AUTH_PASSWORD_VALIDATORS = [
    {
        'NAME':
'django.contrib.auth.password_validation.UserAttributeSimilarityValid
ator',
    },
    {
        'NAME':
'django.contrib.auth.password_validation.MinimumLengthValidator',
    },
    {
        'NAME':
'django.contrib.auth.password_validation.CommonPasswordValidator',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

Nieprawidłowo zaimplementowana walidacja OTP

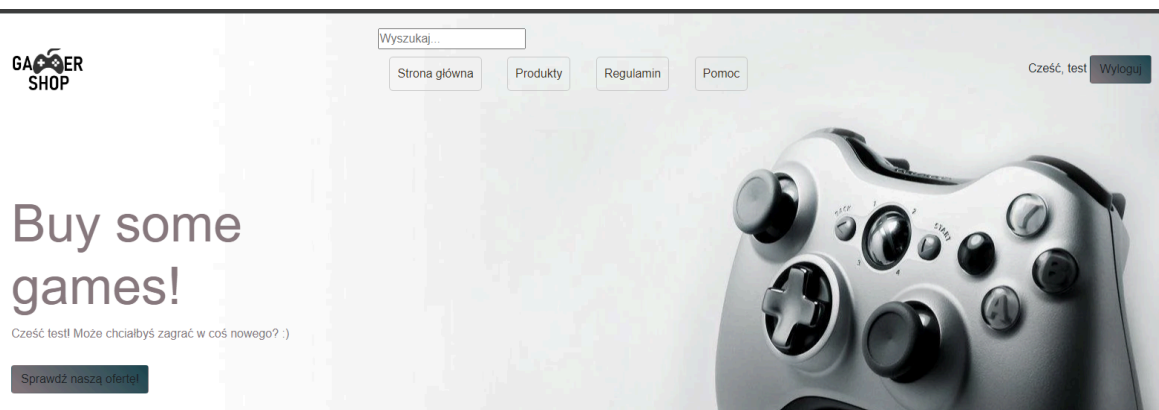
Aplikacja zabezpieczona jest kodami OTP - przy rejestracji oraz przy próbie logowania, użytkownik proszony jest o podanie jednorazowego hasła, aby potwierdzić swoją tożsamość. Rozwiązanie to znacznie zwiększa bezpieczeństwo konta użytkownika, nawet w przypadku, gdy niepowołane osoby uzyskają dostęp do danych logowania.

Niestety, kody OTP w aplikacji są zaimplementowane w zły sposób w przypadku rejestracji, jak i logowania. Konto użytkownika jest rejestrowane oraz logowane **przed** weryfikacją poprawności kodu OTP. Dzięki czemu, można uniknąć podawania kodu OTP w bardzo prosty sposób - wystarczy zmienić stronę na inną.

Można zauważyć, że strona pokazuje, że użytkownik jest zalogowany, nawet bez podania kodu OTP:



Po powrocie do strony głównej, użytkownik ma pełen dostęp do strony:



Błąd występuje w funkcji *userLogin*:

```
@require_http_methods(["GET", "POST"])
def userlogin(request):
    if request.method == 'POST':
        form = UserLoginForm(request.POST)
        if form.is_valid():
            username = form.cleaned_data['username']
            password = form.cleaned_data['password']
            user = authenticate(request, username=username,
password=password)
            if user is not None:
                login(request, user)
                otp_code = generate_otp_code()
                print(otp_code)
                subject = 'Kod weryfikacyjny OTP'
                message = f'Twój kod weryfikacyjny OTP to:
{otp_code}'

                from_email = 'noreply@semycolon.com'
                recipient_list = [user.email]
                send_mail(subject, message, from_email,
recipient_list, fail_silently=False)

                request.session['otp_code'] = otp_code

                return redirect('gameshop:otpaLogin')
            else:
                messages.info(request, 'Login lub hasło błędne')
        else:
            form = UserLoginForm()
            return render(request, 'gameshop/authenticate/login.html',
{'form': form})
```

Z kodu można zauważyć, że jeżeli podane zostaną dane do logowania, użytkownik jest najpierw logowany, a następnie dopiero generowany jest kod OTP. Sprawia to, że podanie kodu OTP nie jest wymagane, aby użytkownik został uznany za zalogowanego.

Jedną z możliwości naprawy kodu, aby kod OTP był faktycznie potrzebny do zalogowania, to usunięcie linijki *login(request, user)* z powyższej funkcji. Dane logowania powinny zostać przekazane do podstrony *otpLogin* w funkcji *return redirect('gameshop:otpLogin')* - ponieważ to na tej podstronie dopiero weryfikowany jest kod OTP.